The Unification of Executors and P0443

Document Number:	P4094R0
Date:	2026-04-17
Intent:	Inform
Audience:	WG21

Reply-to:	Vinnie Falco vinnie.falco@gmail.com C++ Alliance Proposal Team

Table of Contents

Abstract

The unification of three working executor models had unanticipated downstream consequences.

In 2014, three deployed executor models - networking, GPU dispatch, and thread pools - were unified into P0443R0[1], which went through fourteen revisions, was never deployed as unified, and was replaced by P2300R10[2]. This paper examines the published record for the evidence that supported the unification decision and documents a terminology shift that erased the continuation framing from the API surface of P0443R14[3]. Section 6 defines two framings of `execute(F&&)` - the work framing and the continuation framing - that subsequent papers in this series apply.

Revision History

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[4] and Corosio[5] 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 examines the published record. That effort requires re-examining consequential papers, including papers written by people the author respects.

P0443[1] was consequential. It was the vehicle for the committee's decision to unify three independent executor models into a single abstraction. That decision shaped the trajectory of executors, networking, and async programming in C++ for a decade. Decisions of that magnitude deserve periodic review. This paper provides one. Where the record shows that certain questions were not asked, the paper names the questions. The intent is to ensure the committee's record is complete, not to assign blame.

The author is a co-author of P2469R0[6], "Response to P2464: The Networking TS is baked, P2300 Sender/Receiver is not," which is cited in this paper. The reader should be aware that the author had a prior published position on the relationship between the Networking TS and P2300[2].

The author's research method is systematic search of the published record - WG21 papers, published poll outcomes, public blog posts, conference talks, and public mailing list archives. Section 4 of this paper documents the absence of certain analyses from the published record. The author acknowledges that absence of evidence is not evidence of absence. Committee discussions occur in rooms, hallways, dinners, and private channels that leave no public trace. A cost/benefit analysis may have been conducted and never written down. A survey may have been taken and never published. The author cannot prove that these things did not happen. If a reader is aware of a document, analysis, or discussion that this paper's research did not reach, the author welcomes the correction and will update the record in a future revision.

This paper asks for nothing.

---

2. Background

This section describes the three executor models that existed before unification, what "unify" meant in practice, and what happened over the following decade.

2.1 Three Models

By 2014, three independent executor proposals existed, each deployed in its domain.

Kohlhoff (networking). N4046[7] (2014). Executors as lightweight policy objects associated with an execution context. `dispatch`, `post`, and `defer` schedule continuations on that context. Designed for event-driven I/O - the Networking TS and Boost.Asio. The execution context is an I/O reactor: threads block in the kernel on `epoll_wait`, `GetQueuedCompletionStatus`, or `kevent`, waiting for I/O completions. Deployed in production for over a decade.

Hoberock/Garland (GPU and parallel algorithms). N4406[8] and P0058R1[9]. Executors as traits classes for bulk execution. Designed for parallel algorithms and GPU dispatch, where a single call creates thousands of execution agents with a shape parameter describing the index space. Deployed at NVIDIA.

Mysen (thread pools). N4414[10]. Executors as handles to thread pools for submitting units of work. The execution context is a work queue: threads pull tasks and execute them in user space. Designed for Google's internal infrastructure. Deployed at

Google.

Each model worked. Each was deployed. Each served its domain. The requirements on executors differed across domains - networking needed continuation scheduling, GPU dispatch needed bulk execution with shape parameters, thread pools needed simple work submission.

2.2 The Direction to Unify

SG1 directed the authors to find a single abstraction. N4199[11], the minutes of the September 2014 SG1 meeting in Redmond, records a straw poll: "Start with Chris Mysen's proposal?" (SF:9 / WF:5 / N:4 / WA:0 / SA:2). The minutes note that the proposals had "mostly converged, but some differences remain."

Kohlhoff and Allsop described the direction retrospectively in P1791R0[12] (2019):

> *"the view of SG1 was that a single executor abstraction was preferred, and so direction was given to the authors of the various proposals to make an effort to find a consensus proposal that would allow the standardisation of that single, or more precisely, unified abstraction."*

P2469R0[6] (2021) described the same event:

> *"SG1 found itself presented with multiple proposals called 'executor', and eventually instructed the various authors to find a unified model that addressed all concerns."*

Step	Year	What happened	Evidence
Three models	2014	Kohlhoff (networking), Hoberock/Garland (GPU), Mysen (thread pools)	Each deployed, each working
SG1 Redmond	2014	"Start with Mysen's proposal" - directed to unify (N4199[11])	Straw poll, no prototype
P0443R0[1]	2016	"Unifies three separate executor design tracks"	No prototype, no experiment
P0761R2[13]	2018	Design document published	No analysis of domain loss
P0443R14[3]	2020	Final unified proposal	Never deployed as unified

2.3 What Unification Meant

P0443R0[1] (2016) was the result. It unified the three models into a single executor concept with multiple categories (`OneWayExecutor`, `TwoWayExecutor`, `BulkOneWayExecutor`, `BulkTwoWayExecutor`) and customization points. P0285R0[14] (Kohlhoff, 2016) proposed the mechanism: customization points to associate distinct executor types with each execution context. The idea was one execution context, multiple executor views, each satisfying different requirements.

Over the next four years, P0443[1] went through fourteen revisions. A property system (`require`/`prefer`) was added to express domain-specific requirements without burdening all executor types. Kohlhoff and Allsop wrote in P1791R0[12]:

> *"more than 100 papers and revisions have been produced that either directly or indirectly have significantly impacted the consensus position represented by P0443."*

2.4 What Happened Next

P2300R10[2] ("std::execution") emerged in 2021 as a successor to P0443R14[1]. P2403R0[15], the P2300R0 presentation, listed what it removed from P0443R0: polymorphic executor wrappers, the thread pool type, the generic property mechanism, and executor as a distinct concept. It replaced the property system with CPO-based queries and centered the design on senders and receivers.

P2400R2[16], the Library Evolution report for Summer 2021, stated: "we had consensus that we want to proceed with P2300 instead of P0443." P2300[2] was adopted for C++26 at the St. Louis meeting in 2024[17].

2.5 The Arc

Three deployed models became one unified proposal (P0443[1]). The unified proposal was never deployed as unified. It was replaced by a second proposal (P2300R10[2]). P2300R10[2] was adopted. The entire arc - from SG1's direction to P2300R10[2]'s adoption - spans roughly a decade. This paper examines the decision that started the arc.

---

3. The Rationale for Unification

The committee articulated several rationales for unification. Each is presented here in full, with the original language, and validated as reasonable. The people who made this decision were experienced practitioners working under real constraints. The decision was not careless.

3.1 Shared Execution Contexts

Kohlhoff wrote in P0285R0[14] (2016):

> *"the underlying execution context, such as a thread pool, may apply to all use cases. As users, we want to be able to have a single thread pool object that can be used for all of the above use cases. In real world applications, the use cases do not always exist in isolation."*

The desire to avoid resource duplication is a real engineering concern. Applications do mix I/O and computation. A networking execution context and a general-purpose thread pool are architecturally different - one blocks on an OS reactor, the other pulls from a work queue - but the desire to share resources across subsystems is genuine. Section 7.3 examines whether "single thread pool" describes one thing or two. For now, the rationale is presented as stated.

3.2 Multiplicative Explosion

The Executors Design Document, P0761R2[13] (2018), argued:

> *"each facility's unique interface necessitates an entirely different implementation. As the library introduces new facilities, each introduction intrudes upon parallel_for's implementation. Moreover, the problem worsens as the library introduces new functions."*

The cross product of N facilities and M algorithms grows without bound. Standard library maintainers who would implement N x M combinations had legitimate reason to want N + M. This is a real engineering concern.

3.3 Synchronization Coherence

P0761R2[13] also argued that different execution semantics - blocking, non-blocking, continuation - need a common vocabulary to prevent data races. A common vocabulary for synchronization properties is genuinely valuable. Code that must account for different synchronization guarantees across different executor types is harder to reason about.

3.4 Consensus Through Compromise

Kohlhoff and Allsop wrote in P1791R0[12] (2019):

> *"P0443 represents a significant body of compromise and consensus seeking."*

More than 100 papers and revisions. Organizations including Google, NVIDIA, Sandia National Labs, Codeplay, Facebook, Nasdaq, Clearpool.io, Microsoft, and RedHat. Domains including ultra low latency finance, embedded systems, GPU, SYCL,

networking, machine learning, and big data. The breadth of participation was extraordinary. The effort to find common ground was genuine and sustained.

3.5 The Committee's Expressed Preference

P2453R0[18], "2021 October Library Evolution Poll Outcomes," documents the following poll:

> *"We believe we need one grand unified model for asynchronous execution in the C++ Standard Library, that covers structured concurrency, event based programming, active patterns, etc."*
>
> *SF:4 / WF:9 / N:5 / WA:5 / SA:1 - No consensus (leaning in favor).*

The same paper documents Poll 2:

> *"The sender/receiver model (P2300) is a good basis for most asynchronous use cases, including networking, parallelism, and GPUs."*
>
> *SF:24 / WF:16 / N:3 / WA:6 / SA:3 - Consensus in favor.*

P2400R2[16] stated: "we had consensus that we want to proceed with P2300 instead of P0443." The committee leaned toward unification even when consensus on the "grand unified model" premise was not achieved.

3.6 The P2300 Achievement

P2300R10[2] addressed the deficiencies that P2464R0[19] identified in P0443R14[1]: no error channel, no lifecycle for submitted work, and no generic composition. The sender/receiver model provides structured concurrency, sender composition, completion signatures as type-level contracts, and a customization point model that enables heterogeneous dispatch.

P2470R0[20] documented deployments at Facebook ("monthly users number in the billions"), NVIDIA ("fully invested in P2300... we plan to ship in production"), and Bloomberg (experimentation). The unified model produced real value for real users in the domains it serves.

3.7 Summary

These are the reasons the committee chose unification. Each is grounded in real engineering concerns. Each was articulated by experienced practitioners. The decision was not careless.

---

4. The Published Record

Section 3 presented the rationale. This section examines the evidence.

An assertion is a claim about what users need or what the design will provide. Evidence is anything that tests the assertion against reality. The bar is deliberately low: a code snippet from a real codebase showing friction counts. A prototype that demonstrates the unified model working across domains counts. A survey of even a handful of users counts. An experiment comparing approaches counts. Even a hypothetical code example constructed by the author counts - this paper will note it and credit it.

The published record was searched systematically using available tools and all WG21 papers available to the author through the March 2026 mailing. The Evidence column below documents everything found, including partial evidence. Where the column says "(none found in the published record)," the search found nothing - not even a code snippet.

Assertion	Source	Evidence
"we want to be able to have a single thread pool object that can be used for all of the above use cases"	P0285R0[14] (2016)	(none found in the published record) No survey of applications that mix networking and parallel algorithm executors on the same pool. No deployment data showing friction from separate pools.
"each facility's unique interface necessitates an entirely different implementation" (the N x M argument)	P0761R2[13] (2018)	One hypothetical code snippet. P0761R2[13] Section 2 constructs a `parallel_for` with an `if/else` chain over OpenMP, GPU, and thread pool. This is the only code-level evidence in the entire published record for any unification rationale. The snippet is authored by the proposal authors, not drawn from a deployed codebase. No measurement from a real standard library implementation.
"the view of SG1 was that a single executor abstraction was preferred"	P1791R0[12] (2019)	(none found in the published record) N4199[11] records a straw poll "Start with Chris Mysen's proposal?" (SF:9/WF:5/N:4/WA:0/SA:2). No rationale for why a single abstraction is preferred over multiple. No analysis of what would be lost.
"Proposals mostly converged, but some differences remain"	N4199[11] (2014)	(none found in the published record) The minutes note specific differences (begin-work/end-work brackets, scheduling executors, three spawn variants) but no analysis of whether these differences are reconcilable or fundamental.
"serves the use cases of those independent proposals with a single consistent programming model"	P0443R0[1] (2016)	(none found in the published record) No prototype demonstrating that the unified model serves all three use cases. No experiment comparing unified and domain-specific approaches.
"P0443 represents a significant body of compromise and consensus seeking"	P1791R0[12] (2019)	The breadth of participation is documented (Google, NVIDIA, Sandia, Codeplay, Facebook, etc.). More than 100 papers. This is evidence of effort. It is not evidence that the unification preserved domain semantics.
Assertion	Source	Evidence
"some reduction in usability, due to the	P2469R0[6] (2021)	The authors of the unified proposal acknowledge the cost. No measurement of the magnitude.

Assertion	Source	Evidence
increased complexity of the unified interface"		

Every assertion in the left column is reasonable. Every source in the middle column is a published paper by an experienced practitioner. The right column documents what the published record provides in the way of empirical support.

---

5. Questions About the Cost of Unification

In 2014, the costs of unification were hypothetical. In 2026, some are observable. The schedule is a fact. The revision count is a fact. The features that were added and removed are documented in published papers. The polls are published. The deployments are published.

This section presents what is now measurable and asks whether each measurement constitutes a cost. The counterfactuals - what would have happened without unification - remain unmeasurable. This paper says so where it applies.

5.1 Did Unification Delay Networking?

Measurable. The timeline is a fact:

- N1925[21] (2005): first networking proposal.
- Networking TS published as ISO TS (2018).
- Herb Sutter, San Diego 2018[22]: executors "design approved for C++20."
- P1256R0[23] (Vollmann, 2018): "SG1 has decided that the Networking TS should not be merged into the C++ working paper before executors go in."
- P2130R0[24] (Prague 2020 minutes): Pablo Halpern asks "How blocked is networking on the executors wording process?"
- P2464R0[19] (Voutilainen, 2021): "Stop spending energy on standardizing the Networking TS for C++23."
- 2026: networking is not in the C++ standard. Twenty-one years from N1925[21].

The coupling between networking and executors is documented in published papers. The timeline is observable. Is this a cost of unification, or would networking have been delayed regardless? This paper cannot prove causation.

5.2 Did Unification Consume Committee Bandwidth?

Measurable. The paper count is a fact:

- P0443R0[1] (2016) through P0443R14[3] (2020): fourteen revisions over four years.
- P1791R0[12]: "more than 100 papers and revisions."
- P2403R0[15] lists what P2300R10[2] removed from P0443R14[1]: polymorphic executor wrappers, the thread pool type, the generic property mechanism, and executor as a distinct concept.

- P1658R0[25] (2019): documents the controversy around `require_concept` and interface-changing properties, recommends eliminating them.
- The property system (`require`/`prefer`) was built across multiple revisions and then discarded entirely when P2300R10[2] replaced it with CPO queries.

Is fourteen revisions and 100+ papers evidence of healthy design evolution, or is it evidence that the problem was ill-posed? Is discarding the property system a normal course correction, or is it evidence that the unified interface could not stabilize?

5.3 Did Domain-Specific Semantics Survive?

Partially measurable. P2403R0[15] documents what P2300R10[2] removed from P0443R14[1]. Section 6 of this paper documents the terminology shift from continuation-scheduling primitives (`dispatch`/`post`/`defer`) to work-submission (`execute`). P2469R0[6] acknowledges "some reduction in usability, due to the increased complexity of the unified interface." P1791R0[12] states that "requirements that were not universal" were removed during unification.

Was the removal a necessary simplification, or did it erase something the domains needed?

5.4 Could the Domains Have Shipped Independently?

Counterfactual - not measurable. The coupling is documented (P1256R0[23]: networking blocked on executors). The Networking TS was ready as an ISO TS in 2018. GPU dispatch had its own model. Thread pools had theirs. No one can prove what would have happened in an alternate timeline.

5.5 Was the Unified Model Ever Deployed as Unified?

Measurable. P0443[1] was never deployed as a unified model. P2300R10[2] replaced it. P2470R0[20] documents P2300[2] deployments at Facebook, NVIDIA, and Bloomberg - primarily for sender/receiver composition, GPU dispatch, and infrastructure. No published paper documents a deployment where a single P2300[2] scheduler serves networking, GPU dispatch, and thread pool use cases simultaneously in one application.

Does the absence of a unified deployment mean the unification was unnecessary, or does it mean the deployment has not yet happened?

5.6 Did the "One Model" Premise Achieve Consensus?

Measurable. P2453R0[18] documents the October 2021 poll: "We believe we need one grand unified model for asynchronous execution" (SF:4 / WF:9 / N:5 / WA:5 / SA:1 - no consensus, leaning in favor). The premise was tested by poll and did not achieve consensus.

The selected comments in P2453R0[18] show the range of views. One strongly-in-favor comment reads: "There should be one model." Another reads: "I'd be willing to have more specialized models (like the Networking TS/Asio model) for specific components, as long as they could interoperate with the sender/receiver model."

Does "no consensus, leaning in favor" constitute sufficient basis for a decade-long design commitment?

5.7 Summary

Question	2014 Assertion	2026 Outcome
Shared thread pool needed?	"single thread pool for all use cases" (P0285R0)	No survey; reactor and work queue are architecturally different (7.3)
N x M explosion?	Hypothetical `parallel_for` snippet (P0761R2)	No measurement from a real codebase
Domain semantics preserved?	"serves the use cases of those independent proposals" (P0443R0)	dispatch/post/defer renamed to execute (Section 6); properties removed
Networking blocked?	(not discussed in 2014)	21 years from N1925; still not in the standard
Unified model deployed?	(not discussed in 2014)	P0443 never deployed as unified; P2300 deployed for sender/receiver
One model consensus?	(not polled until 2021)	SF:4 / WF:9 / N:5 / WA:5 / SA:1 - no consensus (P2453R0)
Iteration cost?	(not estimated)	14 revisions, 100+ papers, property system built and discarded
Usability cost?	(not estimated)	"some reduction in usability" (P2469R0); no measurement of magnitude

---

6. The Framing Change

Independently of the scope expansion documented in Section 2, the terminology shifted. What began as continuation-scheduling primitives was progressively renamed, demoted, and erased until the continuation framing was no longer visible on the API surface.

6.1 The Evolution of Terminology

Paper	Year	Continuation status
P0113R0[26]	2015	First-class: `defer` = "continuation of the caller"
P0688R0[27]	2017	Demoted to `prefer(is_continuation)` hint
P0761R2[13]	2018	Optional property, framed as "may execute more efficiently"
P1525R0[28]	2019	Pure work language. One incidental mention of "continuation"
P1660R0[29]	2019	"Eagerly submits work." Executor = "factory for execution agents"
P0443R14[3]	2020	Section heading: "Executors Execute Work." Continuation property gone
P2464R0[19]	2021	"Work-submitter." No `async_result`, no N3747[30]

Each step was locally reasonable.

6.2 The Two Stages

The API that P0443R0[1] inherited from three independent models had a large surface: `execute`, `post`, `defer`, `sync_execute`, `async_execute`, `async_post`, `async_defer`, `bulk_execute`, and their variants. The simplification effort that reduced this surface addressed real complexity. The transition from continuation-scheduling primitives to a work-submission primitive happened in two stages.

Stage 1: `dispatch`/`post`/`defer` become `.execute()` plus a property hint.

P0688R0[27] (Kohlhoff, Hoberock, Mysen, Brown, 2017) replaced the three separate functions with a single `.execute()` function. The continuation semantics that `defer` had carried as a first-class operation were converted into an optional preference: `require(exec, oneway).prefer(is_continuation).execute(task)`. The paper's stated rationale:

> *"As described by P0443R1, `defer()`'s semantics are equivalent to `post()`'s. The distinction is that `defer()` acts as a hint to the executor to create the execution agent in a particular way. The presence of `defer()` in P0443R1 was controversial."*

The "hint" is about optimization latitude, not about what the callable is. `post` is itself a continuation-scheduling primitive - it schedules a resumption handle on an execution context. `defer` adds the information that the callable is a continuation of the caller, which permits the executor to optimize scheduling. An implementation of `defer` that falls back to `post` is always correct. The entire `dispatch`/`post`/`defer` family operates within the continuation framing. The hint distinguishes among continuation-scheduling strategies; it does not distinguish continuation scheduling from work submission.

P0688R0[27] does not analyze what is lost by converting three continuation-scheduling primitives into a single work-submission primitive with an optional property. It does not discuss the implications for networking. P0443R2[31] confirmed: "Applied the simplification proposed by paper P0688." That is the first stage.

Stage 2: All interface-changing properties eliminated.

Three papers drove the second stage:

- P1525R0[28] (Niebler, Shoop, Baker, Howes, 2019), "One-Way execute is a Poor Basis Operation." Analyzes `execute` purely as work submission. Uses the phrase "work submission" throughout. Contains one incidental mention of "continuation" in the context of a deadline executor. Does not discuss the original continuation framing of `dispatch`/`post`/`defer`. Does not mention `async_result` or N3747[30].
- P1658R0[25] (Hoberock, Lelbach, 2019), "Suggestions for Consensus on Executors." Proposes: "Eliminate OneWayExecutor, BulkOneWayExecutor, and interface-changing properties." Introduces "an Executor concept based on a function named execute which eagerly creates a single execution agent." Does not discuss the continuation framing. Does not analyze what networking loses when the continuation property is removed.
- P1660R0[29] (Hoberock, Garland, Lelbach, Dominiak, Niebler, Shoop, Baker, Howes, Hollman, Brown, 2019), "A Compromise Executor Design Sketch." Defines the Executor concept as a type with an `.execute()` member function that "eagerly submits a function for execution on an execution agent created for it by the executor." Pure work language throughout. Does not discuss the continuation framing. Does not analyze the implications of the framing change for networking.

P0443R11[32] (Cologne, July 2019) implemented these changes:

> *"As directed by SG1 at the 2019-07 Cologne meeting, we have implemented the following changes suggested by P1658 and P1660 which incorporate 'lazy' execution: Eliminated all interface-changing properties."*

The `continuation`/`not_continuation` properties were removed along with every other interface-changing property. A vestigial `relationship_t` property survived in P0443R14[3], but the continuation as a first-class concept was gone. That is the second stage.

What the published record does not contain:

- No paper in the chain discusses the shift from the continuation framing to the work framing.
- No paper analyzes what networking loses when the continuation property is removed.
- No paper acknowledges that the callable's role has shifted from resumption handle to unit of work.
- No straw poll asks whether the framing change is acceptable.
- No paper evaluates whether the deficiencies of `execute(F&&)` - no error channel, no lifecycle, no composition - are properties of the work framing rather than properties of the underlying operation.

No published paper in the causal chain discusses the framing change. No straw poll addresses it. The work framing emerged as a side effect of two simplification efforts. That is the record.

6.3 How Rationale Is Lost Across Paper Boundaries

The names `post`, `dispatch`, and `defer` do not self-evidently convey continuation semantics in plain English. "Post work" and "defer work" sound like work submission. "Dispatch" sounds like "execute." The continuation framing was established not by the names but by the published definitions - P0113R0[26] defines `defer` as scheduling "a continuation of the caller," and the Networking TS documentation describes all three operations as scheduling handlers on an execution context. The names were ambiguous. The definitions were not.

When a proposal redesigns an API surface inherited from a prior proposal, the new paper inherits the signatures and names but not necessarily the conceptual model that motivated them. Design rationale that exists only in the prior paper's prose - not encoded in the type system, not enforced by concepts, not visible in the function signatures - is fragile across paper boundaries.

At Stage 1, the continuation framing survived. P0688R0[27] collapsed `dispatch`/`post`/`defer` into `execute`, but Kohlhoff - the author of the continuation framing - was a co-author of P0688R0[27]. The continuation semantics were preserved as a property hint: `prefer(is_continuation)`. The institutional knowledge was still present in the paper because the domain expert was still an author.

At Stage 2, the continuation framing was not carried forward into the analysis. P1525R0[28] inherited `execute` and analyzed it as work submission. P1660R0[29] inherited `execute` and defined it as eagerly submitting work. Neither cited P0113R0[26]'s continuation definitions. Neither restated the rationale that the callable is a resumption handle rather than a unit of work. The Stage 2 authors understood continuations - P1194R0[33], "The Compromise Executors Proposal" (Howes, Niebler, Shoop, Lelbach, Hollman, 2018), explicitly states "Receivers are Continuations," and several of its authors co-authored P1525R0[28] and P1660R0[29]. The continuation concept was relocated to the sender/receiver framework rather than preserved in the `execute` API. What the published record does not contain is an analysis of this shift - no paper discusses why the continuation framing for `execute` was dropped, what was gained, or what was lost.

The continuation framing for `execute` was carried by institutional knowledge rather than by the API surface or the type system. When the property hint was removed and the continuation concept relocated to receivers, the framing for the basis operation dropped out of the published analysis. This is not a criticism of any individual author - the authors understood the concept and chose to place it elsewhere. It is a structural property of multi-paper, multi-author standardization: when a design decision is made deliberately but the rationale is not published, it becomes invisible to future readers of the record.

6.4 The Two Framings

The continuation framing. `dispatch`/`post`/`defer` schedule a continuation on an execution context. The callable is a resumption handle. The operating system performs the work. The result is delivered to the continuation when it wakes up. The executor never touches the result.

The work framing. `execute(F&&)` submits work. The callable is a unit of work. The executor runs it. If dropped, the work and its result are lost. Error handling, lifecycle, and composition are the executor's responsibility.

The structural difference is in what happens to the caller. `post` ends the caller's chain of execution. The caller returns. There is no live caller on the other end to receive a report. The continuation will be resumed later, on a context, and the result will be delivered *to* the continuation when it wakes up. Under the work framing, `execute` is a fork: the caller submits work and continues. The caller is alive, running, and expects to learn what happened. A live caller needs an error channel. A caller that has returned does not.

The executor in the continuation framing is not trivially simple. Asio executors carry real complexity - strands for serialization, associated allocators for memory control, `dispatch` vs `post` semantics for reentrancy. That complexity governs *how* and *where* the continuation resumes. None of it involves reporting results back to a live caller, because there is no live caller. The caller returned. The complexity is scheduling policy, not result delivery.

The `F&&` signature is the final form of the terminology shift documented above. Kohlhoff's original `dispatch`/`post`/`defer` carried the continuation framing in their published definitions (Section 6.3); the callable was understood to be a resumption handle. The rename to `execute(F&&)` removed that constraint. The signature permits fire-and-forget use because nothing in the type system prevents it.

The work framing imposes requirements on the executor that the continuation framing does not. Under the work framing, the executor is responsible for error channels, lifecycle management, and generic composition. Under the continuation framing, the executor schedules a resumption, the OS performs the work, and the result is delivered to the continuation when it wakes up. A continuation carries no result because the result does not yet exist. An error channel for a resumption handle would report on work that has not happened. The generalization from `dispatch(handler)` to `execute(F&&)` introduced these requirements.

---

7. Questions About the Cost of Multiple Models

Section 5 examined the costs of the path taken. This section examines the costs of the path not taken. If the three models had proceeded independently, what would the committee have paid?

7.1 Would Multiple Models Fragment the Ecosystem?

Three executor concepts instead of one. Libraries must choose which to target. Interop requires bridges. Fragmentation is a real concern.

The committee has shipped or voted to ship multiple design approaches for the same domain before. The bar for the following table is strict: two fundamentally different design approaches to the same problem. Variations within one approach do not count - type erasure of an existing model is not a second model.

Domain	Models	Status
Parallel execution	C++17 execution policies (`par`, `seq`, `par_unseq`) and P2300R10[2] senders with `bulk`	Shipped
Formatted output	`iostream` and `std::format`/`std::print`	Shipped
Async task types	`std::execution::task` (P3552R3[34]) is simultaneously a coroutine and a sender	In-flight

Three precedents. This paper does not inflate the list. The `task` precedent is the sharpest - the committee has already voted to ship a type that fuses two async models into one. Is "one model for async execution" consistent with the committee's practice in parallel execution and formatted output?

7.2 Would Multiple Models Be Harder to Teach?

"Harder to teach" is frequently cited as a cost of multiple models. More surface area in the standard is a real concern.

The published record contains no measurement of teachability for any executor design. Not for the unified model. Not for multiple models. Not for P0443[1]. Not for P2300R10[2]. No user survey. No classroom study. No measurement of time-to-productivity. No comparison of error rates between approaches.

The "harder to teach" argument is asserted in both directions - proponents of unification say one model is easier to teach; proponents of domain-specific models say a smaller model that matches the student's domain is easier to teach. Neither side has measured it.

Is "harder to teach" a cost that can be assessed without evidence? The committee uses teachability as a criterion for design decisions, but the published record contains no methodology for evaluating it. This paper does not argue that teachability is irrelevant. It observes that it has never been measured, and that an unmeasured cost can be asserted for any position.

7.3 Would Applications Need Multiple Thread Pools?

P0285R0[14] argued: "we want to be able to have a single thread pool object that can be used for all of the above use cases."

A networking execution context and a general-purpose thread pool are architecturally different. A networking context blocks its threads on an OS reactor - `epoll_wait`, `GetQueuedCompletionStatus`, `kevent`. The threads are parked in the kernel waiting for I/O completions. A general-purpose thread pool is a work queue where threads pull tasks and execute them in user space. These have different blocking models, different scheduling properties, and different performance characteristics.

An application that runs networking on an I/O reactor and parallel algorithms on a work-stealing pool is not using "two thread pools for the same purpose." It is using two different execution contexts for two different purposes. No survey was done to determine how many applications actually need a single context serving both.

7.4 Are Interop Bridges Expensive?

If models are separate, bridging between them has a cost. P4092R0[35] and P4093R0[36] demonstrate coroutine-to-sender and sender-to-coroutine bridges. The implementations exist.

Is the bridge cost bounded and acceptable, or does it accumulate into a tax that makes separate models impractical? The implementations are published.

7.5 Would Independent Schedules Have Delivered Value Earlier?

If each model ships for its domain when it is ready, networking ships when networking is ready. GPU dispatch ships when GPU dispatch is ready. Neither blocks the other.

Is shipping three things on three schedules better than shipping one thing on one schedule?

---

8. Anticipated Objections

Q: Unification was the right call - look at P2300.

A: This paper does not argue otherwise. Section 3 documents the rationale. Section 5 presents questions about observable costs. Section 7 presents questions about the alternative.

Q: Multiple models would have fragmented the ecosystem.

A: Section 7.1 presents this as an open question. This paper does not argue fragmentation is zero. It asks whether fragmentation and specialization are the same thing, and presents three precedents where the committee chose multiple approaches for the same domain.

Q: This is hindsight bias.

A: Section 4 documents what evidence was available at the time the decision was made. The questions in Sections 5 through 7 could have been asked in 2014. They were not.

Q: The continuation framing is a retroactive reinterpretation.

A: P0113R0[26] (Kohlhoff, 2015) defined `defer` as scheduling "a continuation of the caller." Section 6.1 documents every paper in the chain. The continuation framing is the original framing. The work framing is the replacement.

Q: The committee did poll on one model.

A: Section 3.5 cites the poll. It achieved no consensus (SF:4 / WF:9 / N:5 / WA:5 / SA:1, leaning in favor).

Q: You are arguing for your own library.

A: Section 1 discloses this. The evidence in Sections 4 through 7 stands or falls on the published record, not on who assembled it.

This paper examined the published record for the evidence that supported the decision to unify three independent executor models into a single abstraction. The rationale is documented in Sections 2 and 3. The evidence is documented in Section 4. The observable outcomes are documented in Section 5. The framing change that the unification produced is documented in Section 6.

Good stewardship of the standard means revisiting consequential decisions when new evidence is available.

Acknowledgments

The author thanks Chris Kohlhoff for the executor model that started the journey and for the candid retrospective in P1791R0[12]; Jared Hoberock, Michael Garland, and Chris Mysen for P0443R14[1] and P0761R2[13]; Eric Niebler, Kirk Shoop, Lewis Baker, and their collaborators for P2300R10[2]; Ville Voutilainen for P2464R0[19]; Bryce Adelstein Lelbach for the published poll outcomes in P2453R0[18] and P2400R2[16]; Detlef Vollmann for P1256R0[23]; Jamie Allsop, Richard Hodges, and Klemens Morgenstern for co-authoring P2469R0[6]; and Steve Gerbino for feedback on this paper.

References

[1] P0443R0 - "A Unified Executors Proposal for C++" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, 2016).

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

[3] P0443R14 - "A Unified Executors Proposal for C++" (Jared Hoberock et al., 2020).

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

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

[6] P2469R0 - "Response to P2464: The Networking TS is baked, P2300 Sender/Receiver is not" (Christopher Kohlhoff, Jamie Allsop, Vinnie Falco, Richard Hodges, Klemens Morgenstern, 2021).

[7] N4046 - "Executors and Asynchronous Operations" (Christopher Kohlhoff, 2014).

[8] N4406 - "Integrating Executors with Parallel Algorithm Execution" (Jared Hoberock, Michael Garland, Olivier Giroux, 2015).

[9] P0058 - "An Interface for Abstracting Execution" (Jared Hoberock, Michael Garland, Olivier Giroux, 2015).

[10] N4414 - "Executors and Schedulers Revision 5" (Chris Mysen, 2015).

[11] N4199 - "Minutes of Sept. 4-5, 2014 SG1 meeting in Redmond, WA" (Hans-J. Boehm, 2014).

[12] P1791R0 - "Evolution of the P0443 Unified Executors Proposal to accommodate new requirements" (Christopher Kohlhoff, Jamie Allsop, 2019).

[13] P0761R2 - "Executors Design Document" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Michael Wong, 2018).

[14] P0285R0 - "Using customization points to unify executors" (Christopher Kohlhoff, 2016).

[15] P2403R0 - "Presentation on P2300 - std::execution" (2021).

[16] P2400R2 - "Library Evolution Report: 2021-06-01 to 2021-09-20" (Bryce Adelstein Lelbach, 2021).

[17] Herb Sutter, "Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)," 2024

[18] P2453R0 - "2021 October Library Evolution Poll Outcomes" (Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, 2022).

[19] P2464R0 - "Ruminations on networking and executors" (Ville Voutilainen, 2021).

[20] P2470R0 - "Slides for presentation of P2300R2: std::execution (sender/receiver)" (Eric Niebler, 2021).

[21] N1925 - "Networking proposal for TR2 (rev. 1)" (Chris Kohlhoff, 2005).

[22] Herb Sutter, "Trip report: Fall ISO C++ standards meeting (San Diego)," 2018

[23] P1256R0 - "Executors Should Go To A TS" (Detlef Vollmann, 2018).

[24] P2130R0 - "WG21 2020-02 Prague Record of Discussion" (Nina Ranns, 2020).

[25] P1658R0 - "Suggestions for Consensus on Executors" (Jared Hoberock, Bryce Adelstein Lelbach, 2019).

[26] P0113R0 - "Executors and Asynchronous Operations, Revision 2" (Christopher Kohlhoff, 2015).

[27] P0688R0 - "A Proposal to Simplify the Unified Executors Design" (Chris Kohlhoff, Jared Hoberock, Chris Mysen, Gordon Brown, 2017).

[28] P1525R0 - "One-Way execute is a Poor Basis Operation" (Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes, 2019).

[29] P1660R0 - "A Compromise Executor Design Sketch" (Jared Hoberock, Michael Garland, Bryce Adelstein Lelbach, Michał Dominiak, Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes, David S. Hollman, Gordon Brown, 2019).

[30] N3747 - "A Universal Model for Asynchronous Operations" (Christopher Kohlhoff, 2013).

[31] P0443R2 - "A Unified Executors Proposal for C++" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, 2017).

[32] P0443R11 - "A Unified Executors Proposal for C++" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, Lee Howes, Kirk Shoop, Eric Niebler, 2019).

[33] P1194R0 - "The Compromise Executors Proposal: A lazy simplification of P0443" (Lee Howes, Eric Niebler, Kirk Shoop, Bryce Lelbach, D. S. Hollman, 2018).

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

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

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