Document	**D0000**
Date:	2026-02-09
Reply-to:	Vinnie Falco <vinnie.falco@gmail.com>
Audience:	All of WG21

On Universal Models

Abstract

Software engineering has a recurring pattern: smart people see commonality across domains and conclude that one abstraction should serve them all. Sometimes they are right. TCP/IP and IEEE 754 are genuine universal models that emerged from practice and proved themselves across decades of deployment. More often they are wrong, and the universal framework loses to pragmatic, specialized alternatives. This paper examines the evidence for and against a universal execution model in C++, focusing on `std::execution` (P2300 (https://wg21.link/p2300)). It proposes no wording changes. It asks the committee to consider whether the evidence supports the current direction, or whether specialization with interoperation might serve the C++ community better.

1. Introduction

The desire for a universal model is one of the most natural instincts in software design. A programmer looks at callbacks, futures, coroutines, and sender/receiver pipelines and thinks: these are all doing the same thing. Surely one abstraction can unify them.

That instinct is often productive. But it has a well-documented failure mode.

Joel Spolsky identified it in 2001, coining the term “architecture astronaut”:

> *"When you go too far up, abstraction-wise, you run out of oxygen. Sometimes, smart thinkers just don't know when to stop, and they create these absurd, all-encompassing, high-level pictures of the universe that are all good and fine, but don't actually mean anything at all."*
>
> *Joel Spolsky, "Don't Let Architecture Astronauts Scare You" (https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/) (2001)*

The MIT Exokernel paper put the same observation in more formal terms:

> *"It is fundamentally impossible to define abstractions that are appropriate for all areas and implement them efficiently in all situations."*
>
> *Engler et al., "Exokernel: An Operating System Architecture for Application-Level Resource Management" (https://people.eecs.berkeley.edu/~brewer/cs262b/hotos-exokernel.pdf) (1995)*

And Ted Elliot captured the tradeoff precisely:

> *"An all-powerful abstraction is a meaningless one. You've just got a new word for 'thing'."*
>
> *Ted Elliot, "The One Ring Problem" (https://tedinski.com/2018/01/30/the-one-ring-problem-abstraction-and-power.html) (2018)*

None of this means universal models are impossible. They exist and they matter. But the history of computing suggests they are rare, and that the instinct to create them runs far ahead of the evidence needed to justify them.

This paper examines `std::execution` through that lens. The people who designed it are talented. The work they have done contains genuine insights. The question is whether the evidence supports declaring it the universal execution model for C++, or whether the direction has gotten ahead of itself in a domain that is genuinely difficult.

The asymmetry of risk makes this question worth asking. If `std::execution` is truly universal, it will prove itself through adoption and this paper will have been an unnecessary caution. If it is not universal and we mandate it anyway, the cost compounds for decades. The C++ standard still cannot connect to the internet. Getting the execution model wrong makes that problem harder, not easier.

2. The Historical Record

2.1 OSI vs TCP/IP

The Open Systems Interconnection model is the canonical example of a universal framework that lost to a pragmatic alternative.

OSI was designed by talented, well-intentioned engineers solving a real problem. Its seven-layer architecture was comprehensive, well-documented, and backed by international standards bodies, governments, and major corporations. By the mid-1980s, its adoption appeared inevitable. As the IEEE Spectrum article (https://spectrum.ieee.org/osi-the-internet-that-wasnt) documents, "The Defense Department officially embraced the conclusions of a 1985 National Research Council recommendation to transition away from TCP/IP and toward OSI," and "the Department of Commerce issued a mandate in 1988 that the OSI standard be used in all computers purchased by U.S. government agencies."

And yet TCP/IP won. Einar Stefferud captured the outcome:

> *"OSI is a beautiful dream, and TCP/IP is living it!"*
>
> *IEEE Spectrum, "OSI: The Internet That Wasn't" (https://spectrum.ieee.org/osi-the-internet-that-wasnt) (Andrew Russell, 2013)*

The same article identifies what went wrong:

> *"Openness and modularity, the key principles for coordinating the project, ended up killing OSI."*

Louis Pouzin, a veteran of the effort, observed in 1991 (quoted in the same article (https://spectrum.ieee.org/osi-the-internet-that-wasnt)):

> *"Government and corporate policies never fail to recommend OSI as the solution. But, it is easier and quicker to implement homogeneous networks based on proprietary architectures, or else to interconnect heterogeneous systems with TCP-based products."*

OSI did not fail because its designers were incompetent. It failed because the universal framework was too heavy to compete with pragmatic, deployed alternatives. The comprehensive design that looked like a strength on paper became a liability in practice.

A natural objection is that OSI failed simply because TCP/IP had too much momentum — an installed base that was too large to displace. But OSI had enormous institutional momentum of its own. The U.S. government mandated it for procurement (GOSIP (https://en.wikipedia.org/wiki/Government_Open_Systems_Interconnection_Profile), 1988). European governments imposed similar requirements. ISO, the ITU, and major corporations backed it (IEEE Spectrum (https://spectrum.ieee.org/osi-the-internet-that-wasnt)). If momentum were the deciding factor, the mandates should have worked. They didn't. TCP/IP kept winning despite active institutional resistance, because it was simpler to implement, faster to deploy, and cheaper to maintain. The momentum TCP/IP accumulated was a consequence of its narrow, pragmatic design — not an independent variable that happened to favor it.

2.2 "Everything Is an Object"

Object-oriented programming went through a similar cycle. The claim that "everything is an object" positioned OOP as a universal modeling philosophy. Richard Gabriel argued at OOPSLA 2002 that this claim gave OOP a privileged position that starved research into alternative paradigms (Gabriel, "Objects Have Failed" (https://dreamsongs.com/Files/ObjectsHaveFailed.pdf)).

The Gang of Four's *Design Patterns* (1994) articulated the corrective: "Favor object composition over class inheritance." The industry eventually learned that deep inheritance hierarchies were brittle and that composition through narrow interfaces produced better designs (Wikipedia: Composition over inheritance (https://en.wikipedia.org/wiki/Composition_over_inheritance)).

2.3 The Pattern

Universal models designed top-down consistently lose to specialized models that emerge from practice. This is not a new observation. The question is whether C++ async is repeating it.

3. Universal Models in C++

3.1 The “Grand Unified Model” Vote

On 2021-09-28, SG1 polled:

> *“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.”*

The result was **no consensus:** 4 SF, 9 WF, 5 N, 5 WA, 1 SA. P2453R0 (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2453r0.html) documents this poll and its interpretation. The committee did not achieve consensus that a universal model was even needed. Yet the direction proceeded as though it had.

This is worth pausing on. The foundational premise of the current direction did not achieve consensus among the people voting on it.

3.2 Ranges

Ranges are the most recent precedent for a universal abstraction in C++. They were positioned as the universal iteration and algorithm model. Adoption tells a more nuanced story.

Google bans `<ranges>` from most of its codebase. Daisy Hollman (then at Google) explained at CppCon 2024 (https://cppcon2024.sched.com/event/1gZgc/why-google-doesnt-allow-ranges-in-our-codebase) that ranges are “perhaps the largest and most ambitious single feature ever added to the C++ standard library” but “it’s unreasonable to expect that those trade-offs will result in the same cost-benefit ratio in every context.”

Compile-time overhead is substantial. Range-V3 headers compile in 3.44 seconds versus 0.44 seconds for STL `<algorithm>`, roughly an 8x slowdown (NanoRange wiki: Compile times (https://github.com/tcbrindle/NanoRange/wiki/Compile-times)). Deeply nested range adapters exhibit a “cubic stack blowup” in template instantiation (Hacker News discussion (https://news.ycombinator.com/item?id=40317350)). Daniel Lemire’s analysis suggests that “std::ranges may not deliver the performance that you expect” (https://lemire.me/blog/2025/10/05/stdranges-may-not-deliver-the-performance-that-you-expect/) (2025).

Ranges brought real value to the language. But the pattern of a universal abstraction that proved too costly for the largest codebases is worth noting as precedent.

3.3 IETF TAPS

The IETF Transport Services (TAPS) initiative represents a top-down universality attempt for transport protocols. The working group was chartered in 2014. After a decade, the core documents remain Internet-Drafts. The TAPS charter page (https://datatracker.ietf.org/wg/taps/about/) notes:

> *"TAPS has delivered all the deliverables it was chartered for... With a hope that TAPS will be deployed gradually."*

After a decade of work, deployment remains aspirational.

One proprietary implementation exists: Apple's Network.framework (2018). P3482R1 (https://wg21.link/p3482) acknowledges: "Unfortunately, at present, Apple's Network Framework is the only such implementation." One open-source implementation, NEAT (EU Horizon 2020 project), was active from 2015 to 2018 and abandoned when EU funding ended (https://www.neat-project.org/).

P3482 (https://wg21.link/p3482) (Rodgers & Kühl) proposes basing C++ networking on TAPS. The pattern is familiar: abstract away protocol selection entirely, let the system choose. But web servers need TCP. DNS needs UDP. The use cases dictate the protocol.

TAPS contains useful analysis of transport protocol capabilities. But as a foundation for C++ networking, it follows the same top-down universality pattern that OSI exemplified.

3.4 The Networking TS Schism

The Networking TS was really two things bundled together: an execution model (io_context, completion handlers, executors) and portable wrappers for I/O objects (sockets, timers, etc.). The controversy was never about the sockets. It was entirely about the execution model.

The polls in P2453R0 (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2453r0.html) make this clear:

- **Poll 1** asked whether the Networking TS/Asio async model is "a good basis for most asynchronous use cases, including networking, parallelism, and GPUs." Result: weak consensus against. But the interpretation noted: "It doesn't mean that the Networking TS async model isn't a good fit for networking. There were many comments to the contrary."
- **Poll 3** asked to "Stop pursuing the Networking TS/Asio design as the C++ Standard Library's answer for networking." Result: **no consensus** (13 SF, 13 WF, 8 N, 6 WA, 10 SA). The Networking TS is not dead.

- **Poll 5** asked whether it is "acceptable to ship socket-based networking… that does not support secure sockets." Result: **no consensus**. No objection to the sockets themselves, only to shipping without TLS.

Nobody objected to portable socket wrappers. The schism was entirely about which execution model async operations should use. The I/O objects were uncontroversial because they did not claim universality.

3.5 std::execution (P2300)

`std::execution` is the central case study. It proposes sender/receiver as the universal async model for C++. The authors are talented, the structured concurrency ideas have real value, and the work represents years of effort. The question is whether the evidence supports declaring it universal.

3.5.1 GPU-Oriented Design

The P2300R10 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html) author list draws predominantly from GPU and large-scale compute companies. The reference implementation, `stdexec`, is hosted at nvidia.github.io (https://nvidia.github.io/stdexec/). No author of a shipping, production-deployed networking library appears on the paper. This matters because the design machinery reflects the domains its authors work in:

- P2300R10 §1.1 frames the motivation around "GPUs in the world's fastest supercomputer."
- P2300R10 §1.2 prioritizes "the diversity of execution resources and execution agents, because not all execution agents are created equal."
- The second end-user example (§1.3.2) is "Asynchronous inclusive scan," a classic GPU parallel primitive using `bulk` to spawn data-parallel execution agents. This is not an I/O pattern.
- `bulk` (§4.20.9) spawns N execution agents for data-parallel work. No networking analog exists.
- `continues_on` / transfer moves work between execution contexts, a CPU-to-GPU pattern. Networking does not transfer between hardware backends.
- Completion domains dispatch algorithms based on execution resource, so GPU backends can substitute custom implementations. TCP reads have one implementation per platform, not multiple hardware backends.

The entire sender algorithm customization lineage (P2999R3 (https://wg21.link/p2999r3), P3303R1 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3303r1.html), P3826 (https://wg21.link/p3826)) is about domain-based algorithm dispatch. These papers contain zero mentions of networking, sockets, or I/O.

Among the 10 companion papers in flight (see §3.5.5), the authors come from NVIDIA, Intel, Meta, or are working on `std::execution` machinery itself. When a framework claims universality across domains, the absence of I/O and networking domain experts from its design process is worth noting.

3.5.2 Networking Deferred

The official stdexec documentation states under "Standardization Status (as of 2025)":

> *"Interop with networking is being explored for C++29."*
>
> *nvidia.github.io/stdexec (https://nvidia.github.io/stdexec/)*

The framework ships in C++26 without the use case that started the entire executor discussion a decade ago.

stdexec's only I/O example (io_uring.cpp (https://github.com/NVIDIA/stdexec/blob/main/examples/io_uring.cpp)) contains zero socket operations, zero reads, zero writes. It demonstrates only timers (`schedule_after` , `schedule_at`). When a user asked about file reading, the maintainer directed them to a third-party repo (NVIDIA/stdexec#1062 (https://github.com/NVIDIA/stdexec/issues/1062)). The framework has never been proven for I/O in practice.

3.5.3 Type Erasure and ABI

libunifex issue #244 (https://github.com/facebookexperimental/libunifex/issues/244) (still open) shows a user trying to implement networking with SSL sockets who hits a fundamental type-erasure wall. Eric Niebler responds:

> *"There currently isn't a generalization of `any_sender_of<>` than can handle more than `std::exception_ptr` ."*

Lewis Baker proposes a fix:

> *"Longer term, it probably makes sense to allow the any_sender_of type to be parameterisable with both a list of set_value overloads and a list of set_error overloads."*

The fix was never implemented. C++26 `std::execution` provides no type-erased sender at all. This matters for three reasons.

ABI stability. WG21 has maintained ABI stability across C++14, C++17, and C++20 (P1863R1 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1863r1.pdf)). Breaking ABI costs the ecosystem "engineer-millennia." Templates and header-only libraries create ABI fragility because "template implementations must appear in headers, forcing recompilation into every translation unit that uses them" (Gentoo C++ ABI analysis (https://blogs.gentoo.org/mgorny/2012/08/20/the-impact-of-cxx-templates-on-library-abi/)). Type-erased interfaces provide stable ABI boundaries: implementation changes stay behind the erasure wall, no recompilation needed. A framework that structurally resists type erasure maximizes ABI risk. A design that embraces type erasure, like coroutine handles which are inherently type-erased, would give the committee what it wants: async capabilities with ABI stability. This is a design that solves the committee's own stated problem.

Error model mismatch. Networking code vastly prefers `error_code` over exceptions. Boost.Asio's own documentation explains why EOF is an error_code: "An EOF error may be used to distinguish the end of a stream from a successful read of size 0" (Boost.Asio Design (https://www.boost.org/doc/libs/release/doc/html/boost_asio/design/eof.html)). Conditions like cancellation and EOF are alternate success states, not catastrophic failures. In libunifex #244 (https://github.com/facebookexperimental/libunifex/issues/244), Niebler confirms `any_sender_of` is hard-coded to `exception_ptr` while io_uring propagates `error_code`. The framework that claims universality produces a type-erased form that is itself not universal.

Compile times. Asio issue #1100 (https://github.com/chriskohlhoff/asio/issues/1100) is a feature request for type-erased handlers. The author states: "Some of us still care about compile times and being able to apply the Pimpl idiom. This is not possible when our libraries are forced to be header-only because of Asio." Kohlhoff responded by implementing `any_completion_handler`. A framework that structurally resists type erasure forces the template-heavy model on every user.

3.5.4 Design Immaturity

D4007R0 (https://wg21.link/p4007) ("std::execution Needs More Time") documents a fundamental timing gap between the backward-flow context model and coroutine frame allocation. Eric Niebler, in P3826R3 (https://wg21.link/p3826) (2026-01-05), characterizes part of the design in his own words:

> *"The receiver is not known during early customization. Therefore, early customization is irreparably broken."*

When the lead author of a framework describes part of its design as "irreparably broken," that is evidence worth weighing.

3.5.5 Papers Still in Flight

At least 10 companion papers are actively fixing, extending, and patching `std::execution` in the 2025-2026 mailings. Each reveals design immaturity, GPU/parallel focus, or both:

- P2079 (https://wg21.link/p2079) "System execution context": A global thread pool for parallel forward progress. Explicitly GPU/parallel: "System scheduler works best with CPU-intensive workloads." I/O is deferred as future work.
- P3164 (https://wg21.link/p3164) "Improving diagnostics for sender expressions" (Niebler): Fixes a design flaw where type errors in sender expressions are diagnosed too late.
- P3373 (https://wg21.link/p3373) "Of Operation States and Their Lifetimes" (Leahy): Reveals that basic lifetime semantics were not properly designed. The paper observes: "the analogue thereof for asynchronous code under the framework of std::execution moves in only one direction (i.e. such storage is only ever 'allocated')."
- P3388 (https://wg21.link/p3388) "When Do You Know connect Doesn't Throw?" (Leahy): The framework cannot answer a basic question: will `connect` throw? LEWG voted unanimously (SF:8 F:7 N:0 A:0 SA:0) that this is a real deficiency.
- P3425 (https://wg21.link/p3425) "Reducing operation-state sizes" (Baker): Operation states are bloated. Filed as a national body comment against C++26. The Hagenberg review noted no deployment experience. Real-world frameworks like HPX and pika already solved this problem. `std::execution` regressed relative to existing practice.
- P3481 (https://wg21.link/p3481) "std::execution::bulk() issues": Explicitly GPU-focused: "For GPUs is important to have a version of `bulk` that ensures one execution agent per iteration." Fundamental API gaps remain: "the following are unclear: Can `bulk` invoke the given functor concurrently? Can `bulk` create decayed copies?"
- P3552 (https://wg21.link/p3552) "Add a Coroutine Task Type" (Kühl, Nadolski): `std::execution` shipped in C++26 without a coroutine task type, the primary way users are expected to interact

with it. The paper lists 12 unsolved design objectives.

- P3557 (https://wg21.link/p3557) "High-Quality Sender Diagnostics" (Niebler): Sender misuse produces "megabytes of incomprehensible diagnostics." The paper "exposed several bugs in the Working Draft for C++26."

- P3564 (https://wg21.link/p3564) "Make concurrent forward progress usable in bulk" (Hoemmen, NVIDIA): Explicitly GPU: "CUDA distinguishes ordinary bulk execution ('device kernel') launch… from so-called 'cooperative' bulk execution launch." Forward progress guarantees are fundamentally broken.

- P3826 (https://wg21.link/p3826) "Fix Sender Algorithm Customization" (Niebler): Predecessor papers P2999R3 (https://wg21.link/p2999r3) and P3303R1 (https://wg21.link/p3303r1) address domain-based algorithm dispatch for GPU backends.

Of these 10 papers, zero are about networking. At least four are explicitly GPU/parallel focused (P2079, P3481, P3564, P3826). The rest fix fundamental design deficiencies: lifetimes, exception safety, diagnostics, memory bloat, and a missing task type. A framework with this many open design questions is not ready to be declared universal.

3.5.6 The Design Space Remains Open

D4003 (https://wg21.link/p4003) ("IoAwaitables: A Coroutines-Only Execution Model") demonstrates an alternative execution model purpose-built for I/O that diverges significantly from `std::execution`. Its existence proves the design space has not converged.

If WG21 commits to one execution model as universal, it may close off the design space for alternatives like IoAwaitables, TooManyCooks (https://github.com/tzcnt/TooManyCooks) (a C++20 coroutine runtime optimized for raw performance), and Asio's completion token model. The history of C++ suggests that enabling multiple approaches, and letting the ecosystem converge naturally, has served the language better than mandating convergence from above.

4. Narrow Abstractions Win

C++ has a strong track record with universal abstractions. But the ones that succeed share a distinctive property: they are narrow.

4.1 Where Universality Succeeded in C++

STL Iterators. Stepanov's traversal protocol works across every container type. His key insight: "One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types" (Stepanov, "The Standard Template Library," 1994 (https://stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf)). Iterators capture one essential property, traversal, and leave everything else to the user. The same `std::sort` works on arrays, vectors, and deques. It works because the abstraction is narrow.

RAII. Constructor acquires, destructor releases. This pattern works across every resource type: memory, files, sockets, locks, GPU handles. Bjarne Stroustrup introduced it in the 1980s as a way to make resource management exception-safe (cppreference: RAII (https://en.cppreference.com/w/cpp/language/raii)). It emerged from practice, not from committee design. It is minimal, stable, and universal across every C++ domain. It works because the abstraction is narrow.

Allocators. The allocator model lets every standard container work with any memory strategy: pool allocators, arena allocators, `std::pmr` resources, or the default heap. Containers become composable building blocks regardless of the memory strategy underneath (cppreference: Allocator (https://en.cppreference.com/w/cpp/named_req/Allocator)). It works because the abstraction is narrow.

Iterators abstract over traversal. Allocators abstract over memory strategy. RAII abstracts over resource lifetime. Each captures one essential property and leaves everything else to the user.

`std::execution` tries to abstract over scheduling, context propagation, error handling, cancellation, algorithm dispatch, and hardware backend selection all at once. That is not one essential property. That is six.

4.2 Should C++ Choose Tradeoffs for Its Users?

C++ has historically given programmers control over their tradeoffs. You don't pay for what you don't use. You choose the abstractions appropriate to your domain. The narrow abstractions above embody this principle: iterators don't choose your container, allocators don't choose your memory strategy, RAII doesn't choose your resource.

A mandated universal execution model would represent a departure from this tradition. It would say: we have decided which tradeoffs are right for async, across all domains, for all users.

Perhaps that is the right call. But consider the evidence.

4.3 Multiple Valid Execution Models in the Wild

In async, no similarly narrow universal abstraction has emerged. Instead, multiple valid execution models coexist, each optimized for different tradeoffs:

TooManyCooks (https://github.com/tzcnt/TooManyCooks) is a C++20 coroutine runtime with its own execution model (work-stealing thread pool, `tmc::task`, `tmc::spawn_tuple`), optimized for raw coroutine throughput via continuation stealing. It does not use `std::execution`. It is equally valid for its domain.

Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) **/ Networking TS** has its own execution model, optimized for I/O completion with platform-native proactors (IOCP on Windows, epoll on Linux, kqueue on BSD). It supports multiple continuation styles (callbacks, futures, stackful coroutines, C++20 coroutines) and user-defined ones through completion tokens. Kohlhoff's N3747 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf) ("A Universal Model for Asynchronous Operations," 2013) showed how a single initiating function can adapt to caller-chosen completion styles. This is a different kind of universality, proven across two decades of production use.

Capy (https://github.com/cppalliance/capy) has its own execution model, optimized for coroutine-first ergonomics with forward-flowing context (executor, allocator, stop token propagated from caller to callee). Corosio (https://github.com/cppalliance/corosio) uses it. Even within the same domain (async I/O), multiple valid execution models coexist because they optimize for different things.

gRPC (https://github.com/grpc/grpc), 44.3k GitHub stars. Google's production RPC framework, written primarily in C++. It has its own async model with both completion-queue-based and callback-based (https://grpc.io/docs/languages/cpp/callback) APIs. It does not use `std::execution`.

libuv (https://github.com/libuv/libuv), 26.5k GitHub stars. The event-driven async I/O library that powers Node.js. It provides its own event loop backed by epoll, kqueue, IOCP, and event ports, with cross-platform support for TCP/UDP sockets, DNS, file I/O, IPC, and child processes (libuv docs (https://docs.libuv.org/en/stable)). It does not use `std::execution`.

Seastar (https://github.com/scylladb/seastar), 9.1k GitHub stars. The framework behind ScyllaDB. Seastar uses a shared-nothing design that "shards all requests onto individual cores" with "explicit message passing rather than shared memory between threads" (seastar.io (https://www.seastar.io/)). Its futures-and-promises execution model is fundamentally incompatible with sender/receiver. It does not use `std::execution`.

cppcoro (https://github.com/lewissbaker/cppcoro), 3.8k GitHub stars. A C++ coroutine abstractions library created by Lewis Baker, who is a co-author of P2300 (https://wg21.link/p2300). It provides its own task types, schedulers, and async primitives. It does not use `std::execution`. When a co-author of the framework builds a separate library with a different execution model, that is evidence the design space has not converged.

The existence of these models is not a failure to be corrected. It is evidence that the problem space is too rich for a single wide model to dominate.

If the cost of mandating the wrong model is borne by every C++ programmer for decades, this question deserves careful, unhurried consideration.

5. What Works and What Doesn't

Genuine universal models do exist. It would be intellectually dishonest to ignore them. But they share characteristics that distinguish them from the models that fail.

TCP/IP. David Clark (1988) describes how TCP/IP's design philosophy evolved through "the repeated pattern of implementation and testing that occurred before the standards were set" (Clark, "The Design Philosophy of the DARPA Internet Protocols" (https://www.cs.princeton.edu/~jrex/teaching/spring2005/reading/clark88.pdf)). Key features like the datagram service and the IP/TCP layering "were not part of the original proposal" but emerged from iterative deployment. The standard formalized what had already proven successful in operational use.

IEEE 754. Before the standard, floating-point arithmetic was chaos. William Kahan recalls numbers that "could behave as non-zero in comparisons but as zeros in multiplication" (Kahan interview (https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html)). IEEE 754 emerged from Intel's practical need for a coprocessor, designed by Kahan based on decades of hands-on experience with IBM, Cray, and CDC systems. It codified best practices from existing hardware, not theoretical ideals.

Both emerged from practice, not from committee-designed frameworks. Both were minimal and stable. Both achieved broad voluntary adoption across disparate domains without coercion. Both proved themselves through deployment before being standardized.

The models that fail look different. They are wide, comprehensive, and designed top-down. The models that succeed alongside them look different too. They are narrow, specialized, and composable.

Unix pipes. Doug McIlroy (1978): "Make each program do one thing well... Expect the output of every program to become the input to another" (Wikipedia: Unix philosophy (https://en.wikipedia.org/wiki/Unix_p hilosophy)). The pipe, a byte stream, is the narrowest possible contract. It enabled an ecosystem of specialized tools that compose freely.

TCP/IP's narrow waist. The Internet's hourglass architecture puts a single, simple spanning layer (IP) at the center. "Simplicity and generality at the waist outperform richer, feature-heavy designs in real-world adoption and evolution" (Wikipedia: Hourglass model (https://en.wikipedia.org/wiki/Hourglass_model)). Innovation happens above and below the waist, not at the waist. OSI tried to make the waist wide. It failed.

Asio's completion tokens. Kohlhoff's N3747 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf) ("A Universal Model for Asynchronous Operations," 2013) showed how a single initiating function adapts to caller-chosen completion styles: callbacks, futures, stackful coroutines, C++20 coroutines, and user-defined ones. The narrow contract is the completion token. The model does not impose a continuation style; it lets each caller choose. Two decades of production use validate this approach.

The pattern is consistent. Narrow contracts enable broad ecosystems. Wide abstractions constrain them. When universality succeeds, it is minimal. When it fails, it is comprehensive.

If a universal model exists for async in C++, it will follow the practice-first pattern: broad voluntary adoption across disparate domains, without mandating it. If you have to mandate it, it is not universal. `std::execution` has not yet passed this test. If it does not, the alternative is not chaos. It is specialization with interoperation through narrow contracts, the approach that has worked everywhere else.

6. The Problem Is Already Solved

A key premise of `std::execution` is that C++ needs a standard framework for heterogeneous and parallel computing. But the ecosystem has not been waiting. The coordination problems that `std::execution` aims to solve are already solved by widely adopted, production-proven libraries.

6.1 GPU and Heterogeneous Computing

NVIDIA CCCL (Thrust, CUB, libcu++), 2.1k GitHub stars (GitHub (https://github.com/NVIDIA/cccl)). Thrust is described as "the C++ parallel algorithms library which inspired the introduction of parallel algorithms to the C++ Standard Library" (NVIDIA docs (https://nvidia.github.io/cccl/thrust/)). It is included in the NVIDIA HPC SDK and CUDA Toolkit. NVIDIA is not waiting for `std::execution` to ship GPU parallel algorithms. They already ship them.

Kokkos, 2.4k GitHub stars (GitHub (https://github.com/kokkos/kokkos)). A performance portability layer from Sandia National Laboratories that enables "manycore performance portability through polymorphic memory access patterns" (Kokkos documentation (https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Introduction.html)). Code written with Kokkos runs on CPUs, Intel Xeon Phi, and GPUs without platform-specific rewrites.

RAJA, 560 GitHub stars (GitHub (https://github.com/llnl/RAJA)). Lawrence Livermore National Laboratory's performance portability layer for DOE exascale applications. RAJA has been "proven in production with most LLNL ASC applications and numerous ECP applications" across NVIDIA, AMD, and Intel GPUs (LLNL project page (https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms)).

OpenMP accounts for "45% of all analyzed parallel APIs" on GitHub, making it the "dominant parallel programming model" with "steady and continuous growth in popularity over the past decade" (Quantifying OpenMP, 2023 (https://arxiv.org/pdf/2308.08002)). OpenMP 6.0 (November 2024) provides full GPU offload support (OpenMP 6.0 announcement (https://openmp.org/home-news/openmp-arb-releases-openmp-6-0-for-easier-programming)).

6.2 Task Parallelism and Async

Taskflow, 11.6k GitHub stars (GitHub (https://github.com/taskflow/taskflow)). "A General-purpose Task-parallel Programming System" that supports heterogeneous CPU-GPU computing and has demonstrated solving "a large-scale machine learning workload up to 29% faster, 1.5x less memory, and 1.9x higher throughput than the industrial system, oneTBB, on a machine of 40 CPUs and 4 GPUs" (Taskflow paper (https://arxiv.org/pdf/2004.10908)).

oneTBB (Intel), 6.5k GitHub stars (GitHub (https://github.com/uxlfoundation/oneTBB)). "A flexible performance library" for parallel computing that has been in production use for over 15 years (oneTBB documentation (https://uxlfoundation.github.io/oneTBB/)).

HPX, 2.8k GitHub stars (GitHub (https://github.com/STEllAR-GROUP/hpx)). "A general purpose C++ runtime system for parallel and distributed applications of any scale" that has demonstrated 96.8% parallel efficiency on 643,280 cores (HPX website (https://hpx.stellar-group.org/)).

folly (Meta), 30.2k GitHub stars (GitHub (https://github.com/facebook/folly)). Meta's production C++ library including `folly::Futures` and `folly::coro`, which power async operations across Meta's infrastructure at scale. Meta is not waiting for `std::execution`. They ship folly.

Abseil (Google), 17k GitHub stars (GitHub (https://github.com/abseil/abseil-cpp)). "The fundamental building blocks that underpin most of what Google runs," drawn from Google's internal codebase and "production-tested and fully maintained" (Abseil about page (https://abseil.io/about/)). Google is not waiting for `std::execution` either.

6.3 The Cost of Adding More

Every feature added to the C++ standard must be implemented and maintained by standard library vendors. There are exactly three major implementations: libstdc++ (GNU), libc++ (LLVM), and Microsoft's STL. Christopher Di Bella (Google, libc++ contributor) observes: "Due to its vast complexity, there have only been a handful of standard library implementations to date" (C++Now 2024 talk (https://www.youtube.com/watch?v=bXlm3taD6lw)).

The committee itself is strained. Bryce Adelstein Lelbach notes the committee has received "10x more proposals over the past decade" and describes it as "300 individual authors, not 1 team" (Convenor candidacy (https://brycelelbach.github.io/cpp_convenor/)). P2656R2 (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2656r2.html) observes that "the community is struggling to manage the challenges of the complexity and variability of the tools, technologies, and systems that make C++ possible."

David Sankel (Adobe) captured the risk in P3023R1 (https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/p3023r0.html):

> *"The surest way to sabotage a standard is to say yes to everything."*

Adding `std::execution` to the standard carries a cost. That cost must be weighed against the benefit.

6.4 The Cost of Not Standardizing Is Negligible

The strongest argument for standardization is that it solves a coordination problem: if everyone needs the same thing and nobody can agree on which library to use, the standard breaks the deadlock by picking one.

But no such coordination problem exists for `std::execution`.

The library is already available. `stdexec`, the reference implementation of `std::execution`, is available today as a standalone library (GitHub (https://github.com/NVIDIA/stdexec), nvidia.github.io/stdexec (https://nvidia.github.io/stdexec/)). It can be installed with a single command via vcpkg (https://vcpkg.link/ports/stdexec): `vcpkg install stdexec`. Anyone who wants sender/receiver can use it right now. Standardization adds nothing to its availability.

Package managers have eliminated the distribution problem. vcpkg offers over 2,600 C++ libraries (vcpkg.io (https://vcpkg.io/en/)). Conan hosts nearly 1,900 recipes with over 9,600 references (conan.io/center (https://conan.io/center)). The era when the standard library was the only reliable way to distribute a C++ library is over. Modern C++ projects routinely depend on dozens of third-party libraries obtained through package managers. `std::execution` on vcpkg gives developers the same access as `std::execution` in the standard, without burdening implementers.

Boost proved the model. Boost has achieved over 10 million downloads and is described by Herb Sutter and Andrei Alexandrescu as "one of the most highly regarded and expertly designed C++ library projects in the world" (boost.org (https://www.boost.org/users/)). Many Boost libraries thrive for years or decades without being standardized. Some are eventually adopted into the standard after proving themselves through widespread deployment. That is the TCP/IP pattern: prove it first, standardize it later.

Google proved it too. As noted in section 6.2, Google published Abseil (https://github.com/abseil/abseil-cpp) (17k GitHub stars) as a standalone open-source library, making their internal C++ building blocks available to everyone. As Abseil's documentation states, these are "the fundamental building blocks that underpin most of what Google runs," and they are "production-tested and fully maintained" (abseil.io/about (https://abseil.io/about/)). Google did not push Abseil through the C++ standard. They did not burden standard library implementers. They did not consume committee time. They simply published it, and the community adopted it voluntarily. Abseil is now available to every C++ programmer, without costing the standard a single page.

Nobody is blocked. NVIDIA ships CUDA. Meta ships folly. Google ships Abseil. Intel ships oneTBB. ScyllaDB ships Seastar. The ecosystem is not waiting for the standard to provide an execution model. Every major company that needs heterogeneous or parallel computing has already built or adopted one. Delaying `std::execution` from the standard does not prevent anyone from using it. It only prevents the standard from locking in a design before the evidence justifies it.

The asymmetry is stark. If `std::execution` is removed from C++26 and offered as a standalone library:

- Users who want it lose nothing. They install it from vcpkg.
- Implementers gain relief from a massive implementation burden.
- The design gains time to mature, fix the 10+ papers in flight, and prove itself across domains.

- The committee gains bandwidth for higher-priority work. Every major programming language ships networking in its standard library: Python (https://docs.python.org/3/library/socket.html), Java (https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/package-summary.html), Go (https://pkg.go.dev/net), Rust (https://doc.rust-lang.org/std/net/), C# (https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets), JavaScript/Node.js (https://nodejs.org/api/net.html). None of them ship a sender/receiver execution framework. Networking is the more universal need, and C++ is the only major language that still lacks it.

If `std::execution` stays in C++26 and the design proves wrong:

- Every C++ programmer inherits the cost.
- Three standard library teams must implement and maintain it indefinitely.
- The ABI is locked. Mistakes cannot be corrected without breaking the world.
- The standard's credibility is diminished.

The cost of including `std::execution` in the standard is real and permanent. The cost of not including it is negligible by comparison.

6.5 NVIDIA Already Ships Sender/Receiver for GPU Without the Standard

The GPU use case is the primary motivation for `std::execution`'s design. But NVIDIA already ships a complete sender/receiver GPU integration as a standalone library, and it requires their own non-standard compiler. The standard is not involved.

The code lives in `nvexec`, not `std::execution`. NVIDIA's stdexec repository contains a separate namespace, `nvexec`, with GPU-specific sender algorithms (github.com/NVIDIA/stdexec/tree/main/include/nvexec (https://github.com/NVIDIA/stdexec/tree/main/include/nvexec)). The files are `.cuh` (CUDA header) files, not standard C++ headers. They include GPU-specific reimplementations of `bulk`, `then`, `when_all`, `continues_on`, `let_value`, `split`, `reduce`, and more.

The GPU scheduler uses CUDA-specific types that standard C++ cannot express. The `stream_context.cuh` file (source (https://github.com/NVIDIA/stdexec/blob/main/include/nvexec/stream_context.cuh)) defines a `stream_scheduler` whose completion signatures include `cudaError_t`, a CUDA-specific error type:

```
using completion_signatures =
    STDEXEC::completion_signatures<set_value_t(),
        set_error_t(cudaError_t)>;
```

The scheduler's `schedule()` method is annotated with CUDA execution space specifiers:

```
STDEXEC_ATTRIBUTE(nodiscard, host, device) auto schedule() const noexcept {
    return sender{ctx_};
}
```

The `host, device` annotation maps to CUDA's `__host__ __device__`, a non-standard extension. The `stream_context` constructor calls `cudaGetDevice()`, a CUDA runtime API function.

GPU programming requires non-standard language extensions. NVIDIA's CUDA C/C++ Language Extensions (https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html) documentation enumerates the extensions that standard C++ cannot express:

- Execution space specifiers: "The execution space specifiers `__host__`, `__device__`, and `__global__` indicate whether a function executes on the host or the device."
- Memory space specifiers: "The memory space specifiers `__device__`, `__managed__`, `__constant__`, and `__shared__` indicate the storage location of a variable on the device."
- Kernel launch syntax: "The execution configuration is specified by inserting an expression in the form `<<<grid_dim, block_dim, dynamic_smem_bytes, stream>>>` between the function name and the parenthesized argument list."

None of these are valid C++. Code that uses them cannot be compiled by GCC, MSVC, or Clang without CUDA support. Every NVIDIA GPU user already depends on a non-standard compiler.

What this means. The primary use case for `std::execution`'s completion domains, `bulk` algorithm, and algorithm customization machinery is GPU dispatch. But GPU dispatch already works, today, in a standalone library (`nvexec`), distributed through stdexec, available on vcpkg, requiring NVIDIA's own compiler. Adding `std::execution` to the C++ standard does not change this. NVIDIA's users need `nvcc` regardless. The standard cannot express `__device__`, `__global__`, `<<< >>>`, or `cudaStream_t`. The GPU integration lives outside the standard by necessity, and it will continue to live outside the standard no matter what WG21 decides. The cost of standardization falls on implementers and the committee; the use case that motivates much of the design complexity cannot benefit from it.

7. Conclusion

The desire for a universal model is understandable. The people pursuing it are talented and well-intentioned. The work they have done has genuine value.

But the evidence presented in this paper suggests the direction may have gotten ahead of itself. The domain is genuinely difficult. It is no failure to acknowledge that and take more time.

The asymmetry of risk favors caution. As shown in section 6, delaying `std::execution` costs little. The library is on vcpkg today. The ecosystem is not waiting. But if we mandate a design that proves wrong, the cost compounds for decades. The C++ standard still cannot connect to the internet, and the only networking paper in flight (P3482 (https://wg21.link/p3482)) is based on IETF TAPS, a framework that was never adopted outside a single proprietary implementation. The path forward for networking is not just delayed. It is pointing in a direction that has already failed.

This paper asks the committee to consider, with open minds and good faith, whether the evidence supports the path we are on, or whether specialization with interoperation might serve the C++ community better.

References

1. Joel Spolsky. “Don’t Let Architecture Astronauts Scare You.” 2001. https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/
2. Engler et al. “Exokernel: An Operating System Architecture for Application-Level Resource Management.” MIT, 1995. https://people.eecs.berkeley.edu/~brewer/cs262b/hotos-exokernel.pdf
3. Ted Elliot. “The One Ring Problem.” 2018. https://tedinski.com/2018/01/30/the-one-ring-problem-abstraction-and-power.html
4. Andrew Russell. “OSI: The Internet That Wasn’t.” IEEE Spectrum, 2013. https://spectrum.ieee.org/osi-the-internet-that-wasnt
5. Richard Gabriel. “Objects Have Failed.” OOPSLA, 2002. https://dreamsongs.com/Files/ObjectsHaveFailed.pdf

6. Wikipedia. “Composition over inheritance.” https://en.wikipedia.org/wiki/Composition_over_inheritance
7. P2453R0. “2021 October Library Evolution and Concurrency Networking and Executors Poll Outcomes.” WG21, 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2453r0.html
8. CppCon 2024. “Why Google Doesn’t Allow Ranges in Our Codebase.” Daisy Hollman. https://cppcon2024.sched.com/event/1gZgc/why-google-doesnt-allow-ranges-in-our-codebase
9. NanoRange wiki. “Compile times.” https://github.com/tcbrindle/NanoRange/wiki/Compile-times
10. Daniel Lemire. “std::ranges may not deliver the performance that you expect.” 2025. https://lemire.me/blog/2025/10/05/stdranges-may-not-deliver-the-performance-that-you-expect/
11. IETF TAPS Working Group. Charter page. https://datatracker.ietf.org/wg/taps/about/
12. P3482R1. Rodgers & Kühl. “Design for C++ networking based on IETF TAPS.” WG21. https://wg21.link/p3482
13. NEAT Project. https://www.neat-project.org/
14. P2300R10. Dominiak et al. “std::execution.” WG21, 2024. https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html
15. Bryce Adelstein Lelbach. NVIDIA developer bio. https://developer.nvidia.com/blog/author/blelbach/
16. Eric Niebler. GitHub. https://github.com/ericniebler
17. Georgy Evtushenko. GitHub. https://github.com/gevtushenko
18. Lee Howes. Meta developer blog. https://developers.facebook.com/blog/post/2021/09/16/async-stack-traces-folly-Introduction/
19. P1241R0. “Merging Coroutines into C++.” WG21, 2018. https://open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1241r0.html
20. libunifex. Facebook Experimental. https://github.com/facebookexperimental/libunifex
21. stdexec. NVIDIA. https://nvidia.github.io/stdexec/

22. P2999R3. Niebler. “Sender Algorithm Customization.” WG21, 2023. https://wg21.link/p2999r3

23. P3303R1. Niebler. “Fixing Lazy Sender Algorithm Customization.” WG21, 2024. https://open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3303r1.html

24. P3826R3. Niebler. “Fix Sender Algorithm Customization.” WG21, 2026. https://wg21.link/p3826

25. D4007R0. Falco. “std::execution Needs More Time.” WG21, 2026. https://wg21.link/p4007

26. stdexec io_uring.cpp. https://github.com/NVIDIA/stdexec/blob/main/examples/io_uring.cpp

27. NVIDIA/stdexec#1062. “io_uring reading files.” https://github.com/NVIDIA/stdexec/issues/1062

28. libunifex issue #244. “Question about any_sender_of usage.” https://github.com/facebookexperimental/libunifex/issues/244

29. P1863R1. “ABI breakage.” WG21, 2020. https://open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1863r1.pdf

30. Gentoo. “The impact of C++ templates on library ABI.” 2012. https://blogs.gentoo.org/mgorny/2012/08/20/the-impact-of-cxx-templates-on-library-abi/

31. Boost.Asio. “Why EOF is an error.” https://www.boost.org/doc/libs/release/doc/html/boost_asio/design/eof.html

32. Asio issue #1100. “Feature request: Type-erased handler wrapper.” https://github.com/chriskohlhoff/asio/issues/1100

33. P2079. “System execution context.” WG21. https://wg21.link/p2079

34. P3164. “Improving diagnostics for sender expressions.” WG21. https://wg21.link/p3164

35. P3373. “Of Operation States and Their Lifetimes.” WG21. https://wg21.link/p3373

36. P3388. “When Do You Know connect Doesn’t Throw?” WG21. https://wg21.link/p3388

37. P3425. “Reducing operation-state sizes for subobject child operations.” WG21. https://wg21.link/p3425

38. P3481. “std::execution::bulk() issues.” WG21. https://wg21.link/p3481

39. P3552. “Add a Coroutine Task Type.” WG21. https://wg21.link/p3552

40. P3557. “High-Quality Sender Diagnostics with Constexpr Exceptions.” WG21. https://wg21.link/p3557

41. P3564. “Make the concurrent forward progress guarantee usable in bulk.” WG21. https://wg21.link/p3564

42. D4003. Falco et al. “IoAwaitables: A Coroutines-Only Execution Model.” WG21. https://wg21.link/p4003

43. TooManyCooks. https://github.com/tzcnt/TooManyCooks

44. N3747. Kohlhoff. “A Universal Model for Asynchronous Operations.” WG21, 2013. https://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf

45. Wikipedia. “Unix philosophy.” https://en.wikipedia.org/wiki/Unix_philosophy

46. Wikipedia. “Hourglass model.” https://en.wikipedia.org/wiki/Hourglass_model

47. David Clark. “The Design Philosophy of the DARPA Internet Protocols.” 1988. https://www.cs.princeton.edu/~jrex/teaching/spring2005/reading/clark88.pdf

48. William Kahan. “An Interview with the Old Man of Floating-Point.” https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html

49. Hacker News discussion on std::ranges. https://news.ycombinator.com/item?id=40317350

50. Boost.Asio. https://www.boost.org/doc/libs/release/doc/html/boost_asio.html

51. Capy. https://github.com/cppalliance/capy

52. Corosio. https://github.com/cppalliance/corosio

53. NVIDIA CCCL (Thrust, CUB, libcu++). https://github.com/NVIDIA/cccl

54. NVIDIA Thrust documentation. https://nvidia.github.io/cccl/thrust/

55. Kokkos. https://github.com/kokkos/kokkos

56. Kokkos Programming Guide: Introduction. https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Introduction.html

57. RAJA. Lawrence Livermore National Laboratory. https://github.com/llnl/RAJA

58. RAJA project page. LLNL. https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms

59. “Quantifying OpenMP: Statistical Insights into Usage and Adoption.” 2023. https://arxiv.org/pdf/2308.08002

60. OpenMP 6.0 announcement. https://openmp.org/home-news/openmp-arb-releases-openmp-6-0-for-easier-programming

61. Taskflow. https://github.com/taskflow/taskflow

62. Taskflow paper. https://arxiv.org/pdf/2004.10908

63. oneTBB (Intel). https://github.com/uxlfoundation/oneTBB

64. oneTBB documentation. https://uxlfoundation.github.io/oneTBB/

65. HPX. https://github.com/STEllAR-GROUP/hpx

66. HPX website. https://hpx.stellar-group.org/

67. folly (Meta). https://github.com/facebook/folly

68. Abseil (Google). https://github.com/abseil/abseil-cpp

69. Abseil about page. https://abseil.io/about/

70. Christopher Di Bella. “What Does It Take to Implement the C++ Standard Library?” C++Now 2024. https://www.youtube.com/watch?v=bXlm3taD6lw

71. Bryce Adelstein Lelbach. Convenor candidacy. https://brycelelbach.github.io/cpp_convenor/

72. P2656R2. “C++ Ecosystem International Standard.” WG21, 2023. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2656r2.html

73. P3023R1. Sankel. “C++ Should Be C++.” WG21, 2023. https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/p3023r0.html

74. Stepanov. “The Standard Template Library.” 1994. https://stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf

75. gRPC. https://github.com/grpc/grpc

76. gRPC C++ Callback API Tutorial. https://grpc.io/docs/languages/cpp/callback

77. libuv. https://github.com/libuv/libuv

78. libuv documentation. https://docs.libuv.org/en/stable

79. Seastar. https://github.com/scylladb/seastar

80. Seastar website. https://www.seastar.io/

81. cppcoro. Lewis Baker. https://github.com/lewissbaker/cppcoro

82. stdexec on vcpkg. https://vcpkg.link/ports/stdexec

83. vcpkg. Microsoft. https://vcpkg.io/en/

84. Conan Center. https://conan.io/center

85. Boost background information. https://www.boost.org/users/

86. cppreference. “RAII.” https://en.cppreference.com/w/cpp/language/raii

87. cppreference. “Allocator (named requirement).” https://en.cppreference.com/w/cpp/named_req/Allocator

88. Python standard library: socket. https://docs.python.org/3/library/socket.html

89. Java standard library: java.net. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/package-summary.html

90. Go standard library: net. https://pkg.go.dev/net

91. Rust standard library: std::net. https://doc.rust-lang.org/std/net/

92. C# standard library: System.Net.Sockets. https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets

93. Node.js standard library: net. https://nodejs.org/api/net.html

94. nvexec (NVIDIA GPU sender/receiver integration). https://github.com/NVIDIA/stdexec/tree/main/include/nvexec

95. nvexec stream_context.cuh source. https://github.com/NVIDIA/stdexec/blob/main/include/nvexec/stream_context.cuh

96. NVIDIA CUDA C/C++ Language Extensions. https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html