A Reader's Guide to the May 2026 Mailing

Document Number:	P4170R0
Date:	2026-05-31
Intent:	Inform
Audience:	LEWG

Reply-to: Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

Thirty-four papers audit two decades of async decisions, deploy coroutine-native I/O at a derivatives exchange with zero per-operation allocations, bridge the result to std::execution, and deliver open-source AI evaluation tools for WG21 proposals.

This paper summarizes 34 papers published in the May 2026 mailing. It is a reading guide: an executive summary that identifies the logical series within the collection, describes what each series delivers, and provides individual summaries of every paper. It asks for nothing.

---

1. Disclosure

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

This paper asks for nothing.

---

2. Executive Summary

Eight papers assemble a complete coroutine-native I/O architecture from first principles to production deployment. P4100 lays the roadmap - fourteen papers in two stages, seven pure C++20 abstractions and seven platform I/O facilities, each independently shippable. P4003 defines the minimal execution model in three requirements (executor affinity, stop propagation, allocator delivery), shipping on three platforms with a recycling frame allocator measured at 3.1x throughput over std::allocator on MSVC. P4088 traces the causal chain from a single frame allocation through type-erased streams, separate compilation, and ABI stability - properties the sender model matches only at per-operation allocation cost. P4125 reports qualitative findings from a derivatives exchange whose platform architects called the library a viable production replacement after two weeks of migration, with recursive callback patterns collapsing into simple coroutine loops. P4126 proposes the callback handle - twenty-four bytes of stack storage presenting a coroutine-frame layout - giving senders zero-allocation access to every awaitable in the ecosystem. Together, these papers deliver what no single paper can: a working system from language mechanism through library design through production validation.

Ten papers conduct a structural analysis of std::execution's cost model for I/O workloads. P2583 traces stack overflow in co_awaited task loops to two protocol choices - void-returning completions and non-coroutine composition structs - and proposes a coroutine_handle-returning fix that touches four concepts and twenty-five algorithms. P4123 grants task every proposed fix and still finds forty additional spec-mandated operations per HTTP request-response session, a lost symmetric transfer path, and one heap allocation per type-erased I/O operation. P4090 constructs four sender-based TCP echo servers and demonstrates that the only construction preserving all data produces code identical to the coroutine version. P4091 names the abstraction floor - the boundary where compound results reduce to a single channel - and proves no position in the sender algebra achieves data preservation, composition participation, and genericity simultaneously. P4089 documents that seven of nine surveyed coroutine libraries converged on single-parameter task while std::execution::task's second Environment parameter produces N incompatible types with no general conversion. The compound finding across this cluster is precise: the sender composition algebra is the model's value proposition, and for the I/O operations that produce error_code paired with a byte count, it does not apply.

Seven papers reconstruct the decision chain that kept networking out of C++ for twenty-one years. P4094 audits P0443's fourteen revisions and finds the entire evidentiary basis for unifying networking, GPU dispatch, and thread pool executors is a single hypothetical code snippet. P4095 re-examines the Cologne pivot paper and shows that three of four deficiencies vanish under the continuation framing of execute(F&&). P4097 searches the published record surrounding the October 2021 poll and finds zero networking evidence behind the word "basis" at the time LEWG voted. P4047 grades twenty-seven dated predictions from both proponents and critics, finding eighteen confirmed - safety concerns and networking gaps proved highly predictive; universality claims did not. Read together, these papers document a recurring structural gap: decisions achieved consensus without published evidence in the domain those decisions governed, and no process mechanism required that both analytical framings be examined before the committee acted.

Three papers introduce AI-assisted evaluation methodologies and apply them to active proposals. P4046 (SAGE) distills five structured interviews with veteran committee members into an eleven-principle scoring rubric, then grades the lead author's own paper to demonstrate the pipeline. P4207 introduces the Advocatus Diaboli - a red-teaming framework that killed bad findings through six named challenges before they reach the record - and P4208 applies it to P2900R14, where fourteen of

fifteen charges collapse under cross-examination. Every tool ships under CC0 with full source; the value of the cluster is not any single verdict but a reproducible evaluation infrastructure the committee did not previously possess.

Two papers deliver bidirectional bridges between the coroutine and sender worlds. P4092's await_sender consumes any std::execution sender from inside a coroutine with inline operation state, zero heap allocation beyond the coroutine frame, and automatic error_code inspection that produces io_result values instead of exceptions. P4093's as_sender produces senders from awaitables with a static_assert that fires at compile time when compound results would lose data crossing the abstraction floor. Both implementations print in full in their appendices, compile today on MSVC 19.43 against Capy and beman::execution, and position the coroutine body as the translation layer that the sender algebra structurally cannot be.

Four papers address design principles that cut across the async debate. P4035 identifies the escape-hatch pattern the standard library has applied since condition_variable and proposes it for cstring_view. P4036 demonstrates that six independent I/O ecosystems rejected a generic byte view in favor of a dedicated buffer type and argues the same principle applies one level higher. P4014 delivers a complete progressive tutorial covering all thirty sender algorithms in C++26, from just(42) to a four-layer pipeline, dedicated to the public domain under CC0. P4182 consolidates seven platform categories and six compiler families into structured tables so future authors can cite one paper number instead of writing three paragraphs of platform context.

The full collection yields a qualitatively different picture than any subset. Individual papers deliver specific tools - a bridge adapter, a scoring checklist, a benchmark table, a field interview report. Clusters deliver compound understanding - why the composition algebra fails for I/O, how four decisions chained into a decade without networking, what a complete coroutine-native architecture looks like from language mechanism through type-erased ABI. But reading across clusters reveals the unifying thread: every retrospective finding motivates a specific architectural choice in the constructive papers, every cost analysis has a corresponding zero-cost implementation in the bridge or architecture papers, and every AI evaluation tool is demonstrated against a live proposal before being offered to the committee. The thirty-four papers form a single argument assembled from independently verifiable parts.

Three entry points serve different readers. A committee member evaluating the C++29 networking direction should begin with P4100, which maps the full fourteen-paper series and its two-stage delivery plan. An implementer or library author interested in the sender-coroutine boundary should start with P4092 and P4093, whose complete implementations compile today and whose abstraction-floor enforcement is immediately applicable. A reader interested in AI-assisted committee process - or looking for tools to apply to unrelated proposals - should begin with P4207, which introduces the red-teaming methodology and includes the most dramatic demonstration: a dollar in API tokens reproducing the substantive national body comments on a three-year proposal.

---

3. Individual Papers

3.1. P2583R4 - Symmetric Transfer and Sender Composition

A for-loop that co_awaits `std::execution::task` can overflow the stack because the sender model structurally prevents the C++20 mechanism that every major coroutine library adopted to make this impossible. The paper traces the gap to two protocol choices: sender algorithms compose through non-coroutine structs with no `coroutine_handle<>`, and completion functions return `void`, so even coroutine-backed receivers cannot propagate a handle upstream. A protocol-level fix is proposed and drafted: completion functions and `start()` return `coroutine_handle<>` instead of `void`, struct receivers forward handles from downstream without becoming coroutines, and zero-allocation composition is preserved. The scope is pervasive - four concept-level expressions, twenty-five sender algorithms, every third-party receiver - and the change becomes ABI-breaking once `std::execution` ships with void-returning completions.

3.2. P4003R3 - A Minimal Coroutine Execution Model

The claim that senders provide structured concurrency and coroutines do not has it backwards - P4003R3 argues that the IoAwaitable protocol is the very layer from which structured concurrency constructs are assembled. The protocol answers the three questions every suspended coroutine must resolve - executor affinity, stop token propagation, and frame allocator delivery - and nothing more. A complete implementation ships on three platforms via Capy and Corosio, with benchmarks showing a recycling frame allocator at 3.1x over std::allocator on MSVC and zero per-operation heap allocations under type erasure where exec::as_awaitable requires one per connect. Positioned as a companion to std::execution rather than a replacement, the paper asks LEWG to advance a coroutine vocabulary so minimal that nothing in it can be removed.

3.3. P4007R3 - Open Issues in `std::execution::task`

Four design decisions in `std::execution::task` become permanent the moment C++26 ships - and this paper names them. P4007R3 catalogues every open issue in the proposed coroutine task type, classifies each as fixable post-ship or foreclosed by shipping, and tracks which ones Croydon resolved. The good news is substantial: Croydon closed fourteen issues across scheduler affinity, allocator injection, and bulk execution. The foreclosed list is shorter and harder - frame allocator propagation through coroutine call trees, `co_yield`-based error return instead of `co_return`, and the absence of symmetric transfer are all locked in once the `void`-returning completion protocol ships. The paper asks for nothing; it simply makes the cost of shipping visible.

3.4. P4014R2 - The Sender Sub-Language For Beginners

All thirty sender algorithms in C++26 now have a single progressive tutorial that walks from `just(42)` to a four-layer sensor-fusion pipeline and back again - with every algorithm demonstrated, explained, and mapped to its plain-C++ equivalent. P4014R2 is a complete rewrite that covers value lifting, monadic composition, execution contexts, structured concurrency, data parallelism, async scopes, signal adaptation, the `task` coroutine type, and sender-coroutine interleaving, grounding each concept in its theoretical origin from the Lambda Papers through algebraic effects. It draws working real-world examples from stdexec and sender-examples - including a recursive backtracker and a full `retry` algorithm implementation - then dedicates the result to the public domain under CC0 so anyone can reuse it as documentation or teaching material. For the committee participant who wants the entire sender algebra in one place, or the library author implementing custom algorithms, this is the reference that did not previously exist.

3.5. P4035R1 - The Need for Escape Hatches

Eliminating the unsafe path does not eliminate unsafe usage - it pushes programmers toward worse tools. P4035R1 identifies a design principle the standard library has applied informally since `condition_variable` but never articulated: safe interfaces carry the default name and a wide contract; escape hatches carry a marked name and a narrow contract. The paper traces the pattern through four independent codebases - Boost.URL's `pct_string_view`, BSD directory iteration, the Capy coroutine library, and `condition_variable` itself - then applies it directly to the `cstring_view` constructor debate, proposing a validating default alongside a static `unsafe` factory at zero runtime cost. Two LEWG polls from Sofia and Croydon already rejected closing the implicit `char const*` path; this paper supplies the design vocabulary those votes were reaching for.

3.6. P4036R0 - Why Span Is Not Enough

Six I/O ecosystems - POSIX, Windows, Asio, libuv, Go, and .NET - were designed independently, and every single one rejected a generic byte view in favor of a dedicated buffer type. P4036R0 dissects why `span<byte>` fails as an I/O buffer descriptor: it cannot represent scatter/gather sequences without dangling, it cannot consume bytes at parse boundaries that do not align with buffer boundaries, and its interface is too broad to enable the diagnostic hooks that a bespoke type permits. The paper traces the precedent back to P0298R3, which introduced `std::byte` for the same reason - the generic type's operations did not match the domain - and argues the identical principle applies one abstraction level higher.

3.7. P4041R0 - Is `std::execution` a Universal Async Model?

The reference implementation of `std::execution` has 1,300 stars; the entire sender-based I/O ecosystem built on top of it has 21. This paper assembles the complete post-adoption record - twenty-four changes to the sender sub-language since St. Louis, zero to coroutines - alongside an ecosystem survey that finds no publicly verifiable sender-based production deployment in TCP networking, HTTP, TLS, databases, DNS, or signal handling. Published findings from committee members document stack overflow risks in `std::execution::task`, an allocator model reworked six months after adoption, and a partial-success channel tension first raised in 2020 and still open. The framework's own architect placed ninety percent of async code in the coroutine column; six independent libraries converged on `coroutine_handle<>`-returning `await_suspend` while the standard bridges use `void`.

3.8. P4046R0 - SAGE: Saving All Gathered Expertise

The authors built an AI pipeline that distills veteran committee judgment into a paper-scoring rubric - and the first paper they graded was their own. Five structured interviews with experienced WG21 participants (including Hinnant, Abrahams, Parent, and Gregor) were transcribed, fed through a three-stage agentic workflow (capture, merge, judge), and reduced to eleven corroborated principles such as "standardize proven practice" and "complexity that only grows kills a language." The resulting evaluation model scored the lead author's P4003R3 at 17 out of 22, identifying concrete weaknesses - single-organization development, missing license documentation - that could be fixed before committee review. All transcripts, knowledge files, rule prompts, and evaluations ship as an open GitHub repository; anyone can re-run the pipeline or extend it with new interviews.

3.9. P4047R0 - CRYSTAL BALL: Checking Predictions Against the Record

Eighteen of twenty-seven public predictions about `std::execution` confirmed when checked against the record - but the two sides were right about very different things. The paper collects every dated, falsifiable claim it can find from proponents and critics alike - sourced to committee papers, blog posts, and conference talks spanning 2019 through 2025 - and grades each as Confirmed, Unconfirmed, Partially Confirmed, or Overtaken by Events. Safety concerns, domain viability, networking gaps, and implementation maturity warnings all proved highly predictive; universality claims and customization mechanism design did not. The scorecard is a five-year forensic audit of committee debate that makes visible which categories of prediction aged well and which did not.

3.10. P4048R0 - Networking for C++29: A Call to Action

Twenty-one years after the first networking proposal, this paper argues that the committee already has the talent and the implementations - what it has never had is a production process. P4048R0 proposes the first continuous pipeline for WG21: six teams - four sequential (Implementation, Design, Wording, Testing) and two cross-cutting review teams for architectural integrity and `std::execution` compatibility - pushing eleven papers through LEWG by end of 2028, with every handoff producing a visible artifact in a public GitHub repository. The eleven papers divide into two stages: four pure C++20 abstractions with no platform dependency, then seven platform I/O papers - and the pipeline is designed so that each completed paper is a shippable unit, making partial delivery an explicit feature of the design.

3.11. P4088R1 - What C++20 Coroutines Already Buy The Standard

The one allocation every design avoids - the coroutine frame - is the one that pays for everything networking has needed for twenty-one years. P4088R1 traces a nine-step causal chain from a single heap allocation through concrete operation states, type-erased streams, separate compilation, and ABI stability, demonstrating that five C++20 language mechanisms - none designed for I/O - combine into a substrate that resolves the template explosion, compile-time cost, and ABI instability that stalled every prior networking proposal. Benchmarks over 100 million read operations measure the price: under type erasure, coroutine awaitables add 5 ns and zero allocations per operation, while sender pipelines add 23 ns and one allocation per operation. The paper concedes all six known objections to the coroutine model, then shows that the frame's structural type erasure through `coroutine_handle<>` produces a property set - zero-cost type-erased streams, separately compiled I/O algorithms, transport-independent ABI - that the sender model can match only at per-operation allocation cost.

3.12. P4089R1 - On the Diversity of Coroutine Task Types

Seven of nine surveyed coroutine libraries converged on `task<T>` with a single template parameter; the C++26-approved `std::execution::task<T, Environment>` is one of only two that did not. This paper traces the consequences of that second parameter through the P2300 specification, escalating from trivial empty environments through standard type mismatches to NVIDIA's deployed GPU environment - complete with CUDA streams, pinned memory resources, and stream pools - showing that no general conversion exists and that `write_env` requires the caller to know every missing query by name. Five StackOverflow reports document the same interoperability failure in Boost.Asio, the only production precedent for a two-parameter coroutine type, while four working cross-library bridges (51 to 105 lines each) demonstrate that the one-parameter design already interoperates through the C++20 awaitable protocol. The structural finding reduces to two lines of specification: `queryable` is `destructible`, it cannot be narrowed after C++26 ships, and N libraries with N environments will produce N incompatible task types with no general conversion path.

3.13. P4090R1 - Sender I/O: A Constructed Comparison

Four sender-based TCP echo servers are constructed from P2300R10 and P3552R3, and the only one that preserves all data produces code identical to the coroutine version. The paper demonstrates that the sender composition algebra - `when_all` cancellation, `upon_error`, `retry` - does not apply to compound I/O results without losing the byte count, requiring shared mutable state, or converting routine `ECONNRESET` into thrown exceptions. Each construction is laid out with compilable Godbolt links from Petersen and Voutilainen, a five-column trade-off table scoring data preservation and algebra engagement, and a TLS `stream_truncated` case where the lost byte count determines correctness. The finding is precise: the composition algebra is the sender model's value proposition over coroutines, and for the I/O operations that produce `(error_code, size_t)`, it does not apply - the programmer pays the cost of the sender model and receives the coroutine model's error-handling behavior.

3.14. P4091R1 - Error Models of Regular C++ and the Sender Sub-Language

Every async facility that separates error and value paths has an abstraction floor - a boundary where compound results are reduced to a single value - and for five years nobody named it. P4091R1 gives it a name, then shows that both coroutines and senders have one: `throw` for coroutines, `set_error` for senders. The critical asymmetry is not that one paradigm is broken and the other is not, but that the coroutine floor is opt-in while the sender composition algebra - `retry`, `when_all`, `upon_error` - lives above it, forcing compound results like `(error_code, bytes_transferred)` to either bypass the algebra or lose data crossing the floor. The paper catalogs six positions for handling compound I/O results in sender pipelines, maps each against data preservation, composition algebra participation, and genericity, and finds that no position achieves all three simultaneously.

3.15. P4092R1 - Consuming Senders from Coroutine-Native Code

Where `execution::task` type-erases, allocates an inner coroutine frame, and wraps every `error_code` in `exception_ptr`, P4092R1's bridge does none of the three - and the complete implementation is one class template printed in the appendix. `await_sender` consumes any `std::execution` sender from inside a coroutine with inline operation state, correct stop-token propagation, and automatic dispatch-back to the originating executor, requiring zero bytes beyond the coroutine frame. Error-code completion signatures are inspected at compile time: senders that advertise `set_error(std::error_code)` produce `io_result<T>` values rather than exceptions, enforcing the abstraction floor from P4093R0. The bridge compiles today on MSVC 19.43 against Capy and `beman::execution`.

3.16. P4093R1 - Producing Senders from Coroutine-Native Code

There is a floor below which compound I/O results should not cross into the sender channel model, and P4093R1 enforces it at compile time with a `static_assert` that fires before anyone loses data. The paper ships `as_sender`, a bridge adapter that structurally inspects the awaitable's `await_resume` return type: bare `error_code` routes cleanly through `set_value`/`set_error`, `void` and single values pass through, but any tuple-like whose first element is `error_code` with additional elements - `io_result<size_t>`, `pair<error_code, size_t>`, or any user-defined shape - is rejected outright. The coroutine body becomes the translation layer: it inspects the compound result, performs application logic, and returns the reduced `error_code` that the bridge can route through the three sender channels with zero exceptions and zero allocation beyond the coroutine frame. The complete implementation is provided in an appendix built on Capy and `beman::execution`, alongside a companion `split_ec` adapter that enforces the same floor inside an existing sender pipeline - giving programmers three distinct enforcement points and one architectural invariant.

3.17. P4094R1 - The Unification of Executors and P0443

Three deployed executor models were replaced by one that was never deployed. P4094R1 conducts a systematic audit of the published WG21 record behind the 2014 decision to unify networking, GPU dispatch, and thread pool executors into P0443 - fourteen revisions, over 100 papers, a property system built and discarded - and finds that the entire evidentiary basis for unification consists of a single hypothetical code snippet authored by the proposal authors themselves. The paper traces the terminology shift from continuation-scheduling primitives (`dispatch`/`post`/`defer`) to the work-submission primitive `execute(F&&)` across six papers spanning 2015 to 2021, documenting that no published paper in the causal chain analyzes the framing change or its consequences for networking. Twenty-one years after the first networking proposal, networking is not in the C++ standard, and the "one grand unified model" premise was polled in 2021 and failed to achieve consensus.

3.18. P4095R1 - The Basis Operation and P1525

Three of the four deficiencies that retired the executor concept vanish when the callable is read as a continuation instead of a unit of work. P4095R1 re-examines P1525R0 - the paper whose diagnosis drove the Cologne pivot to sender/receiver - and applies the work/continuation framing distinction from P4094R0 to each of its four claims about `execute(F&&)`. Under the continuation framing, error propagation is moot because no I/O result exists yet, cancellation is unambiguous because an un-resumed continuation has exactly one state, and zero-allocation scheduling is already solved by `coroutine_handle<>`. The paper documents that no analysis at Cologne evaluated P1525R0's diagnosis under the continuation framing, that the three papers driving the pivot do not mention `async_result` or `dispatch`/`post`/`defer`, and that the technical argument which justified the sender/receiver model was applied to a signature whose original meaning had already been erased from the API surface.

3.19. P4096R1 - Coroutine Executors and P2464R0

The three deficiencies that sidelined the Networking TS in 2021 - no error channel, no lifecycle, no generic composition - hold under one framing of `execute(F&&)` and vanish under the other, and the committee's process had no mechanism to require that both framings be examined before it acted. This retrospective applies both the work framing and the continuation framing from P4094R0 to P2464R0's criteria, then evaluates the coroutine executor concept from P4003R3 - which constrains its argument to `coroutine_handle<>`, enforcing the continuation framing through the type system - against each one. The paper documents that the continuation framing, present in the executor lineage as early as P0113R0 (2015), was erased from the API surface by the terminology shift that produced P0443R14 and was never examined in the analysis that redirected the committee's async effort. Five years after that redirection, no sender-based networking has shipped, and the process gap the paper identifies - no mechanism to verify analytical completeness, no mechanism to revisit outcomes against evidence - remains open.

3.20. P4097R1 - The Networking Claim and P2453R0

In October 2021, LEWG reached consensus that sender/receiver is "a good basis" for networking - and this paper finds zero published networking evidence behind that word at the time of the vote. The audit is methodical: it searches the entire published record surrounding the poll in P2453R0, surfaces voter comments from participants who said they could not evaluate the networking claim yet voted Weakly Favor anyway, and documents that the only networking artifact was a hypothetical code example using a placeholder namespace. Five years later, the evidence column remains empty - no sender-based networking deployment has been published, the partial-success error channel problem identified before the poll persists in the 2023 literature, and it has been twenty-one years since N1925 with networking still absent from the standard.

3.21. P4098R1 - Async Claims and Evidence

The committee voted to make networking depend on senders, and no published paper had ever demonstrated sender-based networking. P4098R1 assembles a structured table of every consequential claim from the executor-networking debate - P0443 through P2464 through P2300 - quotes each one verbatim, and searches the entire WG21 record through March 2026 for supporting evidence: deployment data, prototypes, surveys, measurements. The evidence bar is deliberately low - a code snippet from a real codebase qualifies - yet many cells are empty. Where evidence exists, it is concentrated in GPU dispatch and infrastructure; the poll that added "networking" to the sender mandate achieved consensus without a single published networking prototype.

3.22. P4099R1 - The Twenty-One Year Networking Arc

Four decisions, each locally reasonable, each made by experienced practitioners under real constraints, produced a decade without networking in the C++ standard - and this paper names all four and traces the causal chain that connects them. P4099R1 assembles five companion retrospectives into a single timeline from 2014 to 2026, documenting the published evidence available at each decision point and the evidence that was absent. The recurring finding: sender/receiver deployments cited in committee polls served GPU dispatch, thread pools, and infrastructure, but no production sender-based networking deployment appears in the published record as of 2026. The paper credits std::execution as a genuine achievement while surfacing a distinction - work framing versus continuation framing - that no paper in the causal chain ever examined.

3.23. P4100R1 - Coroutine-Native I/O for C++29 (The Network Endeavor)

Two shipping C++20 libraries, three independent Boost adopters, and a derivatives exchange field report back a fourteen-paper series targeting coroutine-native I/O for C++29. P4100R1 is the roadmap: seven Stage One papers deliver pure C++20 abstractions - execution protocol, task type, buffer ranges, stream concepts, combinators - with no platform dependency; seven Stage Two papers add timers, signals, files, TCP, DNS, UDP, and TLS. The central artifact is `any_stream`, a type-erased byte stream whose ABI is a vtable and whose per-operation allocation cost is zero because operation state lives in the coroutine frame that already exists. Every paper in the series ships with published implementation, cross-platform CI, and a migration path from Asio that requires recompiling one `.cpp` file.

3.24. P4123R0 - The Cost of Senders for Coroutine I/O

P4123R0 grants `std::execution::task` every proposed fix, compares against the best possible conforming implementation, and finds that I/O users still pay for what they do not need. The paper walks through [exec.task] and identifies operations no conforming implementation can eliminate: `state<Rcvr>` construction and scheduler extraction per child task when I/O returns awaitables, and per every read, write, timer, and DNS lookup when I/O returns senders - the direction LEWG and SG4 have polled for. A typical HTTP request-response session under the sender model pays 40 additional spec-mandated operations that the coroutine-native model does not pay, loses symmetric transfer on the I/O completion path because `set_value` is void-returning, and requires a heap allocation per I/O operation for type-erased streams through `any_sender::connect`. The paper proposes nothing and asks for nothing - it provides the cost analysis that P3406R0 argued every proposal should ship with, and leaves the committee to decide whether the gap justifies a separate task type for I/O.

3.25. P4124R0 - Combinators and Compound Results from I/O

`when_all` is the one sender algorithm genuinely irreplaceable inside a coroutine body, and it is blind to I/O errors. This paper traces the completion handler through the P2300R10 specification, examines three strategies for routing compound I/O results through the three-channel model, and shows that all three fail: value-channel routing leaves the combinator blind, error-channel routing corrupts downstream composition, and a predicate parameter repeats domain logic at every call site. The proposed resolution is domain-aware combinators - a single `when_all` that dispatches at compile time, selecting an IoAwaitable overload that inspects results directly with zero adapter overhead or delegating to `std::execution::when_all` when children are senders. A twelve-row behavior table and side-by-side comparison demonstrate that one generic implementation plus N per-call-site adapters is more total code than two implementations behind one name plus zero adapters.

3.26. P4125R1 - Coroutine-Native I/O at a Derivatives Exchange

Engineers who built the technology behind NYSE, NASDAQ, and HKEx are replacing Boost.Asio with a coroutine-only I/O library in a derivatives exchange platform - and after two weeks, their platform architect called it a viable production replacement. P4125R1 reports qualitative findings from three structured interviews conducted during the early stages of this migration, covering callback-to-coroutine conversion, incremental adoption strategy, and build integration. Migration was consistently easier than anticipated: recursive callback patterns collapsed into simple coroutine loops, a "springboard function" pattern enabled incremental adoption without disrupting the existing Asio codebase, and build integration took under an hour. No benchmarks exist yet - the quantitative performance question remains open - but the qualitative verdict from production-grade exchange engineers is that coroutine-native I/O is feasible for mission-critical financial infrastructure.

3.27. P4126R1 - A Universal Continuation Model

Every sender pipeline that invokes an awaitable today allocates a coroutine frame it does not need - one heap allocation per I/O operation, at volumes measured in millions per second. P4126R1 proposes the callback handle: twenty-four bytes of stack storage - two function pointers and a data pointer - that present the same layout as a coroutine frame, giving senders zero-allocation access to every IoAwaitable ever written without modifying a single awaitable. The paper traces the frame-erased versus frame-visible coroutine debate from Kohlhoff's 2015 Resumable Expressions through Core Coroutines to C++20, argues that multiple coroutine models serving non-overlapping domains is consistent with committee practice, and presents working bridge code that compiles on all three major compilers today. The immediate payoff is one I/O implementation consumed by both coroutines and senders at zero allocation cost - timers, channels, semaphores, and every awaitable in the ecosystem become sender-composable without a bridge coroutine.

3.28. P4127R0 - The Coroutine Frame Allocator Timing Problem

Exactly two mechanisms can deliver a frame allocator to a C++20 coroutine's `operator new`, and no future customization point within the current language model can add a third. This paper enumerates every coroutine customization point in the language - `await_transform`, `operator co_await`, `initial_suspend`, deferred invocation, promise-based injection, sender-environment bridge wrappers - and proves that each one either fires after the frame already exists or reduces to one of exactly two paths: the allocator in the parameter list (`allocator_arg_t`), or the allocator in ambient state (thread-local storage). The tradeoff is signature pollution versus hidden state, and the paper documents both costs with side-by-side code showing three-deep coroutine chains under each regime - where Path A infects every signature and every call site, breaks separate compilation, and prevents virtual dispatch, while Path B uses a single zero-initialized pointer invisible to coroutine authors. A hybrid `operator new` overload set lets both paths coexist in a single promise type, making the choice per-call-site rather than per-ecosystem.

3.29. P4166R0 - Benefits of Frame-Visible Coroutines for Senders

A coroutine frame model documented in P1492R0 during C++20 standardization - considered and set aside in 2019 - would eliminate the heap allocation that `std::execution::task` is forced to make today. P4166R0 identifies two of the three properties that senders currently hold over coroutines (optimizer visibility and zero-allocation composition) as consequences of a single language choice: the coroutine frame is opaque behind `coroutine_handle<>`. The paper also surfaces a structural constraint in the three-channel completion model: compound I/O results such as an `error_code` paired with a byte count cannot round-trip through mutually exclusive channels, because exactly one channel fires per completion and the others lose their data. If C++ added frame-visible coroutines alongside the existing opaque-frame model, `std::execution::task` embeds the coroutine frame directly (its `sizeof` becomes `constexpr`), coroutine-based sender algorithms gain full optimizer visibility, and compile-time work graphs - the type-level encoding of dependency topology - remain the unique and irreducible sender contribution.

3.30. P4172R1 - IoAwaitable for Coroutine-Native Byte-Oriented I/O

The IoAwaitable executor concept has seven requirements; this paper demonstrates that removing any single one produces a concrete failure in deployed I/O code. P4172R1 is the design audit trail for P4003R3's normative vocabulary, recording what alternatives were rejected and which domain constraint forced each choice across executor shape, frame allocation, cancellation, and environment propagation. The audience model is the primary evaluation filter: three populations with sharply different skill expectations, and a design rule that if the largest population - application developers - must understand anything beyond "call the function, check the error," the facility is wrong. Frame allocator benchmarks show the recycling allocator at 3.1x the throughput of std::allocator on MSVC and 1.28x faster than mimalloc, yet application coroutines carry zero infrastructure parameters - the out-of-band propagation mechanism and type-erased executor_ref keep the entire allocation and dispatch machinery invisible to every handler signature.

3.31. P4178R0 - Trade-offs in Asynchronous Abstraction Design

Every finding in P4178R0 cites exactly two sources, and both of them are P2300R10. The paper pairs eighteen passages from the C++26 senders specification that appear to be in tension, organized into four patterns: claims walked back within the same section, asymmetric evidentiary standards for coroutines versus senders, prior art dismissed then adopted, and stated goals contradicted by the design. The contrast is sharpest in code: Section 4.16 calls a seven-line coroutine retry "trivial," while Section 1.5.2 presents the sender implementation of the same algorithm at roughly 125 lines. Every finding is accompanied by a charitable reading that explains how the tension might resolve, making the analysis harder to dismiss and easier to act on.

3.32. P4182R1 - A Citable Inventory of Platforms, Operating Systems, and Compiler Toolchains

Every WG21 mailing rehearses the same platform deployment facts, and this paper exists so no author ever has to write them again. P4182R1 consolidates seven platform categories - from full-hosted desktop through bare-metal Cortex-M and GPU device code - and six compiler families into structured tables covering coroutines, TLS, PMR, heap availability, exception defaults, allocation style, and scheduling class. It extends the capability survey from P4127R0 with additional SG14-relevant columns and adds matching compiler rows for GCC, Clang, MSVC, Arm GNU bare-metal, NVIDIA nvcc, and the EDG front end. The document is a living reference: future authors cite one paper number instead of writing three paragraphs of platform context that the next author will rewrite from scratch.

3.33. P4207R0 - Prosecute Your Paper To Improve It

A dollar in API tokens and fifteen minutes of wall-clock time produced five formal objections against the fourteen-revision, three-year Contracts proposal - objections that tracked the substantive national body comments filed after adoption. P4207R0 introduces the Advocatus Diaboli, a structured AI red-teaming methodology for WG21 papers in which a built-in counter-examiner kills bad findings through six named challenges before they reach the record, the tool deposes the author before any charge is filed, and sections that survive earn formal certification as battle-hardened. The methodology ships in three cultural translations - a medieval canonical tribunal, a German engineering inspection, and a Chinese imperial court examination - each implementing identical logic under CC0, runnable against any frontier language model. The paper includes falsifiable predictions on one-, two-, and three-year horizons, forming the retrospective accountability mechanism the committee does not yet require for adopted features.

3.34. P4208R0 - C++ Contracts on Trial - Does P2900 Survive Cross-Examination?

Of the roughly fifteen adversarial charges filed against the 119-page C++26 Contracts proposal, fourteen collapsed under cross-examination - twelve because P2900 had already confessed the limitation before any critic could raise it. This paper applies the Advocatus Diaboli framework from P4207 to P2900R14, stress-testing every claim across twenty-one articuli drawn from five years of SG21 deliberation, seven national-body removal requests, and three competing counter-proposals. Eleven sections earn formal approbatio; only one objection survives: the gap between the paper's safety-motivated design narrative and the non-normative weight of its recommended-practice default for enforce semantics. The verdict is cum obiectionibus - the cause proceeds, and the surviving objection concerns framing rather than substance.

---

4. Conclusion

This reading guide covers 34 papers from the May 2026 mailing. The author hopes it helps the reader find the papers most relevant to their work and interests.

References

[1] P2583R4 - "Symmetric Transfer and Sender Composition" (Vinnie Falco, 2026).

[2] P4003R3 - "A Minimal Coroutine Execution Model" (Vinnie Falco, 2026).

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

[4] P4014R2 - "The Sender Sub-Language For Beginners" (Vinnie Falco, 2026).

[5] P4035R1 - "The Need for Escape Hatches" (Vinnie Falco, 2026).

[6] P4036R0 - "Why Span Is Not Enough" (Vinnie Falco, 2026).

[7] P4041R0 - "Is `std::execution` a Universal Async Model?" (Vinnie Falco, 2026).

[8] P4046R0 - "SAGE: Saving All Gathered Expertise" (Vinnie Falco, 2026).

[9] P4047R0 - "CRYSTAL BALL: Checking Predictions Against the Record" (Vinnie Falco, 2026).

[10] P4048R0 - "Networking for C++29: A Call to Action" (Vinnie Falco, 2026).

[11] P4088R1 - "What C++20 Coroutines Already Buy The Standard" (Vinnie Falco, 2026).

[12] P4089R1 - "On the Diversity of Coroutine Task Types" (Vinnie Falco, 2026).

[13] P4090R1 - "Sender I/O: A Constructed Comparison" (Vinnie Falco, 2026).

[14] P4091R1 - "Error Models of Regular C++ and the Sender Sub-Language" (Vinnie Falco, 2026).

[15] P4092R1 - "Consuming Senders from Coroutine-Native Code" (Vinnie Falco, 2026).

[16] P4093R1 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, 2026).

[17] P4094R1 - "The Unification of Executors and P0443" (Vinnie Falco, 2026).

[18] P4095R1 - "The Basis Operation and P1525" (Vinnie Falco, 2026).

[19] P4096R1 - "Coroutine Executors and P2464R0" (Vinnie Falco, 2026).

[20] P4097R1 - "The Networking Claim and P2453R0" (Vinnie Falco, 2026).

[21] P4098R1 - "Async Claims and Evidence" (Vinnie Falco, 2026).

[22] P4099R1 - "The Twenty-One Year Networking Arc" (Vinnie Falco, 2026).

[23] P4100R1 - "Coroutine-Native I/O for C++29 (The Network Endeavor)" (Vinnie Falco, 2026).

[24] P4123R0 - "The Cost of Senders for Coroutine I/O" (Vinnie Falco, 2026).

[25] P4124R0 - "Combinators and Compound Results from I/O" (Vinnie Falco, 2026).

[26] P4125R1 - "Coroutine-Native I/O at a Derivatives Exchange" (Vinnie Falco, 2026).

[27] P4126R1 - "A Universal Continuation Model" (Vinnie Falco, 2026).

[28] P4127R0 - "The Coroutine Frame Allocator Timing Problem" (Vinnie Falco, 2026).

[29] P4166R0 - "Benefits of Frame-Visible Coroutines for Senders" (Vinnie Falco, 2026).

[30] P4172R1 - "IoAwaitable for Coroutine-Native Byte-Oriented I/O" (Vinnie Falco, 2026).

[31] P4178R0 - "Trade-offs in Asynchronous Abstraction Design" (Vinnie Falco, 2026).

[32] P4182R1 - "A Citable Inventory of Platforms, Operating Systems, and Compiler Toolchains" (Vinnie Falco, 2026).

[33] P4207R0 - "Prosecute Your Paper To Improve It" (Vinnie Falco, 2026).

[34] P4208R0 - "C++ Contracts on Trial - Does P2900 Survive Cross-Examination?" (Vinnie Falco, 2026).