Sender I/O: A Constructed Comparison

Document Number:	P4090R1
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

The sender composition algebra does not apply to compound I/O results without losing data, and the construction that preserves all data produces code identical to the coroutine version.

Four sender-based TCP echo servers are constructed from P2300R10 [1] and P3552R3 [2] and compared against a coroutine-native echo server. The sender composition algebra - `when_all` cancellation, `upon_error`, `retry` - does not apply to compound I/O results without losing data, requiring shared state, or converting routine errors to exceptions. One construction - "just use `set_value`" - preserves all data by bypassing the composition algebra entirely, producing code nearly identical to the coroutine version. Both paradigms are equivalent when compound results stay on the value channel. The composition algebra is the sender model's error-handling value proposition over coroutines - the reason to accept its additional complexity for I/O. If it does not apply to compound I/O results, the programmer pays the cost of the sender model and receives the coroutine model's error-handling behavior. Compile-time work graphs, lazy evaluation, and scheduling remain genuine sender additions. The finding is about cost, not defect.

Revision History

R1: May 2026 (pre-Brno mailing)

- Formatting corrections.

R0: April 2026 (post-Croydon mailing)

- Initial version.
- Revised prior to publication to incorporate reflector discussion with Petersen and Voutilainen. Renamed approaches A1/A2/B/C to "just use `set_value`" / "just split the result" / "just use `set_error`" / "just decompose it." Reframed Section 5

to acknowledge that "just use `set_value`" is the sender equivalent of the coroutine idiom. Added Section 5.1 (the equal-footing observation), Q11, Q12, Q13. Restructured the trade-off table with composition algebra rows and compile-time work graph row. Reframed the invitation to ask whether the composition algebra applies to compound I/O results. Expanded structured concurrency guarantees in Section 11. Added Disclosure paragraph on `std::execution` support. Added direct quotations from Petersen and Voutilainen with permission.

---

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++.

Vinnie 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. The Coroutine-Native Echo Server

Corosio [4] `do_session`:

```
capy::task<> do_session()
{
    for (;;)
    {
        auto [ec, n] = co_await sock_.read_some(
            capy::mutable_buffer(
                buf_, sizeof buf_));

        auto [wec, wn] = co_await capy::write(
            sock_, capy::const_buffer(buf_, n));

        if (ec || wec)
            break;
    }
    sock_.close();
}
```

Both values visible. No exceptions.

---

3. Why I/O Results Are Different

Infrastructure operations have binary outcomes:

Operation	Success	Failure
`malloc`	Block returned	Allocation failed
`fopen`	File handle returned	Open failed
`pthread_create`	Thread running	Creation failed
GPU kernel launch	Kernel running	Launch failed
Timer arm	Timer armed	Resource limit

Every row is binary. The three channels map unambiguously.

I/O operations return compound results:

Operation	Result
`read`	`(status, bytes_transferred)`
`write`	`(status, bytes_written)`

Every OS delivers both values together. [5] Chris Kohlhoff identified the consequence for senders in P2430R0 [6]:

> *"Due to the limitations of the `set_error` channel (which has a single 'error' argument) and `set_done` channel (which takes no arguments), partial results must be communicated down the `set_value` channel."*

Four approaches follow.

---

4. Constructing the Sender Echo Server

Each approach is constructed from P2300R10 [1] and P3552R3 [2].

Kühl enumerated five channel-routing options in P2762R2 [7] Section 4.2 and noted: "some of the error cases may have been partial successes...using the set_error channel taking just one argument is somewhat limiting." Shoop identified the same difficulty in P2471R1 [8]: completion tokens translating to senders "must use a heuristic to type-match the first arg."

P2300R10 [1] Section 4.14 demonstrates the pattern:

```
return read_socket_async(socket, span{buff})
    | execution::let_value(
          [](error_code err,
             size_t bytes_read) {
              if (err != 0) {
                  // partial success
              } else {
                  // full success
              }
          });
```

"Just use `set_value`" piped into a `let_value` decomposition. The example does not show what happens to `bytes_read` when the error must reach `set_error`. The specification's own motivating example for I/O is the approach that bypasses the composition algebra.

The constructions in Sections 5-8 are illustrative. Petersen provided four compilable implementations [9] demonstrating the same trade-offs (https://godbolt.org/z/7W51hYE7c). Voutilainen provided a compilable channel ping-pong example [9] (https://godbolt.org/z/h5cv5fbTE).

---

5. "Just Use `set_value`"

The I/O sender calls `set_value(error_code, size_t)` for all outcomes - route everything through the value channel. This paper calls this pattern "just use `set_value`." Structured bindings work (P3552R3 [2] Section 4.4):

```
auto do_session(auto& sock, auto& buf)
    -> std::execution::task<void>
{
    for (;;)
    {
        auto [ec, n] = co_await async_read(
            sock, net::buffer(buf));

        auto [wec, wn] = co_await async_write(
            sock, net::const_buffer(buf, n));

        if (ec || wec)
            break;
    }
}
```

Nearly identical to Corosio. Both values visible. No exceptions. Wrapping the pair in `std::expected<size_t, error_code>` is a variant of this approach - the compound result stays on the value channel as a single value. The C++23 monadic operations on `expected` (`and_then`, `or_else`, `transform`) provide value-channel composition for the wrapped result. This is the same pattern: the programmer inspects the compound result with value-level operations, not with channel-level algorithms. `upon_error` does not see inside the `expected`. `retry` does not fire on it. `when_all` does not cancel siblings. The `expected` approach is "just use `set_value`" with monadic syntax. It belongs in the first column of the trade-off table.

This is the sender instantiation of the industry advice documented in P4091R0 [10]: use the value channel whenever the result is not 100% failure. POSIX, Asio, Go, and Rust all follow this convention. The coroutine-native echo server (Section 2) does the same thing - it returns `(error_code, size_t)` through the value channel and inspects both with `if (ec)`.

The function signature says `std::execution::task<void>`. The body says `if (ec || wec) break`. The programmer is inside a sender coroutine, but the error handling is the coroutine model: structured bindings, `if`, `break`. No sender algorithm participates in the error decision. The sender machinery - `Environment`, `affine_on`, `AS-EXCEPT-PTR` - is present in the type but unused at the call site. To use it for I/O errors, the programmer must leave `if (ec)` and write pipes. Section 6 shows what that looks like: `co_await (async_read(...) | then([&](...) { ... }) | upon_error([&](...) { ... }))`. The function body now contains two programming models - `if (ec)` for the coroutine model and `| upon_error(...)` for the sender model - in the same scope. The unification promise holds for infrastructure operations where `co_await` hides the pipes. It breaks for compound I/O results where the pipes must be explicit.

What the Composition Algebra Does Not See

The I/O sender never calls `set_error`. The composition algebra is not engaged:

- `when_all` does not cancel siblings on I/O failure - the failure arrived through `set_value`.
- `upon_error` is unreachable for I/O errors.
- `retry` does not fire on I/O errors.

The coroutine-native echo server also does not use these facilities. It has no `upon_error`, no `retry`, no `when_all` cancellation on I/O failure. It uses `if (ec)`. The two approaches are equivalent.

5.1 The Equal-Footing Observation

Ian Petersen observed this symmetry on the LEWG reflector (March 14, 2026) [9]:

> *"Applying that decades-old industry advice to the sender sublanguage is easy: call `set_value` with an error code and your partially-successful value. If we're going to compare and contrast coroutines with senders, we should compare and contrast them on equal footing. Your coroutine examples don't use the coroutine error channel, but your assertions about the deficiencies of senders insist that sender code must use the sender error channel."*

The observation is correct. An earlier draft of this paper held the two paradigms to different standards - measuring coroutines by their natural idiom (value-based error handling) while measuring senders against channel-based composition that the coroutine baseline does not use. The authors are grateful to Petersen for identifying this.

"Just use `set_value`" works. It preserves both values, avoids exceptions, and produces code nearly identical to the coroutine version. On equal footing, the two paradigms are equivalent for compound I/O results.

That equivalence raises a deeper question. The sender model provides facilities the coroutine model does not: `when_all` sibling cancellation, `upon_error` handlers, `retry` policies, compile-time work graphs. These are the sender model's value proposition - the reason to accept its additional complexity over coroutines. If "just use `set_value`" is the correct approach for I/O, those facilities are unused for I/O errors. The composition algebra - the part that makes senders more than coroutines - does not apply to compound I/O results. The programmer who writes `auto [ec, n] = co_await async_read(...)` inside `std::execution::task` pays the cost of the sender model - the `Environment` template parameter, `AS-EXCEPT-PTR`, `affine_on`, the symmetric transfer gap - and receives the same behavior as a one-parameter coroutine-native task type with `if (ec)`.

The sender model under "just use `set_value`" does provide compile-time work graphs and lazy evaluation that the coroutine model does not. These are genuine additions. Scheduling algorithms - `continues_on`, `on`, `schedule` - also work with "just use `set_value`" because they transfer execution context without inspecting the value. The finding in this paper is not about the scheduling subset of the sender algebra. The question is whether the facilities most relevant to error handling - the composition algebra - apply to compound I/O results. Compile-time work graphs do not route errors. Lazy evaluation does not inspect byte counts. The composition algebra does, and it is the facility that does not apply.

Sections 6 through 8 explore what happens when the composition algebra is applied to compound I/O results. Each construction attempts to use the error channel. Each pays a cost.

---

6. "Just Split the Result"

Route the error code through `set_error`; capture the byte count in shared state. This paper calls this pattern "just split the result."

```
auto do_session(auto& sock, auto& buf)
    -> std::execution::task<void>
{
    for (;;)
    {
        std::size_t n = 0;

        co_await (
            async_read(sock, net::buffer(buf))
            | then([&](std::size_t bytes) {
                  n = bytes;
              })
            | upon_error(
                  [&](std::error_code ec) {
                      // ec visible, n stale
                  }));

        co_await (
            async_write(
                sock, net::const_buffer(buf, n))
            | then([&](std::size_t bytes) {
                  n = bytes;
              })
            | upon_error(
                  [&](std::error_code ec) {
                      // ec visible, n stale
                  }));
    }
}
```

All three channels in use. `upon_error` reachable. `when_all` cancels siblings. `retry` fires.

The Cost

The byte count bypasses the channels:

- `retry` fires on `set_error` but the byte count is in `n`, not in the error channel.
- `upon_error` receives `error_code` alone. `n` reflects the previous stage, not the failed one.
- **Shared mutable state** across continuation boundaries.

"Just use `set_value`" with the error code moved to `set_error` and the byte count moved to shared state.

---

7. "Just Use `set_error`"

`set_value(size_t)` on success, `set_error(error_code)` on failure. This paper calls this pattern "just use `set_error`." P3552R3 [2] converts `set_error` to an exception via `AS-EXCEPT-PTR` (the exposition-only function that converts `set_error` arguments to `exception_ptr`). No non-throwing path:

```
auto do_session(auto& sock, auto& buf)
    -> std::execution::task<void>
{
    try {
        for (;;)
        {
            auto n = co_await async_read(
                sock, net::buffer(buf));

            co_await async_write(
                sock, net::const_buffer(buf, n));
        }
    } catch (std::system_error const& e) {
        // ECONNRESET, EPIPE, EOF arrive here
    }
}
```

The Cost

Three specifications chain to produce the exception path. The I/O sender calls `set_error(ec)` (P2300R10 [1]). The sender-awaitable's `await_resume` converts it via `AS-EXCEPT-PTR` (P3552R3 [2]). `task`'s coroutine machinery rethrows. Each specification made a deliberate design choice. The consequence of the three choices combined is that every routine `ECONNRESET` becomes a thrown exception.

- **Byte count.** 500 of 1,000 bytes written before `ECONNRESET` - gone.
- **Non-throwing path.** Every `ECONNRESET` requires `make_exception_ptr` + `rethrow_exception`.
- **Visible error path.** The error hides in `catch`, separated from the `co_await` site.

---

8. "Just Decompose It"

Pipe "just use `set_value`" into `let_value`, inspect both values, re-route the error code to `set_error`. This paper calls this pattern "just decompose it."

```
#include <exec/variant_sender.hpp> // stdexec

async_read(sock, net::buffer(buf))
    | let_value(
          [](std::error_code ec, std::size_t n) {
              auto succ = [&] {
                  return just(n); };
              auto fail = [&] {
                  return just_error(ec); };
              using result =
                  exec::variant_sender<
                      decltype(succ()),
                      decltype(fail())>;
              if (ec)
                  return result(fail());
              return result(succ());
          })
    | upon_error([](std::error_code ec) {
          // reachable
      });
```

`exec::variant_sender` is from stdexec, not P2300R10 [1]. Even with `variant_sender` standardized, the data-loss problem persists: `just_error(ec)` carries only the error code (Q7). The return-type constraint and the data-loss constraint are independent.

The handler sees both values. Downstream, `upon_error` is reachable, `when_all` cancels siblings, `retry` fires.

The Cost

`just_error(ec)` carries only the error code. `set_error` takes a single argument (P2300R10 [1] [exec.recv.concepts]).

- `retry` never sees the byte count.
- `upon_error` receives `error_code` alone.
- **Partial write.** 500 of 1,000 bytes before `ECONNRESET` - gone once `just_error` emits.

"Just decompose it" is a hybrid of "just use `set_value`" and "just use `set_error`." The byte count is still lost on the error path.

---

9. The Trade-Off

"Just use `set_value`" and the coroutine-native approach are equivalent - both use the value channel, both preserve data, neither engages the composition algebra for I/O errors. The remaining three constructions attempt to use the composition algebra. Each pays a cost.

Property	"Just use `set_value`"	"Just split the result"	"Just use `set_error`"	"Just decompose it"	Corosio
Data preservation					
Partial write preserved	Yes	Yes	No	No (on error path)	Yes
Byte count in completion sig	Yes	No (side state)	No (discarded)	No (on error path)	Yes (return value)
Retry sees byte count	No	No	No	No	Yes (in scope)
Composition algebra					
Uses composition algebra for I/O	No	Yes	Yes	Yes	No
`when_all` cancels on I/O error	No	Yes	Yes	Yes	No
`upon_error` reachable for I/O	No	Yes	Yes	Yes	No
`retry` fires on I/O error	No	Yes	Yes	Yes	No
Channels used for I/O	1 of 3	3 of 3	3 of 3	3 of 3	Values (no channels)
Compile-time work graph	Yes (lazy)	Yes (lazy)	Yes (lazy)	Yes (lazy)	No
Static completion sig checking	Yes	Yes	Yes	Yes	No
Heterogeneous child composition	Yes	Yes	Yes	Yes	No
Error handling					
Error code at call site	Yes (`if (ec)`)	Yes	No (in `catch`)	Yes (in handler)	Yes (`if (ec)`)

Property	"Just use `set_value`"	"Just split the result"	"Just use `set_error`"	"Just decompose it"	Corosio
Byte count at call site	Yes	Yes	No (discarded)	Yes (in handler)	Yes
Exception on `ECONNRESET`	No	No	Yes	No	No
Shared mutable state required	No	Yes	No	No	No

The first and last columns are symmetric. Both use the value channel. Both preserve data. Neither uses the composition algebra for I/O errors. The three middle columns attempt to use the composition algebra and each pays a different cost: shared mutable state, exception round-trips, or data loss on the error path. Every construction that engages the composition algebra for I/O has a nonzero cost. The only construction without a cost - "just use `set_value`" - is the construction where the composition algebra does not participate. The cost of engaging the composition algebra for compound I/O results is nonzero. The benefit over `if (ec)` is zero.

Infrastructure operations face no such trade-off. Their outcomes are binary. A retry policy that distinguishes zero-progress failures from partial-progress failures needs the byte count - retrying a read that transferred zero bytes but not one that stalled after partial transfer.

When the Byte Count Determines Correctness

For composed operations (`async_read` with a completion condition), the byte count on error is often diagnostic - the application logs it but does not branch on it. For protocol-layer decisions and raw operations, the byte count determines correctness. The TLS `stream_truncated` case below is the clearest example. Partial-write recovery is another. The distinction matters: the byte count is not always decision-making data, but when it is, it must survive.

Many HTTP servers - including Google's - skip TLS `close_notify`. The composed read returns `(stream_truncated, n)`. If `n` equals `Content-Length`, the body is complete and the truncation is harmless. If `n` is less, the body is incomplete. The byte count determines correctness.

Coroutine-native:

```
auto [ec, n] = co_await tls_read(
    stream, body_buf);
if (ec == stream_truncated
    && n == content_length)
    ec = {};  // body complete, ignore truncation
```

"Just use `set_error`":

```
try {
    auto n = co_await tls_read(
        stream, body_buf);
} catch (std::system_error const& e) {
    // stream_truncated arrives here
    // n is gone - cannot check content_length
}
```

Under "just decompose it," the `let_value` handler sees both values - but only if it has the HTTP framing context. A generic TLS adapter does not. Once `just_error(stream_truncated)` emits, the byte count is gone.

This boundary is the **abstraction floor**:

Region	What the code sees
Above the floor	`error_code` alone - composition works
Below the floor	`(error_code, size_t)` - both values intact

The coroutine-native model's abstraction floor is `throw` - opt-in and crossed only by an explicit decision. The sender model's floor is `set_error` - required to engage the composition algebra.

An HTTP/2 multiplexer issuing concurrent reads via `when_all` faces the same table at every I/O boundary.

Where the Composition Algebra Does Apply

The finding in this paper is limited to the I/O layer, where results are compound. The composition algebra applies to protocol-layer binary outcomes. Retry a request, cancel a stream, timeout a connection - these are binary outcomes. The composition algebra handles them well.

A real networking application has both layers. The I/O layer produces `(error_code, size_t)`. The protocol layer consumes the I/O result, applies application logic, and produces a binary outcome (request succeeded / request failed). The composition algebra applies to the binary outcome, not to the compound I/O result.

The question is where the reduction from compound to binary happens. Under "just use `set_value`," it happens inside a `let_value` handler or a coroutine body - the same place it happens in coroutine-native code. The composition algebra takes over after the reduction. This is the abstraction floor (P4093R0 [11] Section 4): compound results below, binary outcomes above, composition algebra above the floor.

Both paradigms produce the binary outcome the composition algebra consumes. A coroutine body reduces `(error_code, size_t)` to a binary outcome with `if (ec)`. A `let_value` handler does the same. The composition algebra applies above the floor regardless of which paradigm produced the reduction below it. The sender model's additional complexity at the I/O layer does not change the binary outcome the protocol layer receives.

This paper does not argue that the composition algebra is useless for networking. It argues that the composition algebra does not apply to the I/O operations that produce compound results. Protocol-layer composition is orthogonal to this finding.

---

10. Qt Sender and Coroutine Side by Side

Ville Voutilainen's libunifex-with-qt [12] contains a chunked HTTP downloader written as both a sender pipeline and a coroutine.

Sender pipeline - "Just split the result":

```
auto repeated_get =
    let_value(
        just(req)
        | then([this](auto request) {
              return setupRequest(request,
                  bytesDownloaded, chunkSize);
          }),
        [this](auto request) {
            auto* reply = nam.get(request);
            return qObjectAsSender(reply,
                    &QNetworkReply::finished)
                | then([reply] {
                      return reply;
                  });
        })
    | then([this](auto* reply) {
          bytesDownloaded +=       // shared state
              reply->contentLength();
          reply->deleteLater();
      })
    | then([this] {
          return bytesDownloaded   // shared state
              == contentLength;
      })
    | repeat_effect_until();
```

Coroutine - both values visible:

```
exec::task<void> doFetchWithCoro()
{
    while (bytesDownloaded != contentLength)
    {
        req = setupRequest(req,
            bytesDownloaded, chunkSize);
        auto* reply = nam.get(req);
        co_await qObjectAsSender(reply,
            &QNetworkReply::finished);
        bytesDownloaded +=
            reply->contentLength();
        reply->deleteLater();
    }
}
```

Voutilainen's example is a demonstration of Qt/stdexec integration, not production code, and the authors are grateful for it - it is the only published side-by-side comparison of a sender pipeline and a coroutine performing the same work. The sender pipeline is "just split the result" in the trade-off table. The byte count is in `this->bytesDownloaded`, not in any channel. `repeat_effect_until` reads the data member directly. The coroutine version accesses the same data member, but the coroutine's sequential execution - guaranteed by the executor - serializes access to `bytesDownloaded`. The sender pipeline's lambda captures create aliasing across continuation boundaries that the type system does not prevent.

Asked whether the byte count could be made visible to downstream algorithms through the channel rather than through shared state, Voutilainen assessed the options on the LEWG reflector (March 14, 2026) [9]:

> *"the short answer is, 'yes, intrusively'. Not fully-generically."*

Voutilainen described several constructions - capturing the count in a successor's function object, using shared state, or passing data through the environment - and characterized each as "limitedly-general and slightly intrusive." The assessment is consistent with the trade-off table: making the byte count visible to the composition algebra requires moving it outside the channel.

---

11. Structured Concurrency

Capy [3] provides `when_all` and `when_any` as coroutine-native primitives with structured cancellation: child operations complete before the parent continues, and stop tokens propagate through `io_env`.

```
capy::task<dashboard> load_dashboard(
    std::uint64_t user_id)
{
    auto [name, orders, balance] =
        co_await capy::when_all(
            fetch_user_name(user_id),
            fetch_order_count(user_id),
            fetch_account_balance(user_id));
    co_return dashboard{name, orders, balance};
}
```

```
capy::task<> timeout_a_worker()
{
    auto result = co_await capy::when_any(
        background_worker("worker"),
        capy::delay(100ms));

    if (result.index() == 1)
        std::cout << "Timeout fired\n";
}
```

Both models provide structured concurrency. In the sender model, the operation state protocol - formalized by `async_scope` (P3149R9 [13]) - guarantees that child operation states are destroyed before the parent's receiver is called. In the coroutine model, `capy::when_all` and `capy::when_any` guarantee the same property through coroutine frame lifetimes: child coroutines complete and their frames are destroyed before the parent coroutine resumes. Stop tokens propagate through `io_env` at `await_suspend` time. The mechanism differs - operation state protocol vs. coroutine frame scoping - but the structured concurrency guarantees are equivalent: no child outlives its parent, cancellation propagates downward, and results are available only after all children complete. The sender `when_all` additionally provides compile-time work-graph visibility, static type checking of completion signatures, and heterogeneous child composition (GPU + network + timer in one expression) that the coroutine `when_all` does not.

They differ in where the compound result is visible when the composition decision is made (Q8, Q10). The trade-off table (Section 9) applies at every I/O boundary inside a structured concurrency scope.

---

12. Frequently Raised Concerns

Q1: Is the echo server too minimal to be representative?

The echo server is deliberately minimal. The compound-result problem is per-operation: adding protocol complexity adds more call sites with the same trade-off, not a different one. The sender model's composition strengths are orthogonal to the channel-routing decision each I/O operation must make. If the committee believes a more complex pipeline would change the finding, we invite the sender community to provide one. We will construct the comparison and publish the results.

Q2: Are these gaps being addressed by ongoing work?

Several of the ergonomic issues documented here are the subject of active committee work, including P3796R1 [14] and P3980R0 [15]. The authors welcome that work. Voutilainen confirmed on the LEWG reflector (March 14, 2026) [9] that one needed facility is not yet in the standard:

> *"We do not need the dispatch() I spoke about before, but we do need the variant_sender."*

`exec::variant_sender` exists in stdexec but is not part of P2300R10 [1]. Even with `variant_sender` standardized, the trade-off documented in Section 9 is structural at two levels. First, `set_error` takes a single argument; no iteration of the sender algorithms can deliver both the error code and the byte count through `set_error` without changing the channel model itself. Second, even if `set_error` accepted compound results, the I/O sender must still choose which channel to call - and that choice is context-dependent. `ECONNRESET` is fatal in an HTTP request handler but expected in a long-polling connection. The classification that determines the channel is application logic, not I/O logic. No context-free channel assignment is correct for all protocols. Ergonomic improvements to `let_value`, `variant_sender`, or `task` do not alter either constraint.

Q3: Could the byte count be stored in the operation state instead of shared state?

A variant of "just split the result" stores the byte count in the operation state rather than a local variable, scoping the lifetime to the operation. This localizes the lifetime hazard and is a meaningful improvement. The structural problem is unchanged: the byte count remains outside the completion signature, invisible to `retry`, `upon_error`, and every generic algorithm that operates on channel data.

Q4: Could the entire compound result be sent through `set_error`?

Sending `tuple<error_code, size_t>` through `set_error` preserves both values but changes the completion signatures expected by generic sender algorithms. `retry` fires on a tuple. `upon_error` receives application data. Asked whether `when_all` works correctly when a child sends a compound result through `set_error`, Voutilainen replied on the LEWG reflector (March 14, 2026) [9]:

> *"Yes, use an algorithm other than `when_all`, so that it doesn't cancel the others on one error, and collects all results."*

Voutilainen demonstrated this approach in a compilable channel ping-pong example [9] (https://godbolt.org/z/h5cv5fbTE) that sends the full tuple through `set_error`, preserves both values, and routes them back to the value channel downstream. This is the closest construction to satisfying the invitation's constraints. It preserves data, avoids exceptions, and avoids shared state. The remaining cost: the standard `when_all` cancels siblings on `set_error` regardless of the error type, and downstream algorithms must use `if constexpr` guards to distinguish the tuple from `std::exception_ptr`. Voutilainen's construction works within the three-channel model. It changes what the error channel carries, which changes the contract downstream algorithms expect. If the committee wishes to pursue that direction, it deserves its own paper and its own design review.

Q5: Do coroutines provide structured concurrency?

Yes. See Section 11.

Q6: Is the invitation constructed to be unanswerable?

The invitation asks whether the three-channel model can represent compound I/O results without loss, using the model's own facilities and semantics. The constraints - preserve data, retain composition, do not alter channel semantics, use specified facilities - are the properties the model claims to provide. If no construction can satisfy all of them simultaneously, that is the finding. The authors will incorporate any construction that does and re-evaluate every finding in this paper.

Q7: Does "just decompose it" work if the return-type constraint is solved?

`let_value` requires its callable to return a single sender type. `just(n)` and `just_error(ec)` are different types. Section 8 solves this with `exec::variant_sender` (stdexec). Suppose the return-type constraint is solved in the standard (for example, by a future `variant_sender`). The byte count is still lost: `just_error(ec)` carries only the error code. `set_error` takes a single argument. Solving the return-type problem does not solve the data-loss problem.

Q8: What does the coroutine-native approach lose?

Compile-time work graphs, lazy pipeline evaluation without coroutine frames, and generic algorithms over heterogeneous sender types. These are real costs. They are the sender model's strengths. This paper does not argue that coroutines replace senders. It argues that compound I/O results do not fit three mutually exclusive channels. The sender model serves its domain. The finding is about domain, not defect.

Q9: Does the coroutine-native model provide composition?

`when_all` and `when_any` (Section 11). Both values visible inside the combinator. The sender `when_all` cannot see the byte count because the channel split already happened. The coroutine `when_all` can, because the coroutine body inspects `(ec, n)` before the composition decision is made. Both models provide structured concurrency. They differ in where the compound result is visible.

Q10: Does the sender model compose across execution contexts in ways coroutines cannot?

Yes. Compile-time work graphs connecting GPU dispatch, thread pool submission, and event loop scheduling in a single statically typed pipeline - that is the sender model's domain and its genuine strength. Q8 acknowledges the cost. The compound-result problem is orthogonal. A GPU kernel launch is a binary outcome. A `read` is not. The sender model's cross-context composition works because its target operations are infrastructure operations with binary outcomes. The finding in this paper is about the operations that are not binary. This paper identifies the domain boundary.

Q11: Does "just use `set_value`" answer the comparison?

For data preservation, yes. "Just use `set_value`" preserves both values, avoids exceptions, and produces code nearly identical to the coroutine version. On equal footing, the two paradigms are equivalent.

For the composition algebra, no. The composition algebra - `when_all` cancellation, `upon_error`, `retry` - is the sender model's value proposition over coroutines. Under "just use `set_value`," those facilities are unused for I/O errors. If the composition algebra does not apply to compound I/O results, the sender model's additional complexity over coroutines buys nothing for I/O error handling in this domain. Scheduling, work graphs, and context transfer remain genuine additions (Q8, Q10). The domain-boundary finding is about the composition algebra, not the entire sender model.

Petersen provided four working sender implementations on the LEWG reflector (March 14, 2026) [9] and confirmed their equivalence to the coroutine idiom. Asked whether all four are equivalent to `auto [ec, buf] = co_await read(socket, buffer); switch (ec) { ... }`, Petersen replied:

> *"In one word, yes."*

The equivalence is the finding. Both paradigms handle compound I/O results the same way - through values.

Q12: Is this paper applying a double standard?

An earlier draft measured coroutines by their natural idiom (value-based error handling with `if (ec)`) while measuring senders against channel-based composition (`upon_error`, `retry`, `when_all` cancellation) that the coroutine baseline does not use. Petersen identified this asymmetry [9]:

> *"The data is only 'just there' in a coroutine if you collapse everything into the value channel, like the coroutine examples in your papers. Using the error channel (`set_error` in senders, `throw` in coroutines) requires a side channel if you want to convey both error and value at once."*

This revision corrects the asymmetry. Both paradigms are measured by the same ruler: the value-channel approach works for both. The remaining question is whether the sender model's composition algebra - the facilities that go beyond what coroutines provide - applies to compound I/O results. Sections 6 through 8 explore that question. Each construction that

attempts to use the composition algebra pays a cost. That is not a double standard. It is measuring the composition algebra against its own claims.

Q13: Does the composition algebra apply to protocol-layer decisions?

Yes. See Section 9, "Where the Composition Algebra Does Apply."

Q14: Could the sender model be extended for compound results?

Petersen proposed on the LEWG reflector (March 14, 2026) [9] that the findings in this paper could motivate new sender algorithms designed for compound results - an `error_code`-sensitive `when_all`, an `error_code`-aware `retry`, or adapters that bridge networking pipelines to the existing error-channel-based algorithms. This is a legitimate direction. The findings in this paper do not foreclose it. If such algorithms are designed and they satisfy the invitation's constraints, the finding changes. The authors welcome that work and will re-evaluate. The question is timing: should the standard ship the current algorithms for networking before the compound-result-aware algorithms are designed, or should both iterate together?

---

13. Invitation

"Just use `set_value`" answers the data-preservation question. Both values survive. No exceptions. The code is nearly identical to the coroutine version. The authors accept this.

The remaining question is whether the sender composition algebra applies to compound I/O results. The composition algebra - `when_all` sibling cancellation, `upon_error`, `retry` - is the sender model's value proposition over coroutines. If it applies to I/O, the sender model provides something coroutines do not. If it does not, the two models are equivalent for this domain.

Construct a sender-based echo server that uses the composition algebra for I/O errors and:

- preserves compound I/O results on the error path (the byte count survives into `upon_error`, `retry`, and `when_all`),
- does not alter the completion signatures expected by generic sender algorithms from those algorithms' specified behavior,
- avoids exception round-trips for routine error codes, and
- avoids shared mutable state across continuation boundaries.

If no such construction exists, the composition algebra does not apply to compound I/O results, and the sender model's additional complexity over coroutines buys nothing for I/O error handling. Scheduling, work graphs, and context transfer remain genuine additions. The coroutine-native approach loses compile-time work graphs and lazy pipeline evaluation (Q8). The sender approach loses composition algebra applicability for compound I/O results. Both costs are real. The question is which cost is higher for networking. That finding identifies the domain boundary.

The authors will incorporate any such construction and re-evaluate every finding.

---

14. Acknowledgments

The authors thank Ian Petersen for identifying an asymmetry in an earlier draft, for providing four working sender implementations that clarified the equal-footing observation, and (with Jessica Wong and Kirk Shoop) for `async_scope` - his critique materially improved this paper; Ville Voutilainen for working through the dispatch pattern, the channel ping-pong construction, and the `variant_sender` analysis with characteristic generosity and precision; Jens Maurer for reflector discussion on design freedom inside sender chains; Dietmar Kühl for the channel-routing enumeration in P2762R2 [7] and for `beman::execution`; Chris Kohlhoff for identifying the partial-success problem in P2430R0 [6]; Kirk Shoop for the completion-token heuristic analysis in P2471R1 [8]; Peter Dimov for the refined channel mapping in P4007R3 [16], "Open Issues in `std::execution::task`"; Michał Dominiak, Eric Niebler, and Lewis Baker for `std::execution`; Fabio Fracassi for P3570R2 [17], "Optional variants in sender/receiver"; and Herb Sutter for identifying the need for tutorials and constructed comparisons.

Any person quoted in this paper who believes their words have been presented out of context or who wishes a quotation removed may contact the authors, who will comply without question.

References

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

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

[3] cppalliance/capy - Coroutine I/O primitives library.

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

[5] IEEE Std 1003.1-2024 - POSIX `read()` / `write()` specification

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

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

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

[9] LEWG reflector (lib-ext), March 2026. Threads: "Complicated success at coroutine/sender composition boundaries" (https://lists.isocpp.org/lib-ext/2026/03/31375.php) and "std::execution - dynamically selecting a channel" (https://lists.isocpp.org/lib-ext/2026/03/31377.php). Compilable examples: Petersen's four sender implementations (https://godbolt.org/z/7W51hYE7c) and Voutilainen's channel ping-pong (https://godbolt.org/z/h5cv5fbTE).

[10] P4091R0 - "Two Error Models" (Vinnie Falco, 2026).

[11] P4093R0 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[12] Ville Voutilainen, libunifex-with-qt - Qt/stdexec integration examples (2024).

[13] P3149R9 - "`async_scope` - Creating scopes for non-sequential concurrency" (Ian Petersen, Jessica Wong, Kirk Shoop, et al., 2025).

[14] P3796R1 - "Coroutine Task Issues" (Dietmar Kühl, 2025).

[15] P3980R0 - "Task's Allocator Use" (Dietmar Kühl, 2026).

[16] P4007R3 - "Open Issues in `std::execution::task`" (Vinnie Falco, Mungo Gill, 2026).

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