The Basis Operation and P1525

Document Number:	P4095R0
Date:	2026-04-17
Intent:	Inform
Audience:	SG1, LEWG

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

Table of Contents

Abstract

Revision History

- R0: April 2026 (post-Croydon mailing)

1. Disclosure
2. What P1525R0 Argued
   - 2.1 No Reliable Error Propagation
   - 2.2 No Reliable Cancellation Signal
   - 2.3 No Zero-Allocation Scheduling
   - 2.4 The Asymmetry
   - 2.5 The Achievement
3. What P1525R0 Did Not Analyze
   - 3.1 The Continuation Framing
   - 3.2 The Universal Async Model
   - 3.3 The Networking Use Case
   - 3.4 The Framing Change
   - 3.5 Summary
4. The Framing Dependency
   - 4.1 Error Propagation
   - 4.2 Cancellation
   - 4.3 Zero-Allocation Scheduling
   - 4.4 The Asymmetry
   - 4.5 Summary
5. The Cologne Pivot
   - 5.1 What Happened
   - 5.2 What the Published Record Does Not Contain
6. The Coroutine Executor Under P1525R0's Criteria
7. Anticipated Objections

Acknowledgments

References

Abstract

Of the four deficiencies that P1525R0[1] identified in `execute(F&&)`, three do not arise under the original framing of the callable as a continuation, and the fourth addresses a different question.

This paper documents what P1525R0[1], "One-Way execute is a Poor Basis Operation," analyzed, what it did not analyze, and applies the two-framing distinction from P4094R0[2] to its diagnosis.

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

P1525R0[1] was consequential. It provided the technical argument that the executor concept's basis operation was insufficient for building higher-level async control structures. The argument was adopted at Cologne. The sender/receiver model that replaced the executor concept traces its committee lineage to this paper. Decisions of that magnitude deserve periodic review. This paper provides one. The intent is to ensure the committee's record is complete, not to assign blame.

The coroutine executor concept did not exist in 2019. C++20 coroutines were ratified in 2020. P4003R0[5], "Coroutines for I/O," was published in 2026. The analysis this paper provides was not available when P1525R0[1] was written.

This paper asks for nothing.

---

2. What P1525R0 Argued

P1525R0[1] examined the `OneWayExecutor` concept of P0443R10[6]:

```
void execute(F&& f);
```

The paper identified four deficiencies and proposed an alternative basis operation.

2.1 No Reliable Error Propagation

P1525R0[1] Section 2:

> *"Any errors that happen, whether during task submission, after submission and prior to execution, or during task execution, are handled in an implementation-defined manner, which can vary from executor to executor. The implication is that no generic code can respond to asynchronous errors in a portable way."*

The paper documented four error-handling strategies - ignore and propagate a default, cancel dependent execution, log before propagating, reschedule on a fallback - and showed that none could be built generically on top of `execute(F&&)`. Appendix B demonstrated a `fallback_executor` that reschedules on error using the Sender/Receiver error channel.

2.2 No Reliable Cancellation Signal

P1525R0[1] Section 3:

> *"In functional programming circles, optional is represented as the Maybe monad, which has two constructors: Just and None... The same is true of composing asynchronous computations. If a preceding computation is cancelled, dependent computations should likewise be canceled, bypassing the normal control flow."*

The paper argued that destruction-without-execution is insufficient for communicating cancellation (Section 3.3), because a destructor call could mean four different things - post-value cleanup, post-error cleanup, moved-from cleanup, or cancellation - and distinguishing them requires tracking extra state.

2.3 No Zero-Allocation Scheduling

P1525R0[1] Section 4.1:

> *"It is not possible to build this kind of non-allocating executor-schedule operation if one-way execute() is the basis operation."*

Appendix C demonstrated a `thread_dispatcher` where `co_await ex.schedule()` embeds the queue entry in the coroutine frame, eliminating the heap allocation that `execute(F&&)` requires for type erasure.

2.4 The Asymmetry

P1525R0[1] Section 2.3.1:

> *"we cannot implement schedule generally in terms of one-way execute."*

Appendix A demonstrated the reverse: `execute` implemented in terms of `schedule` and `submit`. The paper concluded that `schedule` - which returns a Sender - is the more foundational abstraction.

2.5 The Achievement

These are real deficiencies of the `execute(F&&)` signature. The sender/receiver model that P1525R0[1] proposed - Scheduler, Sender, Receiver, with `schedule()` returning a Sender and three completion channels (`set_value`, `set_error`, `set_done`) - addressed all four. P2300R10[7] provides compile-time sender composition, structured concurrency guarantees, and a customization point model that enables heterogeneous dispatch. The people who wrote P1525R0[1] were experienced practitioners who identified genuine structural problems and proposed a design that solved them.

---

3. What P1525R0 Did Not Analyze

P4094R0[2] Section 6 documented a terminology shift: what began as continuation-scheduling primitives (`dispatch`/`post`/`defer`) was progressively renamed to `execute(F&&)`. The continuation framing - where the callable is a resumption handle and the operating system performs the work - was no longer visible on the API surface by the time P1525R0[1] was written.

3.1 The Continuation Framing

P1525R0[1] defines its subject in Section 1.1.1:

> *"For the purpose of this document, by 'one-way execute,' we mean a void-returning function that accepts a nullary Invocable and eagerly submits it for execution on an execution agent that the executor creates for it."*

"Submits it for execution." "Execution agent that the executor creates." The language is the work framing throughout. The word "continuation" appears in several places - Section 2.3's deadline executor, Section 3.3's cancellation discussion - but always as a generic term for the callable. P1525R0[1] never engages with the continuation *framing* from P0113R0[8]: the specific meaning where the callable is a resumption handle, the caller has returned, and the OS performs the work. `dispatch`, `post`, and `defer` - the continuation-scheduling primitives that `execute` replaced - do not appear.

3.2 The Universal Async Model

N3747[9], "A Universal Model for Asynchronous Operations" (Kohlhoff, 2013), documented `async_result` - the mechanism that adapts any completion token to any async operation. M executors and N completion models require one implementation of each operation plus N specializations of `async_result`. P1525R0[1] does not mention `async_result`, completion tokens, or N3747[9].

3.3 The Networking Use Case

P1525R0[1]'s examples are thread pools, deadline executors, GPU contexts, and a `when_any` algorithm. No example involves an I/O reactor. No example involves `async_read`, sockets, or the operating system performing work on behalf of the caller. No analysis addresses what a networking execution context needs from a basis operation.

3.4 The Framing Change

P0688R0[10] (2017) replaced `dispatch`/`post`/`defer` with `.execute()` plus a property hint. P0443R11[11] (2019) eliminated the property. P1525R0[1] analyzed the result. The paper does not acknowledge that `execute(F&&)` replaced three continuation-scheduling primitives, or that the callable's role shifted from resumption handle to unit of work.

3.5 Summary

Subject	P1525R0 coverage
Continuation framing	One incidental mention
`execute(F&&)` as work	Analyzed in full (Sections 2, 3, 4)
`async_result` / N3747	-
`dispatch`/`post`/`defer`	-
Framing change documentation	-
Networking use case	-

---

4. The Framing Dependency

P4094R0[2] Section 6.4 defined two framings of `execute(F&&)`:

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

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.

This section applies both framings to each of P1525R0[1]'s four deficiencies.

4.1 Error Propagation

P1525R0[1] Section 2.3:

> *"errors that happen after submission but before execution have no place in-band to go"*

Under the work framing, this is true. The caller submitted work and continued. The caller is alive, running, and expects to learn what happened. The error has no channel back to the caller.

Under the continuation framing, the caller returned. There is no live caller on the other end to receive a report. The executor accepted a resumption handle. The I/O has not completed. The operating system has not returned a result. There is no error code. There is no byte count. The only thing passed to the executor is: this is how you resume me. An error channel for a resumption handle would report on work that has not happened.

4.2 Cancellation

P1525R0[1] Section 3.3 lists five reasons a callable's destructor might be called:

> *"There are lots of reasons the destructors of a continuation might get called... Only for reason (4) should a destructor call be interpreted as the 'done' signal. In order to distinguish (4) from the other four cases, a continuation would need to keep state."*

The five reasons are: (1) post-value cleanup - the callable executed, produced a value, and is now being destroyed; (2) post-error cleanup - the callable executed, produced an error, and is now being destroyed; (3) the callable was moved from and the empty shell is being destroyed; (4) the callable was never executed - this is cancellation; (5) the callable was never executed because the executor is shutting down.

Under the work framing, the callable can be in any of these five states when its destructor runs. The destructor cannot distinguish them without extra state. The ambiguity is real.

Under the continuation framing, the callable is a continuation that has not yet been resumed. It has not executed. It has not produced a value. It has not produced an error. It is waiting. Reasons (1) and (2) - post-value and post-error cleanup - do not apply because the continuation has not been resumed and therefore has not produced a result. If the executor destroys a continuation without resuming it, the continuation was cancelled. There is only one state an un-resumed continuation can be in: waiting. The four-way ambiguity that P1525R0[1] describes does not arise because two of the five reasons - post-value cleanup and post-error cleanup - presuppose that the callable has already executed, and a continuation that has not been resumed has not executed.

4.3 Zero-Allocation Scheduling

P1525R0[1] Appendix C demonstrated a `thread_dispatcher` where `co_await ex.schedule()` embeds the queue entry in the coroutine frame:

> *"This allows the caller to place the state-machine for the async operation inline within the coroutine frame or as a member of some other object without forcing the state-machine to be heap-allocated."*

The coroutine executor's `dispatch(coroutine_handle<>)` achieves the same result. The awaiter that calls `dispatch` is a local variable in the coroutine frame. The queue entry is the awaiter. No type erasure. No heap allocation. The mechanism that P1525R0[1] demonstrated as a property of the Sender/Receiver model is also a property of the coroutine model - because both embed the operation state in the coroutine frame.

The allocation concern in P1525R0[1] Section 4.1 applies to `execute(F&&)` because the callable must be type-erased and heap-allocated. It does not apply to `dispatch(coroutine_handle<>)` because `coroutine_handle<>` is already a fixed-size, type-erased handle to a suspended coroutine.

4.4 The Asymmetry

P1525R0[1] Section 2.3.1:

> *"we cannot implement schedule generally in terms of one-way execute."*

Under the work framing, this is true. `execute` has no error channel, so `schedule`'s Sender cannot deliver errors to a Receiver.

Under the continuation framing, the executor does not need to implement `schedule`. The executor schedules continuations. Composition is provided by `async_result` (N3747[9]) on the completion-model axis and by `co_await` on the operation-composition axis. The asymmetry that P1525R0[1] identified is real under the work framing. Under the continuation framing, the question is different: the executor is not the composition mechanism.

4.5 Summary

Deficiency	Work framing	Continuation framing
Error propagation	Real. No channel back to caller after submission.	No error exists. The I/O has not completed. The handle carries a resumption, not a result.
Cancellation	Real. Destruction of untyped callable is ambiguous.	Un-resumed continuation has one state: waiting. Destruction is cancellation.
Zero-allocation	Real. `execute(F&&)` requires type erasure and heap allocation.	`coroutine_handle<>` is fixed-size. The awaiter embeds in the frame.
Asymmetry	Real. `execute` cannot implement `schedule`.	Different question. The executor is not the composition mechanism. `async_result` and `co_await` are.

The four deficiencies are real under the work framing. Under the continuation framing, three do not arise and the fourth addresses a different question. P1525R0[1] analyzed `execute(F&&)` under the work framing. The work framing was the only framing visible on the API surface in 2019.

---

5. The Cologne Pivot

In July 2019, SG1 acted on P1525R0[1]'s diagnosis.

5.1 What Happened

Three papers were published simultaneously in June 2019:

- P1525R0[1] (Niebler, Shoop, Baker, Howes): the diagnosis.
- P1658R0[12] (Hoberock, Lelbach): the prescription. "Eliminate OneWayExecutor, BulkOneWayExecutor, and interface-changing properties."
- P1660R0[13] (Hoberock, Garland, Lelbach, Dominiak, Niebler, Shoop, Baker, Howes, Hollman, Brown): the sketch. "An Executor concept based on a function named execute which eagerly submits a function for execution."

SG1 at Cologne (July 2019) directed P0443R11[11] to implement the changes. The P0443R11[11] changelog:

> *"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. Sender, Receiver, and Scheduler concepts were introduced. The executor concept was reduced to a single `execute` function. The Cologne pivot was the moment the committee adopted the sender/receiver model as the foundation for async programming in C++.

5.2 What the Published Record Does Not Contain

- No paper at Cologne analyzes P1525R0[1]'s diagnosis under the continuation framing.
- No paper asks whether the deficiencies of `execute(F&&)` are properties of the signature or properties of the work framing.
- No straw poll addresses whether the framing change - from continuation-scheduling to work-submission - is acceptable for networking.
- No paper evaluates what the Networking TS loses when the continuation property is removed from the executor concept.

The three papers that drove the pivot - P1525R0[1], P1658R0[12], P1660R0[13] - do not mention `async_result`, N3747[9], or the continuation framing. The pivot was driven by analysis of `execute(F&&)` under the work framing. That is the record.

---

6. The Coroutine Executor Under P1525R0's Criteria

Section 4 analyzed the two framings in the abstract. The coroutine executor concept (P4003R0[5]) makes the continuation framing concrete:

```
std::coroutine_handle<> dispatch(
    std::coroutine_handle<> h) const;

void post(std::coroutine_handle<> h) const;
```

The handle type constrains the callable to `coroutine_handle<>` - a fixed-size, type-erased handle with exactly two operations: `resume()` and `destroy()`. `post` throws `std::system_error` if scheduling fails; if an exception is thrown, the caller retains ownership. After ownership is transferred, the implementation resumes or destroys the handle. The awaiter that calls `dispatch` is a local variable in the coroutine frame; the queue entry is the awaiter. Composition is provided by `co_await` on the operation axis and by `async_result` (N3747[9]) on the completion-model axis.

Criterion	`execute(F&&)` (P1525R0)	`dispatch(coroutine_handle<>)`
Error propagation	No channel. Implementation-defined.	`post` throws on scheduling failure. No in-band channel needed.
Cancellation	Destruction ambiguous for untyped callable.	`destroy()` unambiguous. Two operations, two dispositions.
Zero-allocation	Requires type erasure and heap allocation.	Awaiter embeds in coroutine frame. Fixed-size handle.
Composition asymmetry	`execute` cannot implement `schedule`.	Does not arise. `co_await` and `async_result` compose.

7. Anticipated Objections

Q: P1525R0 was right - `execute` is a poor basis operation.

A: Under the work framing, the diagnosis is correct. This paper documents that the work framing is not the only valid reading of the operation, and that the continuation framing produces different conclusions. Sections 4 and 6 present the evidence.

Q: The coroutine executor did not exist in 2019.

A: Correct. C++20 coroutines were ratified in 2020. The coroutine executor concept was published in P4003R0[5] (2026). The analysis this paper provides was not available when P1525R0[1] was written. This paper does not argue that P1525R0[1]'s authors should have known. It documents what is now observable.

Q: The continuation framing is a retroactive reinterpretation.

A: P0113R0[8] (Kohlhoff, 2015) defined `defer` as scheduling "a continuation of the caller." P0688R0[10] (2017) converted it to a property hint. P4094R0[2] Section 6 documents every paper in the chain. The continuation framing is the original framing. The work framing is the replacement.

Q: P1525R0's authors had deep async expertise.

A: They did. The issue is not who the authors were but what was visible on the API surface they were analyzing. The continuation framing was not encoded in the type system, not enforced by concepts, and not visible in the `execute(F&&)` signature. P4094R0[2] Section 6.3 documents how design rationale is lost across paper boundaries in multi-author standardization. The framing was not in the paper P1525R0[1] was analyzing.

Q: This paper argues for the author's own library.

A: Section 1 discloses this. The evidence in Sections 3 through 6 stands or falls on the published record, not on who assembled it.

Acknowledgments

The author thanks Eric Niebler, Kirk Shoop, Lewis Baker, and Lee Howes for P1525R0, which identified genuine structural deficiencies in the executor concept and proposed the sender/receiver model that became P2300R10; Christopher Kohlhoff for the continuation framing, N3747, and the executor model that started the journey; Ville Voutilainen for P2464R0, which applied P1525R0's analytical framework to the Networking TS; Steve Gerbino and Mungo Gill for Capy and Corosio implementation work; and Jared Hoberock and Bryce Adelstein Lelbach for P1658R0 and the Cologne pivot.

References

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

[2] P4094R0 - "Retrospective: The Unification of Executors and P0443" (Vinnie Falco, 2026).

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

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

[5] P4003R0 - "Coroutines for I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[6] P0443R10 - "A Unified Executors Proposal for C++" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, H. Carter Edwards, Gordon Brown, David Hollman, 2019).

[7] P2300R10 - "std::execution" (Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 2024).

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

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

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

[11] 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).

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

[13] 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).