Master Evidence: What Belongs in the C++ Standard Library

0. The answer

What belongs in the C++ standard library?

Almost nothing, and the burden of proof sits on the proposer. The standard has a finite Complexity Budget, and the C++26 working draft already runs 2,679 pages, 77 percent of it library text. Admission is permanent: once a component enters, its API and ABI freeze, and from that day it stops competing with its downloadable rivals. std::regex takes 28 minutes on a grep task the system egrep finishes in 22 seconds. An unordered_map 200-300% faster is known, API-compatible, and "disallowed by ABI constraints". The committee's own papers call both "bad but insufficiently terrible" to remove. No working admission rule has ever policed the door: the direction paper concedes the founding criteria "didn't have much effect", and no standing document has replaced them in twenty-five years. So the default is no. A component earns yes by clearing four bars at once, proven in field reports from years of real deployment, because design is not evidence and utility alone never is:

- It solves a Coordination Problem, a concept everybody needs but every library implements differently, so its value scales with the square of adoption. string_view and format are the model.
- It passes the GitHub Test: standardization delivers something that downloading the library cannot.
- It passes the Reach Test: the constituency is a large share of the roughly ten million C++ developers, not a niche.
- It still pays after subtracting the Standardization Penalty it suffers on entry, the Interaction Tax it levies on everything standardized after it, and the committee time its admission consumes. Its Return on Complexity beats the best proposal the admission displaces.

In practice only vocabulary types, facilities that need compiler support, and thin portable OS abstractions clear these bars with any regularity. Everything else, the domain libraries and conveniences and "batteries", belongs in the package ecosystem, where it stays alive and keeps improving. A useful library that needs no standard is simply a useful library.

1. Executive summary

Every library proposal puts the same question to the committee: should this component be added to the C++ standard library? This file assembles the public evidence for answering that question, and the evidence supports one verdict: the default answer is no, and the burden of overwhelming proof sits on the proposer. The committee has never operated a working admission rule of its own; its direction paper concedes that the founding criteria for the standard library "didn't have much effect" (P0939R0, 2018), and section 4 shows that no standing document has supplied criteria in the twenty-five years since the library working group first said criteria would need to be developed. The costs of admission are measured and large: section 6 shows the working draft grown to 2,679 pages with the library holding 77 percent of the clause text, and section 8 shows component after component ceasing to compete with its downloadable rivals from the day it entered the standard. The benefits that survive those costs are rare: the one reliable class is the vocabulary component that solves a genuine Coordination Problem for a huge constituency, and section 5 documents both the few solved cases and the fragmentation that persists everywhere else.

The three strongest numbers in the file are these:

Number	Fact	Source
28 minutes vs 22 seconds	libc++ std::regex took 1,655 seconds on a grep task BSD egrep does in 21.92 and CTRE does in 11.11; all three vendors say the fix is ABI-blocked	P1433R0, 2019; section 8
200-300% disallowed	An API-compatible std::unordered_map that fast is known and "disallowed by ABI constraints"; the Prague 2020 polls then declined to break ABI	P1863R1, 2020; section 8
115 papers, 219 revisions, 0 implementations	The executors lineage 2012-2026, adopted June 2024 at St. Louis with a third of the plenary against, ships in no mainstream standard library as of mid-2026	wg21.link index; sections 6-8

The three strongest quotes are these:

> *We think these are reasonable criteria, but in practice they didn't have much effect.*

The Direction Group wrote this about the standard library's founding inclusion criteria (P0939R0, 2018).

> *Proposal authors need to be aware that as soon as something is standardized, it is essentially done. ... How many std::unordered_maps or std::regexes do we want in the standard library?*

Four authors from LEWG leadership published this warning in a numbered committee paper (P3001R0, 2023).

> *The Standard Library is the worst package manager, and Standard Library maintainers aren't domain experts.*

The principal maintainer of Microsoft's standard library wrote this in a public forum (Stephan T. Lavavej, r/cpp, 2024).

The value model compresses the whole framework into one admission rule: a component is admitted if and only if its Return on Complexity meets or beats lambda, which is the return of the best proposal its admission displaces (formulas.md).

The file reads as an inverted pyramid: the vocabulary and thesis come first (section 2), the value model and its evidence obligations follow (section 3), and the evidence mass descends from there - the committee's own criteria record from 2001 to 2026 (section 4), coordination problems and reach data (section 5), the priced Complexity Budget (section 6), factual dossiers on fourteen components (section 7), the Standardization Penalty record (section 8), corroborating voices (section 9), cross-language records (section 10), the counterargument inventory (section 11), and the source index (section 12). The file holds one hundred fifty evidence cards, selected from roughly five hundred gathered in the 2026-07-08 research campaign, and every card links to its public source.

2. Vocabulary

The campaign runs on eight coined terms, and every section of this file uses them with exactly these meanings:

Term	Meaning
Coordination Problem	The class of problem that justifies standardization: a concept everybody needs but every library implements differently
Complexity Budget	The finite resource of specification the standard can carry; every addition spends it
The GitHub Test	The threshold question: what does standardization deliver that downloading the library does not
Reach Test	The audience question: how large is the constituency that collects the benefit
Return on Complexity (ROC)	Value delivered per unit of Complexity Budget spent
Interaction Tax	The permanent ongoing cost each component imposes on everything standardized after it
Standardization Penalty	The immediate loss of value on entry: the component can no longer compete or evolve in the marketplace
Standardization Dividend	The net payout to the community: gross benefit times reach, minus committee cost, Penalty, and Tax

The thesis, stated once and in full, is this. Every library proposal asks the committee to spend Complexity Budget on a new component, and the default answer to that request is no. A component earns yes only through overwhelming evidence. It must solve a Coordination Problem, meaning a concept that everybody needs but every library implements differently. It must pass the GitHub Test by delivering value that downloadable availability alone cannot deliver. It must pass the Reach Test, because only a huge constituency can justify a permanent claim on the standard. It must still pay for itself after subtracting the Standardization Penalty it will suffer on entry, the Interaction Tax it will levy on everything standardized after it, and the fixed cost of the committee time its admission consumes. And its proposal must be built on field reports from years of real deployment, because design is not evidence. Utility alone is never evidence of a need for standardization; a useful library that needs no standard is simply a useful library.

3. The model

The value model turns the thesis into arithmetic. The full statement of the model, with its variables, its formulas, the scaling law, and the ordinal sufficiency principle, lives in formulas.md; the compressed form follows here. A component's Standardization Dividend is the discounted stream of its benefit beyond downloadability multiplied by its reach, reduced by the Standardization Penalty (the fraction of value forfeited to ossification, which grows with the domain's evolution velocity and the length of the freeze horizon), charged annually with the Interaction Tax the component levies on the rest of the standard, and reduced once by the fixed cost of the committee time its admission consumes. Because the Complexity Budget is finite, the admission rule is comparative rather than absolute: admit a component if and only if its Return on Complexity meets or beats lambda, the return of the best proposal the admission displaces. Vocabulary components are the only class that reliably clears this bar, because their value scales with the square of adoption: the value lives in every pair of independently authored libraries that can now interoperate. Convenience components scale linearly with adoption, and they usually fail the GitHub Test outright.

Every variable in the model translates into an evidence obligation on the proposing paper:

Requirement	Establishes	Form
Field reports from years of real deployment	B (benefit beyond availability) and the domain velocity behind the Penalty	The bulk of the paper
Reach census with scaling class	N, and whether value scales as N or N squared	Surveys, dependency counts, cross-library interface evidence
Complexity estimate	k (Complexity Budget spend)	Wording size, name count, interaction survey
Docket comparison	lambda (the shadow price)	Why this proposal over the alternatives competing for the same budget

A paper that consists only of design supplies none of these obligations, because design is not evidence.

Three workspace documents supply the analytical frameworks that this file deliberately does not restate. The A-G evidence checklist and the five-rung Adoption Ladder are developed in `my-books/wg21-bible/chapters/part-6.md`. The quadratic-conversion statement of the Coordination Problem is developed in `my-books/wg21-bible/chapters/part-3.md`, section 3.5. The N-squared interaction model behind the Interaction Tax is developed in `umbra/campaigns/reform-wg21/master-intelligence-wg21.md`, section 28. This file supplies the public evidence, and those documents supply the frames.

4. The committee's own criteria on record, 2001-2026

This section documents a pattern in the committee's own record. The stated criteria for admission to the standard library were always "existing practice" plus proposal quality. Those criteria were never operationalized into an admission rule. The committee's own direction papers say the criteria had no effect. The divergence between statement and practice is visible in the public vote record.

2000-2005: the TR1 era

The TR1 work item was chartered on "existing practice" from day one in 2000. The new work item proposal that created the Library TR, N1283, "A New Work Item Proposal: Technical Report for Library Issues", written by Nicolai Josuttis and dated 2000-10-26, states its motivation in three lines:

> *Motivation: to encapsulate existing practice; to focus attention on development of appropriate APIs; to provide logical additions to the container and algorithm parts of the C++ Standard Library*

The earliest criteria document, N1314 in 2001, explicitly refused to define what was out of scope. The document is N1314, "Notes on Standard Library Extensions", written by Matt Austern and dated 2001-05-17, and it records the working group's position:

> *At this early stage, the library working group was reluctant to identify any possible directions that would definitely be ruled out.*

The evaluation criteria that N1314 did state in 2001 measured proposal quality, not need, and the paper conceded that real criteria did not yet exist. On what a proposal should contain, Matt Austern's N1314, dated 2001-05-17, says:

> *A proposal should be a well thought out design, and should include specific text that would be suitable for a standard. ... Ideally it should include a reference implementation.*

On the criteria themselves, N1314 concedes:

> *We will need to develop criteria for evaluating proposals.*

No later document ever supplied those criteria, as the SD-9 and Tokyo 2024 cards below show.

N1314 wrote Boost into the pipeline as an official proposal filter in 2001. The suggestion appears in Matt Austern's N1314, dated 2001-05-17:

> *We may also wish to ask BOOST to act as such a filter.*

Boost's own front page at boost.org states the reciprocal mission: "We aim to establish 'existing practice' and provide reference implementations so that Boost libraries are suitable for eventual standardization".

The TR1 document itself, in drafts from 2003 to 2005, states that its purpose is to manufacture existing practice. The purpose sentence appears unchanged in every draft, typo included; N1836, the Draft Technical Report on C++ Library Extensions (ISO/IEC DTR 19768) with Matt Austern as project editor, dated 2005, states it as follows:

> *The goal of this technical report it to build more widespread existing practice for an expanded C++ standard library.*

The identical text appears in N1540, the 2003 draft.

The TR2 call for proposals codified existing practice as the central criterion in 2005 and included an explicit escape hatch. The call, N1810, "Library Extension TR2 Call for Proposals", issued by Hinnant, Dawes, and Austern on 2005-04-29, states the criterion and its rationale:

> *The committee prefers proposals that are based on existing practice. ... First, a proposal must be implementable, and the best evidence that something is implementable is that it has been implemented. Second, if a proposal is based on existing practice, the committee can have more confidence that the proposal solves a real problem and that its interface serves the needs of real users.*

N1810 then adds the escape hatch:

> *In rare cases the committee might be willing to adopt a completely experimental library, even without widespread existing practice, if the proposal is sufficiently compelling.*

2012: the standing call for library proposals

The 2012 call for library proposals restated the regime of existing practice, no breakage, and no third-party dependencies, and it institutionalized a TR-first preference that later evaporated. The call is N3370, "Call for Library Proposals", written by LWG chair Alisdair Meredith and dated 2012-02-26:

> *The committee prefers proposals based on existing practice. ... Avoid breaking existing code. ... the clear preference is for new library components to go into TRs, while modifications go into the standard.*

Practice then diverged from the stated preference: std::format entered C++20 directly from {fmt} via P0645, std::expected entered C++23 with no TS via P0323, and the Networking TS, the component that best satisfied the TS-first rule, never merged, as section 7 documents.

2017-2019: the Direction Group and the Vasa

In P0939, published in 2018, the Direction Group records the founding criteria for library inclusion and admits that they had no effect. Section 4.2 of P0939R0, "Direction for ISO C++", written by Dawes, Hinnant, Stroustrup, Vandevoorde, and Wong and dated 2018-02-10, states:

> *During the early stages of the development of the first standard, we articulated some principle for what to include in the standard library, including Language support; Facilities that everybody needs; Facilities needed for communicating among separately developed libraries. We think these are reasonable criteria, but in practice they didn't have much effect. People worked on and overelaborated individual components (e.g., std::string) that didn't interoperate effectively ... We were rescued from disaster by Alex Stepanov.*

The passage is retained verbatim in section 5.3 of P2000R4.

P0939 also names the cost side of standardization, recording that no feature is cost free. Section 5 of the Direction Group's P0939R0, dated 2018-02-10, reads:

> *A significant part of the C++ community is panicked by fears of massive growth of features and complexity. ... No feature is cost free, there is always the implementation cost, the cost of producing teaching material, the time needed to learn, the opportunities for confusion, and the inevitable distraction from overselling.*

Bjarne Stroustrup's 2018 paper P0977, "Remember the Vasa!", named 43 in-flight proposals and reproduced the committee's own 1992 warning. The paper, P0977R0, dated 2018-03-06, states:

> *Many/most people in WG21 are working independently towards non-shared goals. Individually, many (most?) proposals make sense. Together they are insanity to the point of endangering the future of C++.*

P0977 then quotes Stroustrup's own 1992 committee posting:

> *If every extension that is reasonably well-defined, clean and general, and would make life easier for a couple of hundred or couple of thousand C++ programmers were accepted, the language would more than double in size. We do not think this would be an advantage to the C++ community.*

Two years later the Direction Group described the C++20 that shipped anyway as "a major step forward for C++" in section 6 of P2000R4.

2020-2026: criteria still pending

In P2000R4, published in 2022, twenty-one years after N1314 said criteria "will need to be developed", the Direction Group is still asking for the discussion to start. Section 5.3 of P2000R4, "Direction for ISO C++", written by the Direction Group and dated 2022-10-15, states:

> *We encourage a discussion of criteria of what should be in the standard library and what should not. The aim is to be able to discuss proposed new standard-library components in the context of articulated criteria.*

The committee's standing documents contain no inclusion criteria, and SD-9, the formal LEWG policy document, holds exactly one policy, which concerns [[nodiscard]] placement. As of its 2024-05-14 revision, the entire "List of Standard Library Policies" in SD-9, "Library Evolution Policies", which names Inbal Levi as its reply-to, reads:

> 1. Policy: Library wording should not use [[nodiscard]]

SD-8 covers compatibility rights only, and SD-10 contains language-side design principles. None of the three standing documents defines what belongs in the standard library.

The official minutes of the Tokyo 2024 meeting record LEWG's first-ever policies discussion, held 23 years after N1314 and framed as process efficiency rather than inclusion criteria. The minutes, N4980, "WG21 2024-03 Tokyo Minutes of Meeting", written by Nina Ranns in 2024, state:

> *LEWG had its first policies discussion during the Tokyo meeting. Policies were created to guide authors of standard library proposals, and by doing so, improve the process and save both the group and the authors' time.*

P3001, published in 2023, is the committee's most explicit modern criteria statement, and it appeared as an opposition paper whose citations are talks and blog posts, not any standing document. The paper is P3001R0, "std::hive and containers like it are not a good fit for the standard library", written by Muller, Laine, Lelbach, and Sankel in 2023, and its category list reads:

> *Elements of the standard library ideally fall into one of the following categories: 3.1 Types and functions requiring compiler intrinsics ... 3.2 Core vocabulary types ... 3.3 Cross-platform OS abstractions ... 3.4 Fundamental algorithms and data structures*

P3001 also states the cost frame:

> *Standardizing a feature takes a lot of work, and the committee has limited time. Everything we discuss takes time away from a different feature and means delaying something else.*

The room voted to continue with hive anyway at a 2:1 ratio, as the committee's GitHub paper tracker records.

The public vote record

At the Oxford meeting in 2003, TR1 admissions passed unanimously, and the Boost-sourced components faced zero recorded opposition. The minutes, N1459, "Minutes of J16 Meeting No. 36/WG21 Meeting No. 31", written by Robert Klarer in 2003, record the motion on the hash-table proposal:

> *Move to accept N1456, "A Proposal to Add Hash Tables to the Standard Library (revision 4)", for inclusion in the library extensions technical report. ... J16: favor 19, oppose 0, abstain 0. WG21: favor 9, oppose 0, abstain 0.*

At the Berlin meeting in 2006 the TR1 merge was voted by component, and the special math functions failed to carry WG21, though they were published as a separate standard and merged into C++17 anyway. The Berlin minutes, N1993, written by Robert Klarer in 2006, record the tally:

> *Straw poll on the adoption of TR1 special math functions into the C++0x working draft: J16: favor 11, oppose 7, abstain 6. WG21: favor 3, oppose 5, abstain 2.*

The functions were published as ISO/IEC 29124:2010 and merged in C++17 via P0226. Their 21-year implementation arc is priced in section 6.

The Kona meeting in 2007 produced the committee's only formal scope-exclusion vote in this record, and every excluded item later entered anyway. The Kona minutes, N2452, written by Robert Klarer in 2007, record the resolution:

> *WG21 resolves that for this revision of the C++ standard (aka "C++0x") the scope of concurrency extensions shall be constrained as follows: Include a memory model, atomic operations, threads, locks, condition variables, and asynchronous future values. Exclude thread pools, task launching, and reader-writer locks. J16: favor 25, oppose 1, abstain 1.*

Task launching later entered as std::async in C++11, and reader-writer locks entered as std::shared_timed_mutex in C++14 and as std::shared_mutex in C++17.

Statement-vs-practice divergence pairs

Pair 1 - std::regex satisfied every stated criterion of its era and is now the committee's own example of a stuck facility. On the statement side, regex entered TR1 from Boost.Regex under the existing-practice regime, and the proposal chain is recorded in N2364. On the admission side, twenty years later, the 2023 paper P3001R0 states:

> *Facilities that are bad but insufficiently terrible like std::vector, std::unordered_map, or std::regex are going to stick around. ... How many std::unordered_maps or std::regexes do we want in the standard library?*

Section 8 holds the full penalty record for regex.

Pair 2 - networking was declared urgent, planned first, polled out, and remains absent. The Direction Group's P0939R0, from 2018, called networking "urgently needed (on the C++20 timescale)". P0592R4, the 2019 paper adopted as the committee's C++23 plan, made executors and networking must-work-on-first items: "We have a TS, we have years of existing practice, so once the blocker item is resolved, we should standardize networking." The October 2021 LEWG polls recorded weak consensus against the Networking TS model, as P2453R0 documents, and the full poll table is in section 7. C++23 and the C++26 drafts contain no networking. The component that best satisfied the existing-practice criterion of 2001-2012 is the one the committee declined to admit.

Pair 3 - 2D graphics was recommended by the Direction Group and killed by the process in the same two years. The Direction Group's P0939R0 recommended "An (optional) Graphics API TS" as a C++20-cycle priority in February 2018. Apple's formal feedback paper from the same year, P1225R0, written by JF Bastien and dated 2018-10-02, stated: "we don't think P0267R8 meets the quality bar for acceptance into C++". The graphics proposal reached P0267R10 on 2019-06-17 and has had no revision since. The dossier with the poll inversion is in section 7.

Pair 4 - the removal policy was stated on the record in 2006 and was applied selectively in the years that followed. The statement is in the Berlin minutes of 2006, N1993:

> *Hinnant replied that the LWG has decided that, in such an event, the obsolete library will be removed from the Working Paper.*

In subsequent practice, auto_ptr, bind1st/bind2nd, and random_shuffle were removed in C++17 under N4190, written by Stephan T. Lavavej in 2014, while std::regex and std::unordered_map, the components P3001 names as "bad but insufficiently terrible", remain.

5. Coordination problems and reach

The vocabulary-type mechanism, defined by the committee's own officers

Titus Winters defines vocabulary types and names the ambient cost of having multiple common forms in a 2020 committee paper.

> *The types that are most commonly passed through interfaces in a given codebase are what we call "vocabulary types" ... For non-scalar types (strings, vectors, hashmaps), having multiple common forms also leads to an ambient performance cost as chains of function calls convert back and forth between different forms.*

Winters wrote this in P2125R0, "The Ecosystem Expense of Vocabulary Types", dated 2020-02-20. P2125R0 also cites the Java precedent of Guava Optional versus java.util.Optional, which it describes as "significant and unnecessary friction".

Bryce Adelstein Lelbach, the sitting LEWG chair at the time, states the interoperability rationale verbatim in a 2021 talk devoted to the question of what belongs in the standard library.

> *it's a pain for users if every library defines its own string type and they have to deal with all of them. That's why we have std::string_view in the C++ standard library, so that there's one common string abstraction that can appear in interfaces. Everyone can still have their own string types for their libraries if they want, but by making them convertible to std::string_view they can ensure that their string types will work with everyone else's code.*

The statement comes from the C++Now 2021 closing keynote "What Belongs In The C++ Standard Library?", which Adelstein Lelbach delivered in 2021 while serving as LEWG chair.

String fragmentation: the partially solved case

More than 200,000 public C++ files call the QString/std::string conversion shim, which quantifies the boundary cost of string fragmentation. A GitHub code search for the exact string "QString::fromStdString", restricted to language:C++, returns 202,752 matching files, each of them a site where code pays the boundary conversion between the Qt string ecosystem and the standard string ecosystem. The count comes from the GitHub code search API, queried on 2026-07-08.

The conversion is measurably expensive, with cost differences of up to 10x between QString conversion paths.

> *QString stores 16 bit UTF-16. toStdString reencodes this as utf-8 using an intermediate byte array ... The conversion from QStringView via QString to std::string is by far the slowest. Especially for large strings it is a complete order of magnitude (factor 10) slower than QString to std::string.*

The measurement comes from a Stack Overflow benchmark thread from 2023. Qt's own documentation instructs users to match Qt string types at API boundaries, and the Qt 6 string processing docs warn that "Encoding conversions are conducted implicitly in many cases but this should be avoided if possible". Framework vendors themselves route around their own string types: the wxWidgets 3.2 docs tell users to "convert them to and from wxString only when interacting with wxWidgets" and label the conversion "potentially destructive".

Vocabulary adoption: the mechanism working

Google pre-adopted the C++17 vocabulary types in Abseil and designed them to collapse into aliases of the std types.

> *in builds where, for example, std::optional exists, the Abseil version will merely be an alias ... there is only one type in play.*

The design is documented in the Abseil drop-in types design note from 2017, and the release was announced the same year on the Google Open Source Blog. Abseil's Tip of the Week #1 institutionalized string_view as the parameter type at API boundaries starting in 2012, five years before C++17 shipped.

ICU, a decades-old library with its own UTF-16 string class, added std::u16string_view acceptance at its boundary once the standard type existed, starting with ICU 76 in 2024.

> *Starting with ICU 76, a UnicodeString can be converted to/from a std::u16string_view, and several other member functions also work directly with std::u16string_view inputs.*

The change is documented in the ICU user guide, Strings chapter published by the Unicode-org ICU project.

Boost built a dedicated string_view whose entire purpose is interoperation with the standard vocabulary type.

> *Unlike boost::string_view, boost::core::string_view has implicit conversions from and to std::string_view, which allows Boost libraries that support C++11/C++14 to use it in interfaces without forcing users to forgo the use of std::string_view in their code.*

The boost/core rationale is quoted in boostorg/beast issue #2594, dated 2022-12, and parallel downstream demand for the same interoperation appears in boostorg/charconv #143 and boostorg/static_string #26.

Unsolved fragmentation: JSON and error handling

A WG21 JSON proposal, P0760 from 2018, exists but was never adopted, and its own motivation section documents the fragmentation.

> *There are numerous libraries written in C and C++, usable by C++, but none in the standard. In comparison, JSON is part of the respective standard libraries of languages like GO, Python, Swift, or D.*

The quote comes from the P0760R0 proposal text by Niels Lohmann and Mario Konrad, dated 2018-05-14. The proposal repository answers its own status questions with "Is it done? No. When is it done? We have no idea."

The parallel JSON type ecosystems are all large and mutually incompatible. A GitHub code search restricted to language:C++ and run on 2026-07-08 finds "nlohmann::json" in 684,032 files and "rapidjson::Document" in 72,288 files. Each library exposes its own incompatible value type and conversion protocol: nlohmann uses ADL to_json/from_json, Boost.JSON uses tag_invoke, and a user request for cross-library serialization was closed in nlohmann/json #949 in 2018 with the reply "the library is independent of Boost".

The committee's own 2021 poll record on std::expected states the error-type Coordination Problem in one sentence.

> *We are in dire need of standardized error-or-value type, but there is still a lot of competing types in the area, and discussion regarding the API.*

The comment is anonymous LEWG poll feedback recorded in P2384R1, "2021 Spring Library Evolution Poll Outcomes", from 2021. Boost.Outcome ships machinery "to make easier interoperation between multiple third party libraries each using incommensurate Outcome (or Expected) configurations", as its Boost.Outcome FAQ states, an explicit acknowledgment that the ecosystem fragmented while the standard type was pending.

Herb Sutter's P0709 states that divergent error handling fractured C++ into incompatible dialects.

> *[d]ivergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling*

The P0709 passage is quoted in P2232R0, a 2021 committee paper. SG14 documented the mechanism in a numbered paper, P0364R0 from 2016, which reports that "games programmer simply turn off exceptions, and this has created 2 sets of code divide".

The async coordination failure is stated in P2300's own motivation: the standard's future type was unusable as a basis, so every serious async library grew its own.

> *std::async/std::future/std::promise, C++11's intended exposure for asynchrony, is inefficient, hard to use correctly, and severely lacking in genericity, making it unusable in many contexts. ... These expenses rule out the future abstraction for many uses and makes it a poor choice for a basis of a generic mechanism.*

The passage appears in P2300R10, "std::execution", the proposal adopted for C++26. Facebook brought its parallel future ecosystem of folly::Future and folly::SemiFuture to WG21 as evidence in P0904R0 in 2018, and composing across future types required the explicit adapter code documented in P1678R2 in 2019.

Counter-adoption evidence: standardized vocabulary that failed at the boundary

char8_t, the standardized UTF-8 vocabulary type, generated ecosystem-wide friction and compiler opt-outs instead of adoption.

> *the direct incompatibility between char and char8_t was felt, enough that -fno-char8_t and /Zc:char8_t- needed to be rolled out the moment conforming C++20-aspiring implementations rolled out ... Popular user libraries such as Dear imgui, nlohmann::json, and many others suffer from these issues.*

The account comes from P2513R2, "char8_t Compatibility and Portability Fix", by Meneide and Honermann, adopted as a defect report against C++20 in 2022. A real project's migration post-mortem, recorded in transmission/transmission #8263 across 2024-2025, states: "Many library and OS APIs only accept and return char, most notably {fmt}. ... I don't think the benefit of cleaner interop is worth the cost".

std::string_view carries a known boundary defect that blocks adoption by C-API-wrapping functions.

> *std::stoi doesn't accept string_view. (Why? Because it's a thin wrapper around strtol, and strtol requires null-termination.)*

The observation comes from Arthur O'Dwyer, "string_view and strong typedefs", a blog post published on 2019-10-10.

The Reach Test: sizing the constituencies

The per-domain reach table below assembles the best available number for each domain from primary surveys, registries, and the GitHub API. Because the survey questions are multi-select, the percentages sum to more than 100%.

Domain	Best available number	Source	Year
Hash maps / containers (baseline)	100% by definition; constituency is the whole C++ population, ~10.3M active developers	SlashData State of the Developer Nation, 25th ed.	Q3 2023
Networking	19.37% to 25.38% of C++ devs work on Communications (networking, email) projects; indirect reach far larger (curl alone estimated 20B+ installations)	ISO C++ Developer Survey Lite Q4, 2021-2024; daniel.haxx.se	2021-2024
Embedded / freestanding	32.22% of C++ devs target embedded systems as a platform; 29.92% work on Hardware/IoT projects	ISO survey 2024 Q5 and Q4	2024
Filesystem	No direct survey number; platform proxy: 60.56% target Windows desktop, 57.30% Linux server, 56.75% Linux desktop	ISO survey 2024 Q5	2024
Formatting / logging	No direct survey number; package proxy: spdlog 29,054 stars and {fmt} 23,648 stars, both top-tier among all C++ libraries	GitHub API	2026-07-08
JSON / serialization	Package proxy: nlohmann/json 50,094 stars; protobuf 981,350 dependent GitHub repositories	GitHub	2026-07-08
Regex / text processing	Package proxy: RE2 9,722 stars; std::regex named in 2025 ISO survey write-ins as a broken component	GitHub; isocpp.org	2025-2026
2D graphics / GUI	Qt used by 7.3% of all Stack Overflow respondents (all languages); Qt Group claims a community of 1.5M developers; Dear ImGui 74,356 stars	SO Survey 2024; Qt Group reports	2024
Linear algebra / scientific	12.46% of C++ devs work on Machine learning and 11.59% on AI projects (overlapping multi-select); Eigen has 1,183 GitLab stars but powers TensorFlow, Ceres, ROS, Stan	ISO survey 2024 Q4; GitLab	2024-2026

For comparison, adjacent rows from the same ISO C++ Developer Survey show Gaming at 20.87% (down from 32.89% in 2021), Engineering (avionics, power management) at 31.51% (the largest 2024 category), Developer tools at 27.38%, and Financial at 9.44%.

The baseline population is roughly 10 million C++ developers. SlashData estimated 10.3M active C++ developers worldwide in Q3 2023, out of 38.9M developers total, and Stack Overflow puts C++ at 23.5% of all respondents in 2025. For Reach Test arithmetic, a domain claiming N% of C++ developers therefore sizes as N% of 10M, to order of magnitude.

Networking reach is verified: one in five to one in four C++ developers writes Communications software, and indirect reach is nearly universal. The ISO C++ Developer Survey Lite question Q4 records the share of C++ developers working on "Communications (e.g., networking, email)" projects at 25.09% in 2021, 25.38% in 2022, 23.56% in 2023, and 19.37% in 2024, where the category ranked 3rd of 16 project types. The indirect ceiling comes from curl: its maintainer estimates twenty billion installations (2024), a figure updated to thirty billion in the State of curl 2026 slides. A caution carries over from the reach research: a circulating claim that attributes networking-library percentages ("37% ENet, 45% Boost.Asio") to the Stack Overflow 2025 survey is fabricated, and no such question exists in the published SO 2025 results. The ISO survey numbers above are the ones to use.

Embedded reach covers roughly a third of the C++ population, and it is rising while other segments decline. The share of C++ developers naming embedded systems as a target platform rose from 29.41% in 2021 to 32.22% in 2024, and Hardware/IoT projects held near 30% throughout, according to the ISO survey Q4/Q5 questions. One caution applies here: the JetBrains "37% embedded" figure sometimes quoted is C++20 adoption within the embedded cohort, not cohort size, as the JetBrains 2023 C++ ecosystem report shows.

The number one pain point every measured year is dependency management, not missing standard library components. The answer "Managing libraries my application depends on" is the top-ranked major pain point in every ISO survey year, at 48.46% in 2021, 47.63% in 2022, 47.37% in 2023, and 45.43% in 2024, per the ISO survey summaries, with a year-over-year table available at moderncppdevops.com. The 2024 ISO survey also records how dependencies are actually managed: 68.54% of respondents compile libraries from source in their build, while vcpkg at 19.10% and Conan at 19.34% each reach about one in five.

Package-proxy star counts, taken from the GitHub API on 2026-07-08, size the flagship libraries across domains.

Library (domain)	Stars
OpenCV (vision)	89,736
Dear ImGui (GUI)	74,356
protobuf (serialization)	71,476
nlohmann/json (JSON)	50,094
gRPC (networking/RPC)	44,989
curl (networking)	42,342
OpenSSL (crypto)	30,432
spdlog (logging)	29,054
{fmt} (formatting)	23,648
Abseil (containers/utilities)	17,382
RE2 (regex)	9,722
Boost superproject	8,512
Asio (networking)	5,940
Eigen (linear algebra; GitLab)	1,183

Caveats apply to the star counts. The qtbase and Asio figures understate reach, because Qt develops on code.qt.io and because Asio is consumed mostly via Boost, which still moves ~2M SourceForge downloads per year; Eigen is on GitLab, where starring is much rarer. protobuf's 981,350 dependent repositories is the largest dependents count found for any C++-core library.

6. The Complexity Budget, priced

Page and wording growth

The C++ standard has more than tripled in size since C++98, and the library is where the growth is. The following table sets the published ISO page counts beside the page counts measured directly from the working-draft PDFs:

Version	Published ISO pages	Working draft (measured)	Draft pages
C++98	732 (iso.org)	-	-
C++03	757 (iso.org)	-	-
C++11	1338 (Stroustrup HOPL-IV)	N3337	1324
C++14	1358 (same)	N4140	1365
C++17	1605 (same)	N4659	1622
C++20	-	N4861	1834
C++23	-	N4950	2134
C++26 WD	-	N5046	2679

The C++26 working draft is 545 pages (26%) larger than the C++23 final draft, the largest single-cycle jump in absolute pages in the language's history. The draft page counts were measured for this document programmatically from the cited PDFs on 2026-07-08, and the method is recorded in the 2026-07-08 complexity budget research.

The library clauses grew 140% since C++11 while the language clauses grew 36%, and the library is now 77% of the clause text. The following page spans were read from the PDF bookmark trees of the same working drafts:

Draft	Language pages	Library pages	Library share
N3337 (C++11)	397	763	66%
N4659 (C++17)	446	965	68%
N4861 (C++20)	457	1152	72%
N4950 (C++23)	477	1403	75%
N5046 (C++26 WD)	538	1833	77%

Bjarne Stroustrup corroborates the proportion in print, writing in HOPL-IV, 2020 that "The standard library is about 3/4 of the C++20 standard".

The library name index quadrupled, from 3,543 entries in C++11 to 14,278 in the C++26 working draft. The entries were counted uniformly for this document on 2026-07-08 from the rendered official LaTeX sources: C++11: 3,543, C++14: 3,609, C++17: 5,860, C++20: 8,275, C++23: 9,568, and C++26 WD: 14,278. The bare "Index of library names" section now occupies 122 pages of the C++26 draft, up from 36 pages in C++11. The header count grew from the 54 C++ headers listed in N2800 in 2008 to the 97 C++ headers plus 29 C headers listed in the C++26 header index.

The wording cost of individual library components can be measured from the page spans of their clauses. The following table records the measured spans and the draft each span was measured in:

Component	Version	Pages	Draft
Ranges library	C++20	64	N4861
Ranges library	C++23	151	N4950
Ranges library	C++26 WD	170	N5046
Formatting	C++26 WD	29	N5046
File systems	C++17	54	N4659
Regular expressions	C++17	45	N4659
Execution control (std::execution)	C++26 WD	91	N5046

The Ranges clause grew 87 pages between C++20 and C++23. The new Execution control clause enters at 91 pages, larger than the entire File systems or Regular expressions components, with zero shipping mainstream implementations, as section 8 documents.

A single merge paper can approach the scale of the original library. P0896R4, "The One Ranges Proposal" of 2018, is a 226-page wording document; the entire C++98 library specification was on the order of 350 pages of a 732-page standard.

Proposal authors state their name spend in their papers and worry about it on the record. The authors of P1673R13, the std::linalg proposal (2023), write: "Our proposal would add 61 new unique names to the C++ Standard Library." The author of P3287R3, a 2025 paper on simd namespaces, writes "I agree with the concern and I feel uneasy with the need for simd to grab as many names as it would need to", and lists the existing sub-namespace inventory (chrono, execution, filesystem, linalg, numbers, pmr, ranges, views) as precedent.

Vendor divergence and adoption latency

The vendor divergence matrix shows that a standardized feature is not the same as an available feature. The following matrix records when, if ever, each of the three mainstream standard library implementations shipped each standardized feature:

Feature	Standardized	libstdc++	libc++	MSVC STL
GC interface	C++11	never	never	never
Special math	C++17 (TR1 2005)	GCC 6.1 (2016)	not implemented (20 of 21 "Not Started", LLVM 19)	complete only VS 2022 17.3 (2022)
Parallel algorithms	C++17	GCC 9 via TBB	still tracked incomplete (LLVM 21.1, 2025)	VS 2017 15.7+
Ranges	C++20	GCC 10 (2020)	complete ~Clang 15 (2022)	VS 2019 16.10 (2021)
std::format	C++20	GCC 13 (Apr 2023)	complete Clang 17 (2023)	VS 2019 16.10 (2021)
Modules	C++20	partial	partial	partial; build systems Dec 2023+
std::generator	C++23	GCC 14 (2024)	not implemented	19.43 (2025)
std::execution	C++26 WD	none	none	none

The matrix is compiled from the cppreference compiler support table, the libc++ C++23 status page, the libc++ C++26 status page, and the libstdc++ status tables, all consulted on 2026-07-08.

The special math functions have traveled a 21-year arc from TR1 to the present, and one table traces it.

Date	Event
2005	23 special math functions in TR1 (N1836)
2010	Published separately as ISO/IEC 29124:2010 (N3060) after failing the Berlin vote (section 4)
2011	Left out of C++11 "out of concern for implementability" (libstdc++ docs)
2016	libstdc++ implements (GCC 6.1)
2017	Merged into C++17 via P0226
2022	MSVC completes overloads only in VS 2022 17.3 (cppreference)
2024+	libc++ (LLVM 19): 20 of 21 functions "Not Started"

The arc ends with a mandated C++17 feature unavailable on the default Apple/Android toolchain nine years after standardization.

The garbage collection interface was standardized in 2011, implemented by nobody, and removed in 2023. In P2186R2, "Removing Garbage Collection Support", the 2021 removal paper by Bastien and Meredith that was adopted for C++23, the authors write:

> *libc++, libstdc++, and Microsoft's Standard Library all offer relaxed pointer safety and not strict pointer safety ... This status-quo hasn't changed in 12 years. ... the current specification simply missed the mark, and will not be missed.*

Users wait one to six-plus years after publication for features to become usable, and sometimes they wait forever. The following table pairs each feature's final draft date with its first complete mainstream implementation and the resulting latency:

Feature	Final draft	First complete mainstream implementation	Latency
std::format	Apr 2020	MSVC May 2021; GCC 13 Apr 2023; libc++ 17 (2023)	1-3+ years
Modules (usable with a build system)	Apr 2020	CMake 3.28 + MSVC 17.4/Clang 16/GCC 14, Dec 2023 onward	3.5-6 years
std::generator	May 2023	GCC 14 (May 2024); MSVC 19.43 (2025); libc++ none	1 year to never-yet
Special math	Dec 2017	libc++ none	9+ years and counting

libc++ shipped std::format "implemented but still marked as an incomplete feature" in Clang 14, then behind -fexperimental-library in Clang 15, and unconditionally only from Clang 17, because "Not yet implemented LWG-issues will cause API and ABI breakage"; the staged rollout is documented in the release-notes trail. The libc++ front page still reads "C++17 - In Progress" in 2026, nine years after publication.

Committee throughput: the paper flood against fixed capacity

One pre-meeting mailing now approaches the entire C++98 paper corpus. Herb Sutter wrote in his pre-trip report and trip report for the San Diego meeting of 2018-11: "Record #papers: 274 in pre-San Diego. ... By paper count, it exceeded the previous pre-meeting papers record by about 65%, and started to approach the total number of technical papers to produce the first C++ standard (total of pre-meeting mailings from 1990-1997)". For scale, the entire 8-year C++98 cycle produced 1,148 papers.

The yearly paper counts, tallied on 2026-07-08 from the public open-std.org indexes, put the flood in numbers.

Year	Revision files	Unique papers
2015	179	156
2018	852	571
2019	937	617
2021	538	303
2023	740	432
2024	957	560
2025	828	517

The years 2019 and 2024 each roughly match, in one year, the paper output of the entire 8-year C++98 cycle. The counts were scraped for this document from the public open-std.org paper indexes, and the method is recorded in the 2026-07-08 complexity budget research. Meeting scale grew in parallel: Herb Sutter's trip report from Prague 2020 records a record 252 attendees, 23 active subgroups, and nine parallel tracks.

The Direction Group states in its own words that the flood of proposals exceeds the committee's processing capacity. In its 2020 paper P2000R2, "Directions for ISO C++", the Direction Group writes:

> *One effect of this enthusiasm for C++ is to inundate us with a flood of proposals. The sheer volume of proposals leads to fewer proposals getting through the processes to get accepted. Some proposals become warped from the need to gain support in an environment where time to think and present ideas is limited. ... the sheer number of proposals (constantly on the order of a hundred) and the number of papers (running at more than one hundred for each meeting), implies that not even every good proposal can be accepted.*

LEWG's published throughput accounting shows 74 telecons and 107 papers reviewed in ~17 months, against a standing backlog of 41 active papers. The accounting appears in P2400R2, the Library Evolution report by Adelstein Lelbach et al. dated 2021-09-28, which describes the review cadence:

> *Since 2020-04, Library Evolution has held a 90 minute telecon twice a week to review papers. Typically, 1 or 2 papers are reviewed at each telecon.*

The same report lists Executors, Networking, and Coroutines library support as all flagged "At risk for C++23."

Of the 15 published library-relevant Technical Specifications, roughly half delivered nothing to the standard. The following table records each Technical Specification, its publication year, and its fate:

TS	Published	Fate
Filesystem TS (18822:2015)	2015	Merged into C++17
Parallelism TS 1 (19570:2015)	2015	Merged into C++17
Transactional Memory TS (19841:2015)	2015	Never merged; line ended as an SC22 White Paper in 2026 after 14 years of SG5 work
Library Fundamentals v1 (19568:2015)	2015	Merged into C++17 except invocation traits
Concepts TS (19217:2015)	2015	Failed C++17 merge; merged into C++20 with terse syntax and function concepts stripped at Toronto 2017
Concurrency TS 1 (19571:2016)	2016	Partially merged into C++20; future::then never merged
Library Fundamentals v2 (19568:2017)	2017	Partially merged; observer_ptr never merged despite a Stroustrup abandonment paper
Ranges TS (21425:2017)	2017	Merged into C++20
Coroutines TS (22277:2017)	2017	Merged into C++20
Networking TS (19216:2018)	2018	Never merged (dossier in section 7)
Modules TS (21544:2018)	2018	Merged into C++20
Parallelism TS 2 (19570:2018)	2018	simd merged into C++26 six years later via P1928; task_block, for_loop, vector policies dead
Reflection TS (23619:2021)	2021	Zero words merged; design abandoned for value-based P2996
Concurrency TS 2 (9922:2024)	2024	Published after its headline components (hazard pointers, RCU) had already entered the C++26 draft directly in 2023
Library Fundamentals v3 (19568:2024)	2024	Nothing merged

The registry behind the table is the cppreference experimental TS status table, cross-checked against the open-std final drafts. Two more TS projects (Array Extensions, Transactional Memory 2) and the Numerics TR successor died before publication.

The executors effort generated 90 distinct papers and 165 revision documents from 2012 through 2024, and 115 papers and 219 revisions through mid-2026. The papers were counted from the public wg21.link index by title match on executor/sender/receiver/scheduler/std::execution, and the count is a floor, because papers that discuss executors without a matching title word are excluded. The lineage runs from N3378 in 2012, through the 14 revisions of P0443, which SG1 adopted "by universal consensus" at Cologne 2019 and then retired unadopted, to the 11 revisions of P2300. The first merge of any executor-lineage work into an IS came in 2024, twelve years after the first paper. Along the way, the saga produced six "compromise" papers (P0688, P1055, P1194, P1658, P1660, P1820), six parallel LEWG breakout review reports in one summer (P2202 through P2207), and a standalone paper written solely to document the history of one contested sub-mechanism (P2033).

The Interaction Tax, visible in the wording

A paper exists solely to let new iterators work with old algorithms. P2408R5, "Ranges iterators as inputs to non-ranges algorithms", a C++23 paper, bridges the C++20 iterator concept taxonomy back to the C++98 iterator requirement tables. The paper adds no new functionality; it is pure compatibility wording between two generations of the same library.

The library's own organization confuses the committee that maintains it. The committee paper P1453R0, from 2019, records the confusion:

> *Most sequence algorithms are found in , but some are found in and . This has even been confusing to the committee, which has often forgot to apply changes applicable to all algorithms to the ones in and .*

Freestanding papers must carve exclusion lists because components entangle transitively: Ben Craig's P1642R11, from 2022, documents that a C++20 ranges feature (istream_view) is unusable on freestanding because of the 1990s iostreams design.

7. Case dossiers

Every dossier in this section records facts only: provenance, dates, votes, implementation status, and ecosystem alternatives. Verdicts belong to the model, not to this section.

string / string_view

string_view arrived field-proven, consolidated from four independent production implementations of the same string-reference type. The proposing paper, N3762 (Jeffrey Yasskin, 2013), states: "Google, LLVM, and Bloomberg have independently implemented a string-reference type to encapsulate this kind of argument. ... This paper follows Chromium and Bloomberg in extending string_view to basic_string_view." The naming and semantics were harvested from the deployed types, and N3685 (2013) records the choice: "Existing practice doesn't agree on a name for this operation, so I've kept the name used by Google's StringPiece". Google StringPiece and LLVM StringRef had 5-10+ years of production use at proposal time.

string_view reached the standard through the Library Fundamentals TS and then C++17, and the papers made the vocabulary rationale explicit. P0254R2 (Marshall Clow, 2016) records the path: "In the Library Fundamentals Technical Specification (LFTS), we introduced std::experimental::string_view ... and it has proven to be very popular. In Jacksonville, it was approved for C++17". The same paper names the third-party string zoo of QString, CString, and "innumerable home-grown versions" as the reason string_view is a vocabulary type. That vocabulary framing became the stock rationale for the later interface repairs P2495R3 and P2697R1 of 2022-2023. Adoption evidence at the ecosystem boundary, covering ICU, Boost, and Qt, is collected in section 5.

Networking TS

The Networking TS descends from Asio, which Chris Kohlhoff announced publicly in September 2004 with commercial production deployment already scheduled, and which Boost accepted in December 2005 with review testimony naming five production systems. Kohlhoff's announcement on the Boost mailing list, 2004-09-13 stated: "it will be going into production in a commercial system in a little over a month ... a custom HTTP proxy that supports hundreds of concurrent connections". Testimony posted during the Boost review on the Boost list, 2005-12-12 enumerates five commercial production systems live since November 2004.

The Networking TS's standards timeline runs from first proposals in 2005-2006 through an ISO project in 2014, a week-long special LWG review in 2015, and TS publication in 2018 to a reversal of direction in 2021 that removed it from the program, with nothing shipped in its place by 2026. The first proposal was N1925 (2005), and Kohlhoff's TR2 proposal N2054 (2006) already claimed named production systems. The ISO record for TS 19216:2018 shows the project approved 2014-06-30 and the TS published 2018-04-24. P1446R0 (Wakely, 2019) stated the direction at the time: "Incorporating networking is a priority of the Direction Group ... Users are clamouring for its addition." The reversal came with P2464R0 (Voutilainen, 2021-09-29), which stated: "we should not standardize the Networking TS as it's currently designed". The authors' rebuttal P2469R0 (Kohlhoff, Allsop, Falco, Hodges, Morgenstern, 2021-10-04) answered: "The Networking TS is now six-years-old and would appear to possibly become exactly such a casualty of this ever increasing and unhealthy trend towards standardising unchallengeable ideas over practice." The 2026 retrospectives state the outcome plainly: P4096R0 records that "It was removed from the committee's work program in 2021. In 2026, no replacement has shipped", and P4099R1 carries the timeline.

The decision votes on the Networking TS were taken as LEWG/SG1 electronic polls in October 2021 and were published as a numbered committee paper.

Poll	SF	WF	N	WA	SA	Outcome
Networking TS/Asio model is a good basis for most async use cases	5	10	6	14	18	Weak consensus against
Sender/receiver model (P2300) is a good basis for most async use cases	24	16	3	6	3	Consensus in favor
Stop pursuing the Networking TS/Asio design for networking	13	13	8	6	10	No consensus
Networking should be based on the sender/receiver model	17	11	10	4	6	Weak consensus in favor
Acceptable to ship sockets without TLS/DTLS	9	13	5	6	13	No consensus

These tallies were published in P2453R0, "2021 October Library Evolution and Concurrency Networking and Executors Poll Outcomes", by Adelstein Lelbach, Fracassi, and Craig, dated 2022-02-15. The LEWG vice chair announced the result publicly on r/cpp, October 2021: "LEWG no longer has consensus to standardize the Networking TS. ... The chairs know that the NetTS and Asio represent decades of work". As of late 2025, a Stack Overflow status answer records that SG4 decided the TS "was no longer relevant and should be dropped" and that the sender-based replacement P3482 has had no update since January 2025.

Linear algebra (std::linalg, P1673)

std::linalg (P1673) derives from the BLAS interface, which was standardized 1973-1990 entirely outside any language standard, and from the Kokkos reference implementation. The BLAS lineage runs from a 1973 JPL memorandum through the 1979 ACM TOMS Level 1 paper to LAPACK v1.0, which was built on Level 3 BLAS in 1992, as documented at netlib. P1673R13 (Hoemmen et al.) states: "The set of linear algebra operations in this proposal are derived from a well-established, standard set of algorithms that has changed very little in decades. It is one of the strongest possible examples of standardizing existing practice that anyone could bring to C++." The proposal was adopted into the C++26 working draft at Kona in November 2023, after 13 revisions spanning 2019-2023.

Lead author Mark Hoemmen describes std::linalg as an implementation layer, not a user-facing vocabulary. Writing on r/cpp, 2023 and again in 2024, Hoemmen framed it this way:

> *It's an implementation layer for a product like Eigen, just like the BLAS is an implementation layer for a product like LAPACK. ... std::linalg a.k.a. "std::blas" does not seek to be anything like Eigen. The syntax, goals, and abstraction level all differ.*

The ecosystem alternatives to std::linalg are mature, dominant, and downloadable. Eigen's user list includes TensorFlow, MuJoCo, Ceres, ROS, Stan, OpenCV-adjacent stacks, and the ATLAS experiment at CERN, and Google Scholar shows ~2,765 citations for a library with no academic paper. P1673 itself enumerates the vendor BLAS layers: AMD BLIS, ARM Performance Libraries, Cray LibSci, Intel MKL, IBM ESSL, and NVIDIA cuBLAS. A practitioner on r/cpp, March 2025 stated the GitHub Test in one sentence: "Eigen is header-only so anyone can use it in 2 minutes including download time". The post-adoption defect record and implementation status for std::linalg are in section 8.

regex

The regex library entered the standard from John Maddock's Boost.Regex, passing through TR1 in 2005 into C++11, and it was formally field-proven under the criteria of its era. The proposal chain is recorded in N2364; the TR1 admission votes, tallied in section 4, were unanimous; and `<regex>` shipped in C++11, as cppreference documents.

libstdc++ shipped a non-functional partial `<regex>` for three years, and a working implementation arrived only in GCC 4.9 in April 2014. libstdc++ maintainer Jonathan Wakely explained the situation on Stack Overflow: "It wasn't such a bad idea a few years ago ... No-one thought it would remain unusable for so long ... There are exported symbols from the libstdc++.so library that depend on the regex code, so simply removing it (in, say, GCC 4.8) would not have been trivial". The GCC 4.9 release notes record the arrival of the working implementation. The ecosystem alternatives are RE2, PCRE2, CTRE, and Boost.Regex itself, and the benchmark deltas against them are in section 8.

filesystem

The filesystem library came from Boost.Filesystem, which had been in wide field use since 2002 with Adobe Reader named as a deployment, and the proposal claimed four years of user-driven design correction. The proposing papers N1889 and N1975 (Beman Dawes, 2005-2006) state: "The proposal is based on the Boost Filesystem Library, which has been in use since 2002 and by now is in very wide use. For example, current versions of Adobe Systems products such as Adobe Reader use the Boost Filesystem Library."

The filesystem library reached the standard through the Filesystem TS of 2015 and was merged into C++17 with twelve open LWG issues pending at the adoption vote. P0218R0 (Dawes, 2015) recorded the state at the vote: "There are twelve open Filesystem issues in the LWG Active Issues list. ... The pending issues and proposals described below should be resolved by LWG in the next meeting or two, and do not block adopting the Filesystem TS for C++17." The TS-to-standard transition itself changed semantics that shipped code depended on: operator/ with absolute paths, stem() of dotfiles, absolute() removed and renamed, and operator<< quoting all changed, per the national-body resolutions in P0492R2 and the comparison table in this Stack Overflow answer by T.C.. The defect and security record is in section 8, and Boost.Filesystem's continued evolution after the freeze is documented there as well.

unordered_map

unordered_map was committee-adapted from Matt Austern's 2003 proposal, written when open addressing "was not regarded as sufficiently mature", and four of its API features leak the chaining assumption. N1456 (Austern, 2003) is the standardization basis. Joaquin M Lopez Munoz, the Boost.Unordered maintainer, published the analysis in ACCU Overload 170 (2022): lax hash requirements, user-controlled load factor, pointer stability, and a bucket API together mean "all standard library implementations use some form of closed addressing". The Boost.Unordered rationale states the implementation consequence: "By specifying an interface for accessing the buckets of the container the standard pretty much requires that the hash table uses closed addressing ... losing the efficiency and most of the memory gain, the main advantages of open addressing". The ecosystem alternatives are absl::flat_hash_map, folly F14, boost::unordered_flat_map, ska::flat_hash_map, and ankerl::unordered_dense. The benchmark deltas and the ABI lock-in record are in section 8.

format

std::format came from the {fmt} library, which shipped publicly in December 2012, roughly 6.5 years before std::format's adoption at Cologne in July 2019. The fmtlib/fmt repository was created 2012-12-07 and stands at 23,648 stars, 440 contributors, and 58 releases as of 2026-07-08. The proposal P0645 states: "A full implementation of this proposal is available at https://github.com/fmtlib/fmt/tree/std." Victor Zverovich, the library's author, wrote at vitaut.net, 2019: "the Text Formatting proposal (std::format) made it into the C++20 Committee Draft at the Cologne meeting ... This concludes a three-year long design, implementation, and standardization effort". The availability lag and the post-C++20 correction stack are in section 8.

std::execution (P2300)

std::execution (P2300) is a committee-designed unification of three separately deployed executor models, and the unified design itself was never deployed. P0443R0 (2016) describes the merge: "our programming model unifies three separate executor design tracks ... Google's executor model for interfacing with thread pools, Chris Kohlhoff's executor model for the Networking TS, and NVIDIA's executor model for the Parallelism TS." The 2026 retrospective P4094R0 describes the sequel: the executor work was "unified into P0443R0, which went through fourteen revisions, was never deployed as unified, and was replaced by P2300R10". P2300R7 states the replacement plainly: "we are not asking to update the existing paper, we are asking to retire it in favor of this paper."

std::execution was adopted at the St. Louis plenary in June 2024 with one third voting against. Herb Sutter's trip report from St. Louis, June 2024 records the merge of P2300R10 into the C++26 draft. Jonathan Müller's trip report for think-cell, published in July 2024, records the margin and the conditions of the vote: "it was a very narrow vote with 1/3 voting against adoption", and "committee members were not able to fully understand the intricate design details, and instead just trusted the authors" (Jonathan Müller, think-cell trip report). The St. Louis minutes record a national body asking to postpone the vote over "teachability and user story for beginners."

Zero mainstream standard libraries ship std::execution, and the reference implementation describes itself as experimental while carrying 152 open issues. The cppreference compiler support page shows no supporting version for

libstdc++, libc++, or MSVC STL as of 2026-07-08. The reference implementation, NVIDIA/stdexec, states: "stdexec is experimental and tracks an evolving standard. APIs may change without notice." The post-adoption correction record is in section 8.

hive

std::hive began as plf::colony, author Matthew Bentley's own game-engine component, had ~1.5 years of community feedback at first proposal, and took 29 published revisions over 8.4 years to reach plenary approval. P0447R0 (Matthew Bentley, 2016-10-16) states: "Colony has at this point been receiving community feedback for the past 1.5 years from Boost, SG14 and individual developers." Bentley described the origin in a Game Developer article: "Originally I developed it for my own game engine". The container was plenary-approved for C++26 at Hagenberg, the committee tracker closed 2025-03-15, and LWG's final telecon poll, held 2024-12-06, was 11-0-0. Field reports were added late: the user experience appendix first appeared at revision 17, five years in, as the P0447R26 revision history shows.

The committee renamed the field-proven colony container to hive and continued over a formal opposition paper at 2:1. P2332R0 records the rename: "The original name, colony, while seen as non-problematic by the author and users, was seen as problematic by many members of LEWG." The opposition paper P3001R0, written by four LEWG-leadership authors, argued that "evolutionary limitations and high standardization costs make standardization of libraries such as std::hive undesirable", and the room continued 2:1, as the committee tracker records. Committee cost continued after adoption: the constexpr-hive NB comment and paper cycle ran through 2026 and retargeted C++29, a sequence documented in P3933R1.

Graph library (P1709 / P3126-P3131)

The graph library proposal's own usage-experience section states that there are no users other than the authors, seven years into the effort. P3126R3 (Ratzloff, Lumsdaine, 2025) states: "There is no current use of the library outside of the proposers. ... There is no current deployment experience of the library." The planned first use has slipped year by year: P1709R4 (2023) said "next year", P3126R2 said "in 2024", and P3126R3 says "in 2025".

The graph library's reference implementation has been rewritten twice and self-describes as alpha, and the design direction changed fundamentally at least twice under review. The graph-v2 README states: "This repository is no longer maintained. Please use the successor project graph-v3 instead" and "This library is in the alpha stage ... not recommended for general use." P1709 was retired and split into seven papers, P3126-P3131 plus P3337, which now target C++29, as recorded at stdgraph/P1709 and in the committee tracker.

2D graphics (P0267 / io2d)

The io2d library behind the 2D graphics proposal was written for the proposal itself, modeled on cairo rather than adopted from a shipped field library, and the effort ran from N3888 in January 2014 to P0267R10 in October 2019. P0267R10 states: "The design has undergone numerous reviews and revisions since ... the initial proposal, N3888, in January 2014." No P0267 revision claims pre-existing production deployments of io2d itself: P0267R8 makes no such claim, and P2005R0 describes the library as "largely based on Cairo".

The 2D graphics votes inverted between 2017 and 2018, and co-author Guy Davidson documented the committee-time price. At Kona in 2017, the LEWG poll to advance toward a TS ran 1|8|3|1|1 in favor; by Jacksonville in 2018, P0988R0 (Guy Davidson, 2018) recorded: "Should the paper be advanced to LWG? 1|3|3|3|2. The effort was well and truly mired." The same paper states the cost of review: "The paper is large, at over 140 pages. The opportunity cost this represents to LWG is significant: it would comfortably swallow two committee meetings to verify the wording". The death mechanism involved no formal rejection: SG16 deferred the paper at Belfast 2019 for time, no revision was ever published again, and a December 2019 review in P2005R0 concluded "Fixing P0267 as-is is equivalent to redoing the entire thing", with the paper trail in the committee tracker. The record holds a counter-fact as well: LEWG repeatedly voted to keep seeing the paper, and P0267R10 records every continuation poll passing except Rapperswil 2018.

std::async

std::async was designed in committee in 2009 under the Kona-compromise constraints, was not derived from any shipped library, and was voted in less than two years before C++11 was finalized. N2889 (Crowl, 2009) describes the setting: "There have been repeated requests for a simpler mechanism, all of which were rejected by the committee as not being within the spirit of the Kona compromise. However, there are now national body comments requesting such a facility." The accepted design was N2996, by Sutter and Crowl, merged from competing drafts and voted in October 2009, as Anthony Williams's contemporaneous record documents.

Two years after std::async shipped, the US national body formally requested its deprecation, the feature's own co-designer drafted the deprecation wording, and the deprecation died. US comment 26 on the C++14 CD, recorded in N3733, reads: "Deprecate std::async due to the inability to reconcile the blocking semantics of the destructor of the returned values with the growing expected semantics of std::future's destruction". N3777 (Herb Sutter, 2013-09-23) provides the deprecation wording, and N3780 (Josuttis) killed it with a direct plea: "I strongly urge you to vote against deprecating async()." In 2018, P1044R0 summarized the standing assessment: "Many regard async() as fundamentally broken." The executor replacement effort that std::async's defects necessitated ran from 2012 to the 2024 adoption of std::execution, as section 6 records.

ranges

The ranges library came from range-v3, which had been public since 2013 with ~5 years of field use at merge time, and the merge paper carried usage statistics. P0896R3 (Niebler, Carter, Di Bella, 2018) states: "There is no part of the Ranges TS - concepts included - that has not seen extensive use via range-v3 ... A GitHub search, restricted to C++ files, for the string 'range/v3' ... turns up over 7,000 hits." The design merged into the C++20 draft via P0896R4 at San Diego in November 2018.

The ranges merge paper candidly recorded that no vendor had shipped the Ranges TS. Revision 2 of the merge paper, P0896R2 (2018), stated it directly:

> *As of the time of writing (February 1, 2018), no major Standard Library vendor has shipped an implementation of the Ranges TS. ... Arguably, we have not yet gotten the usage experience for which all Technical Specifications are intended.*

After adoption, views::split was "so unergonomic for its most common intended use-case" that it was renamed and redesigned as a defect report against published C++20 in P2210R2 (Revzin, 2021), and Microsoft STL implemented it as a DR. A plan paper, P2214R0, had to schedule the deferred view backlog for C++23, and const-iteration semantics were still generating committee papers in 2025, P3431R0 among them.

mdspan

mdspan came from the Kokkos View abstraction at Sandia National Laboratories, which was in production HPC use before and during the proposal, and the paper ran through 19 published revisions over almost seven years. The authors' peer-reviewed record, an SC19 workshop paper (Trott et al., 2019), states: "the Kokkos team initiated a collaboration ... The result of this over five-year process is std::mdspan ... as well as benchmarks of the production-quality reference implementation". P0009r0 appeared ahead of the Kona meeting in October 2015, and P0009r18 was approved by LWG telecon on 2022-06-24 with an 18-0-0 poll and entered C++23.

Even with the strongest provenance in the modern record, LWG review time forced submdspan, a function "considered critical" to mdspan, to be amputated to make the C++23 train. P2630R1, "Submdspan" (Trott, Hoemmen, Lebrun-Grandie, 2022), records the removal:

> *This function was considered critical for the overall functionality of mdspan. However, due to review time constraints it was removed from P0009 in order for mdspan to be included in C++23.*

The same paper records that the separation poll was "made purely out of scheduling concerns to make sure that LWG is not overloaded."

Summary provenance table

The following table condenses the provenance facts for every dossier in this section.

Component	Origin library	Years in field at proposal/adoption	Field reports in proposing paper	Outcome
string_view (N3762)	Google StringPiece, LLVM StringRef, Bloomberg, Chromium	5-10+ yrs of independent production use	Yes (comparative analysis of deployed types)	C++17
Networking TS (N2054)	Asio (2003), Boost.Asio (2005)	15 yrs at TS publication; 23 yrs total	Yes (named production systems since 2006)	TS 2018; never merged
linalg (P1673)	BLAS (1973-1990) + Kokkos stdBLAS	Interface decades old; C++ binding new	Partial (BLAS history; C++ interface new)	C++26 draft (Nov 2023); no vendor implementation yet
regex (via TR1)	Boost.Regex	~7 yrs at TR1	Yes (Boost deployment)	C++11; ABI-frozen, see section 8
filesystem (N1889)	Boost.Filesystem (2002)	~4 yrs at proposal; ~14 at C++17	Yes ("very wide use"; Adobe Reader named)	TS 2015; C++17
unordered_map (N1456)	Committee-adapted from prior hash-table practice	Design fixed 2003, pre-open-addressing maturity	No	TR1 2005; C++11; layout frozen
format (P0645)	{fmt} (2012)	~6.5 yrs at adoption (2019)	Yes (full implementation linked)	C++20
std::execution (P2300)	Committee unification (P0443 lineage); libunifex/stdexec prototypes	0 as adopted design	No (prototype experience only)	C++26 draft (St. Louis, June 2024); no vendor implementation
hive (P0447)	plf::colony (author's own, 2015)	~1.5 yrs at R0; ~10 at adoption	Not at R0; appendix added 2021 (R17)	C++26 (Feb 2025) after 29 revisions

Component	Origin library	Years in field at proposal/adoption	Field reports in proposing paper	Outcome
graph (P3126)	NWGraph (academic) + purpose-built graph-v2/v3	0 (written for the proposal; rewritten twice)	No ("no current use … outside of the proposers")	In flight 7+ yrs; targeting C++29
2D graphics (P0267)	io2d, written for the proposal (cairo-modeled)	0 as an independent field library	No	Stalled after Belfast 2019
async (N2996 lineage)	None; committee design (2009)	0	No	C++11; NB deprecation request 2013; replacement effort ran to C++26
ranges (P0896)	range-v3 (2013)	~5 yrs at merge (2018)	Yes (usage stats, test-porting evidence)	C++20; post-adoption redesigns via DR
mdspan (P0009)	Kokkos View (Sandia, production HPC)	Years of production before and during	Yes (benchmarks, SC19 paper)	C++23 (LWG 18-0-0); submdspan deferred

8. The Standardization Penalty record

The ossification mechanism: the Prague 2020 ABI record

A 2023 WG21 paper states the Standardization Penalty in the committee's own words: once a library facility is standardized it is essentially done, and its API and ABI are effectively frozen.

In P3001R0, a 2023 paper by Muller, Laine, Lelbach, and Sankel, the guidance to proposal authors reads:

> *Proposal authors need to be aware that as soon as something is standardized, it is essentially done. The committee has decided against a "standard library 2.0", so whatever facility was standardized, we have to live with it. … Once standardized, a library's API and ABI is effectively frozen, unlike non-standard libraries which can continue to evolve.*

At the Prague meeting in February 2020, WG21 polled the room on breaking ABI; the tallies were published on the public SG15 list, and the outcome is recorded in the official minutes.

Poll	SF	F	N	A	SA
We should consider a big ABI break for C++23	17	44	15	31	20
We should consider a big ABI break for C++SOMETHING	39	41	14	23	14
We should consider incremental ABI for every C++ release	98	35	6	0	2
When performance and ABI conflict, prioritize performance	41	26	39	13	13
We should promise users we won't break ABI, ever	3	3	12	43	70

The tallies are preserved in a public SG15 mailing list attachment from 2020. The official minutes record, with JF Bastien presenting, that "We decided not to promise ABI stability. ... We did not have a consensus for a big ABI break for C++23"; that record appears in N4855 and also in N4870. LEWG's own summary in N4852 reads "No consensus to coordinate an overall ABI break for C++23". No ABI break has occurred through C++26 development.

Titus Winters reported in P1863 that implementers already held an effective veto over ABI-breaking changes, and that an API-compatible unordered_map implementation 200-300% faster is "disallowed by ABI constraints".

Winters wrote in P1863R1, "ABI - Now or Never", published in 2020:

> *It's been the case for years that implementers effectively have a veto on ABI breaking changes - we have already been prioritizing ABI above design or performance concerns. ... we believe we could provide an API-compatible unordered_map/std::hash implementation that improves existing performance by 200-300% on average. This is disallowed by ABI constraints. ... We say "performance" but vote "ABI". This dissonance is harmful for the ecosystem.*

In P2028, Titus Winters reported that ABI constraints foreclose microbenchmark improvements of 300-400% to std::unordered_map, that he will not recommend std::unordered_map "for anything other than toy programs", and that regex and unordered_map are meaningfully inefficient and will be so forever.

The paper, P2028R0, "What is ABI, and What Should WG21 Do About It?", is dated January 2020 and states:

> *More advanced storage and probing techniques (as seen in absl::node_hash_map) could result in microbenchmark improvements of 300-400% over most unordered_map implementations. Without the ability to change these interfaces, we can never provide a salted hash container or a hash that provides any default resistance to hash collision attacks. ... there are things in the standard library that are meaningfully inefficient (regex, unordered_map) and will be so forever.*

Botond Ballo's trip report from the Prague 2020 meeting records that the Library Evolution group had been rejecting improvement proposals for years because they were ABI-breaking.

Ballo published the account in his Prague 2020 trip report on 2020-03-12:

> *the Library Evolution group has had to reject multiple proposals for improvements to existing library facilities over the past several years, because they would be ABI-breaking ... as time goes by the performance of these facilities will lag more and more behind the state of the art.*

The GCC 5 dual-ABI std::string transition shows what one ABI break actually costs an implementation.

Jonathan Wakely described the work on the gcc-patches mailing list in November 2014:

> *Because strings are used pervasively through the library loads of functions need to be compiled twice, using the old and new string, so that both versions are exported from the library ... I have devised a horrible hack where the exception classes always use the old COW std::string type, even when the rest of the program doesn't.*

C++11 outlawed the copy-on-write string in 2011, libstdc++ conformed in 2015, and the dual-ABI linker-error hazard is still documented in the vendor manual a decade later.

Microsoft's STL has promised binary compatibility across all toolsets since 2015, and its known performance bugs are formally deferred to a "vNext" break that has no ETA.

Microsoft guarantees that libraries built by VS 2015 link into applications built by VS 2017 through 2026, per the Microsoft Learn binary-compatibility page. Responding to a measured locale slowdown in microsoft/STL #1652, the maintainers wrote: "We're highly constrained by binary compatibility here, which is why we haven't made changes. ... There is no ETA for vNext".

Template internals are part of a standard library's ABI, and that mechanism is what freezes even header-only components.

Nicol Bolas explained the template mechanism in a Stack Overflow answer from January 2022:

> *Because of how templates get included, even types that nobody outside of the template library knows about are effectively part of the library's ABI. ... standard library implementers don't like breaking ABI because people don't upgrade to standard libraries that break their code.*

Google's Carbon project, reading the record from outside WG21, cites the ABI non-decision as a process failure.

The Carbon documentation page "Difficulties improving C++", published in 2022, states:

> *When pushed to address the technical debt caused by not breaking the ABI, C++'s process did not reach any definitive conclusion. This ... demonstrates how the process can fail to make directional decisions.*

Chandler Carruth's CppNorth 2022 keynote slides tie the same argument to P1863 and P2028.

std::regex: entered the standard in C++11 and was unfixable by 2020

A benchmark reported to WG21 measured libc++ std::regex at 28 minutes on a grep task that the system egrep completes in 22 seconds.

The measurement comes from P1433R0 by Hana Dusikova, dated 2019-01-21, and was run over a 1.3 GB CSV file with the pattern `([0-9]{4,16})?[aA]` on macOS with clang.

Implementation	Time (sec)
RE2	10.35
CTRE	11.11
PCRE2	16.92
boost::regex	20.37
BSD egrep (baseline)	21.92
std::regex (libc++)	1655 (28 minutes)

The paper states:

> *Because of all these problems and limitations users instead turn to non-standard libraries.*

The rust-lang/regex benchmark harness independently measured libstdc++ std::regex as "often being tens or hundreds of times slower than PCRE2 or RE2", a result recorded in rust-lang/regex PR #459 in March 2018, and the cross-engine rebar suite excludes std::regex entirely as "known to be horribly slow" while interpreted-language engines rank mid-table.

All three major standard library implementations have publicly acknowledged std::regex as unfixable in place.

The libc++ maintainer wrote in llvm/llvm-project #60991 in 2023 that "it's been pretty much untouched since it's been implemented more than ten years ago. ... don't expect huge improvements". Microsoft's issue microsoft/STL #405, open since 2019, carries the statement "vNext note: Resolving this issue will require breaking binary compatibility. We won't be

able to accept pull requests for this issue until the vNext branch is available", and the vNext branch has no ETA per #169. In libstdc++, regex crashes under the dual ABI, and GCC bug 118408, filed by Tim Song with maintainer comments running 2025-2026, records the assessment "It may be too late to fix this though".

The committee polled consensus in February 2020 to recommend deprecating std::regex, and no deprecation paper has appeared since.

The SG16 poll on P1844R1 is recorded in cplusplus/papers #597: "Do we want to recommend deprecation of std::regex without guarantee of replacement? SF 4 / F 1 / N 3 / A 0 / SA 0. Consensus? Yes". The follow-up tracking issue, sg16-unicode/sg16 #57, remains open with the label "paper needed". Writing on the public SG16 list in April 2020, Alisdair Meredith recorded the bind: deprecating without removing freezes the standard's binding to a 1999-era ECMAScript spec "forever".

std::unordered_map: the 2003 interface forecloses the 2017 state of the art

Meta's engineers stated the mechanism by which the std::unordered_map specification forecloses faster designs: the standard's reference-stability guarantee forces indirect, individually allocated entries.

The F14 announcement on the Meta engineering blog, written by Bronson and Shi and published on 2019-04-25, explains:

> *The standard guarantees reference stability: References and pointers to the keys and values in the hash table must remain valid until the corresponding key is removed. In practice, this means the entries must be indirect and individually allocated, which adds a substantial CPU overhead.*

Google's SwissTable achieved 2-3x better performance than std::unordered_map by breaking standard guarantees "multiple times".

Matt Kulukundis presented the design at CppCon 2017:

> *Our implementation of this new design gets 2-3x better performance with significant memory reductions (compared to unordered_map) and is being broadly deployed across Google. ... You might be asking, did we just break standards compatibility? Yes. Multiple times.*

Even a conformant rewrite gains a lot: the Boost.Unordered benchmarks show Boost 1.80's standard-compliant boost::unordered_map making lookup "almost twice as fast" relative to std::unordered_map, while the non-conformant boost::unordered_flat_map runs 14-29% faster than absl::flat_hash_map. Martin Ankerl's 2022 cross-benchmark reports that std::unordered_map is "Slow across the board. There is no benchmark where the map is fast compared to most of the competitors". After the Prague vote, no WG21 paper proposes a standard open-addressing hash table; the work happens entirely outside the standard.

std::filesystem: entered the standard in C++17, and deprecations began immediately

The std::filesystem function u8path completed a full add-deprecate-remove cycle inside three standard revisions.

The function was added in C++17 by P0218. It was deprecated in C++20 by the char8_t change of P0482R6, a paper that also changed u8string() return types and thereby made previously well-formed code ill-formed, a break that P1423R3 documents as outside the library's own SD-8 compatibility rights. Meredith proposed the removal for C++26 in P3364R0 in 2024.

A second wave of deprecations is active against the same std::filesystem::path class, because path::string() produces mojibake and data loss.

P2319R5, "Prevent path presentation problems", written by Victor Zverovich across 2024-2025, states:

> *some common path accessors still exhibit broken behavior, which results in mojibake and data loss. This paper proposes deprecating these accessors ... It disproportionately affects non-English C++ users making the language not as attractive for writing internationalized and localized software.*

Both libc++ and libstdc++ shipped the same std::filesystem TOCTOU symlink race that Rust patched as CVE-2022-21658, and both fixed it only days after the Rust advisory.

Rust's advisory of 2022-01-20 reported the attack as reliably reproducible "within a couple of seconds." libc++ fixed std::filesystem::remove_all on 2022-01-26 with the test taken directly from the Rust CVE patch, noting that "Windows doesn't support the necessary APIs to mitigate this issue". libstdc++ rewrote remove_all on openat/unlinkat the same month, retaining the racy code as the fallback, as described on the libstdc++ list in February 2022. The libc++ implementer had warned in a cfe-dev RFC in July 2018 that the directory_entry caching design standardized late in C++17 makes TOCTOU bugs unavoidable in every possible implementation, and he suggested reverting the standard changes.

Boost.Filesystem kept evolving after std::filesystem froze.

Boost.Filesystem v4 (2021 onward) ships deliberate semantic fixes that diverge from std::filesystem (dotfile stem/extension), capabilities the standard library lacks (Windows NT path prefixes, creation_time, durable-copy options), and continued performance work (sendfile/copy_file_range on Linux), all selectable per compile; Andrey Semashev documents the changes in the library's release history. On the performance side, a 2024 C++ Stories measurement found that on Windows, std::filesystem attribute queries tracked a baseline up to 129x slower than direct FindFirstFileEx in uncached scenarios, and the LLFIO author reported on the Boost list in May 2020 that both Boost and std recursive_directory_iterator are "orders of magnitude slower" than race-free NT enumeration, "which cannot be helped due to their design".

std::format: the best case still pays the Standardization Penalty

The standardized std::format permanently trails the living {fmt} library, with feature lag measured in years.

Feature	In {fmt} since	In the standard	Lag
print to stdout	~2014	std::print, C++23 (P2093)	~9 years
Range/tuple formatting	pre-2020	C++23 (P2286), partial	3+ years
fmt::join	pre-2020	not in any standard as of C++26	6+ years and counting
Named arguments	early {fmt}	not in any standard	10+ years and counting
Compile-time format compilation (FMT_COMPILE)	fmt/compile.h	no equivalent	n/a

The feature dates are drawn from the {fmt} API docs and from Barry Revzin's enumeration of the differences, written in 2020. Writing about std::print in 2023, Victor Zverovich noted: "the print function in almost its current form has been available in the {fmt} library ... since version 0.10.0 released 9.5 years ago."

C++20 std::format shipped unsafe and was repaired through retroactive defect reports, an approach viable only because no vendor had shipped the feature yet.

P2216R3 made invalid format strings a compile-time error, stating "We propose making it ill-formed resulting in a compile-time rather than a runtime error". The LEWG record on the paper's tracking issue is explicit about the loophole: "We voted in two breaking changes (wrt C++20) in the design of format. We did this with the understanding that std::format is not yet shipping in any implementation". Further defect reports were filed against published C++20: P2418R2 on non-const-formattable ranges, P2905R2 on runtime format strings, P2909R4 on char-as-integer results, P2572R1 on fill characters, and P2675R1 on width estimation. Even std::print needed two defect reports, P3107R5 and P3235R3, to permit an efficient implementation, and the runtime-format API added by the 2023 defect report is itself being renamed in C++26, as the cppreference feature-test trail records.

The shipped std::format implementations still trail the {fmt} library they copied.

Benchmarking GCC 16.1 in 2026, Matt Godbolt measured fmt::format_to at ~80 ns against libstdc++ std::format_to at ~130 ns and wrote that "it's clearly faster than libstdc++'s std::format (same API, different implementation)"; the numbers are in his benchmark commit of 2026-05-09. Microsoft's own tracking issue, microsoft/STL #1802, documented std::format in 2021 as "more than 3 times slower than its fmt counterpart" at launch, and a 2025 MSVC benchmark published on schneide.blog still has fmt::format at 75-80% of std::format's time. Choosing between the two in pytorch #158346, a PyTorch maintainer wrote: "I'd recommend fmt::format over std::format. It's faster, more efficient, and better spec'd".

std::execution: the Standardization Penalty arriving before the standard ships

A numbered committee paper tabulates the post-adoption correction ledger for std::execution.

Counting against the June 2024 adoption, P4041R0, published in 2026, records 2 removals (P3682R0, and P3149R11 replacing ensure_started), 1 rewrite (P3481R5, addressing bulk), 2 architectural fixes, 1 wording omnibus (P3396R1), 7 post-adoption additions, 5 LWG defects, 4 national-body ballot comment groups, and a 16-item "Sender Sub-Language" correction set. P3187R1 removed ensure_started, start_detached, and execute from the already-adopted design as unsafe, and P2300R10's own revision history confirms the removal "to be replaced with safer and more structured APIs."

In January 2026, the architect of std::execution wrote that early customization is "irreparably broken" and that the fix is a design change happening "uncomfortably close to the release of C++26".

Eric Niebler wrote in P3826R3, "Fix Sender Algorithm Customization", in January 2026:

> *While the sender/receiver concepts and the algorithms themselves have been stable for several years now, the customization mechanism has seen a fair bit of recent churn. ... early customization is irreparably broken and must be removed. ... There are risks with trying to fix the problem now. It is a design change happening uncomfortably close to the release of C++26.*

The paper is the third to fix the same mechanism, following P3303R1 and P3718R0. Implementing the fix in the reference implementation was a breaking change of +4,877/-4,042 lines across 159 files, merged on 2025-11-28 as NVIDIA/stdexec PR #1683.

The std::execution completion protocol structurally cannot perform C++20 symmetric transfer, and shipping the protocol in C++26 forecloses the fix.

P2583R4, a 2026 paper, states that "When a coroutine co_awaits a sender that completes synchronously, the stack grows by one frame per completion", and it reports that five of six surveyed coroutine libraries independently converged on the coroutine_handle-returning mechanism the sender protocol cannot express. The adopted task type concedes the limitation in P3552R3. P4007R2 classifies the gap as foreclosed by shipping C++26 and lists 15+ open issues in std::execution::task. Vendors are hanging back: in P3826R3, Niebler writes that "the major Standard Library vendors seem to be in no rush to implement std::execution".

std::linalg: fix papers arrived before any implementation exists

Between its November 2023 adoption and mid-2026, std::linalg accumulated a ledger of fix papers, all filed by the P1673 lead author against his own facility, alongside a run of LWG issues.

The fix papers are P3050, on conjugated template bloat and framed as "now or never"; P3222R1, on transposed dispatch to optimized BLAS being blocked; and P3371R5, on rank updates inconsistent with the BLAS, which changed overload semantics. The LWG issues are 4136 (undefined behavior in Hermitian algorithms), 4137 (incorrect Mandates/Preconditions/Complexity), 4302 (vector_sum_of_squares deleted from the standard entirely, partly because

LAPACK's algorithm moved on: "If somebody cares sufficiently, they can propose it back for C++29"), 4315 ("nowhere in [linalg] explains how to compute a square root"), and 4514 (norms could return complex results).

No vendor ships std::linalg, and the only usable implementation is the third-party reference library that standardization was meant to obviate.

Microsoft's tracking issue, microsoft/STL #4170, has been open since 2023-11-12 and states "We are not yet accepting PRs for C++26 features". In libc++, the `__cpp_lib_linalg` feature-test macro is unimplemented, and the tracking issue, llvm #105408, is untouched since creation. A user who wants std::linalg must fetch kokkos/stdBLAS, whose BLAS support is "currently experimental."

The long tail of never-fixed components

Across the standard's whole history, components have been recognized as defective within years of shipping and never fixed, as the following table records.

Component	Defect recognized	Outcome
vector	LWG issue 96 opened 1998; removal proposed N2160 (2007)	Closed NAD: "Attempts to fix this directly have not been tractable, and removing it has not been tractable"
valarray	Abandoned by proponents before C++98 shipped; Josuttis: "nobody tried to determine whether the final specification worked" (quoted record)	Dos Reis's redesign "turned down because the standardization was coming to a close"; unchanged since
initializer_list	Fix proposals N2801 (2008), P0065 (2015)	2024 answer from a committee regular: "nope, no plans, nothing can be done" (std-proposals)
optional<T&>	Cut from C++17 over an assignment-semantics deadlock; "the canonical example of failure to implement" (P1683R0)	Landed in C++26, roughly 15 years after the same debate in Boost
strstream	Deprecated at birth in C++98	Removed in C++26, "almost 30 years" later (P2867R1); P2863 is the first systematic Annex D sweep ever
iostreams	N4412 catalogs shortcomings (2015); locale coupling unfixable under bincompat (microsoft/STL #1652)	Coexists with printf and std::format; three parallel formatting systems specified simultaneously (P0645R0 motivation)

9. Corroborating voices

Titus Winters, then LEWG chair, laid out a scope formula for the standard library, together with an explicit exclusion, in a 2018 essay on the Abseil blog.

> *So, what should go in the standard library? Fundamentals. Things that can only be done with compiler support, or compile-time things that are so subtle that only the standard should be trusted to get it right. Vocabulary. Time, vector, string. Concurrency. Mutex, future, executors. The wide array of "sometimes useful data structure" and "random metaprogramming" could and should be shuffled off into a semi-standard repository of available but distinct utility libraries. ... Graphics is too much - many won't want it, and those that think they do may not understand what they are asking for.*

Winters published this scope formula in the Abseil blog essay "What Should Go Into the C++ Standard Library" on 2018-02-27.

Titus Winters argued in his 2018 Abseil blog essay that ongoing optimization research disqualifies a component from the standard.

> *Anything complex enough that research tasks on how to optimize it are still ongoing should probably not be in the C++ standard. At least, not until we figure out how to deal with change a little more aggressively.*

The passage comes from the same 2018-02-27 Abseil blog essay, which also documents the mechanism: "even a change to std::hash is infeasible because we have to maintain binary compatibility with code compiled years ago ... we're paying in both resources and safety because of our addiction to binary compatibility."

In a tweet posted on 2020-02-03, Titus Winters compared the committee's commitment to a stable ABI to aiming for McDonald's.

> *Committing to ABI is like admitting that the standard library is aiming to be McDonald's - It's everywhere, it's consistent, and it technically solves the problem.*

Winters posted the comparison on Twitter on 2020-02-03, and it is preserved with attribution in cor3ntin's post "The Day The Standard Library Died".

Bryce Adelstein Lelbach, then LEWG chair, said during a 2020 CppCon panel that the standard library should hold the things that cannot go anywhere else.

> *Essentially, the things that I think belong in the Standard Library are the things that cannot go anywhere else. ... Vocabulary types. ... Abstractions around the platform. ... Things that require language support. ... our focus should be on a really high quality but focused standard library. But then, we should also be spending more effort on making it easier to use third-party libraries which means enabling things like packaging, et cetera.*

Lelbach said this at the CppCon 2020 Standards Committee Fireside Chat in 2020, and the transcription provenance is documented at langdev.stackexchange.

In a tweet posted on 2020-02-16, Bryce Adelstein Lelbach stated a trilemma for the standard library, and as LEWGI chair he institutionalized a scarcity poll in the committee.

> *Performance. ABI Stability. Ability to Change. You can pick two, choose wisely.*

Lelbach posted the trilemma on Twitter on 2020-02-16, and it is preserved in cor3ntin's ABI post. As LEWGI chair, Lelbach coined the poll "Knowing that our resources are scarce and our time is limited, do we want to give more time to this proposal?", which, as Corentin Jabot wrote in "What is the standard Library?" on 2020-09-20, "has become somewhat of a meme in the committee".

Stephan T. Lavavej, principal maintainer of Microsoft's STL, delivered the implementer's verdict on standard networking in a 2024 Reddit comment.

> *The Standard Library is the worst package manager, and Standard Library maintainers aren't domain experts. What you should want is a good networking library available through a good package manager (e.g. vcpkg) - you shouldn't want it to be in the Standard, where it'll be maintained by generalists (like me!) instead of proper specialists, updated on a toolset update cadence ... and subject to the usual ABI restrictions. ... Asking it to print "3.14" is about the upper limit. Don't ask it to do networking.*

Lavavej wrote the comment on r/cpp in 2024.

Billy O'Neal, MSVC STL maintainer, explained in 2019 why the ABI freeze makes standardizing networking uniquely risky.

> *My being weakly against standardizing networking doesn't come from thinking networking is an unimportant problem. It comes from being unable to reconcile the "ABI stabilized forever, can't fix mistakes ever" world we've created for ourselves with this relatively rapidly changing problem domain.*

O'Neal's explanation was quoted with permission on Arthur O'Dwyer's blog on 2019-10-09.

Corentin Jabot described the process economics of the committee in a 2018 essay, and in a 2020 essay written after the Prague ABI vote he concluded that standardizing a widely deployed library is a bad value proposition.

> *There is maybe one person in those meetings for every 200'000 c++ developers.*

Jabot wrote this in his essay "A cake for your cherry", published 2018-02-21, and after Prague he continued the thought.

> *A high-quality open-source library that is widely deployed reached its target audience, and "standardizing it" (a process which involves trimming features and freezing it in time), represents a bad value proposition. ... If the C++ committee is 10 years ahead of the bulk of the industry, and standardize practices that are 10 years behind, it means that we inflict 20 years old technologies to C++ users, for the next 30 years.*

The later passage appears in his essay "What is the standard Library?", published 2020-09-20.

Bjarne Stroustrup said during a 2020 CppCon panel that the distribution mechanism, not the standard, is the missing piece.

> *If we could get a decent package manager and distribution system, I predict [the C++ Standard Library] would have a logger within a year. Because there are several good ones.*

Stroustrup made the prediction at the CppCon 2020 Standards Committee Fireside Chat in 2020.

Hyrum Wright stated the change-management premise in a tweet posted on 2020-02-16.

> *You will eventually want to change every single decision you make. Build the ability to change into your ecosystem, both tooling and process.*

Wright posted this on Twitter on 2020-02-16, and it is preserved in cor3ntin's ABI post. The general law is in print in Software Engineering at Google, ch. 1, published 2020: "with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody".

Jonathan Müller, writing as foonathan, independently derived the same three criteria (compiler magic, vocabulary types, and ubiquitous facilities) and the same exclusion of graphics in a 2017 blog post.

> *[The standard library should contain things that need] compiler magic, [vocabulary types, and ubiquitous facilities.] ... [On graphics:] It certainly doesn't involve compiler magic and isn't useful for most applications. Furthermore, it will never be good enough for real serious use-cases.*

Müller published the post as "What should be part of the C++ standard library?", dated 2017-11.

10. Cross-language records

The provenance trace of the "where code goes to die" aphorism

The aphorism originated with Kenneth Reitz, who said at PyCon US 2012, speaking about Python, that the standard library is where modules go to die.

> *Near the end of his talk, Reitz says, "The standard library is where modules go to die." An overstatement, certainly, but with more than a germ of truth. Once a library is enshrined in the standard set, it can't change radically because too many programs rely on it - and its bugs, idiosyncrasies, and complications - remaining stable.*

The contemporaneous report of the talk is Dr. Drang's post "Where modules go to die", published 2012-04.

Aaron Turon, the Rust libs lead, adopted the aphorism as design policy in 2016.

> *The standard library also takes a deliberately minimalistic approach, to avoid the well-known pitfalls of large standard libraries that are versioned with the compiler and quickly stagnate, while the real action happens in the broader ecosystem ("std is where code goes to die").*

Turon wrote this in "The Rust Platform", published 2016-07-27. The HN thread discussing the post, from 2016-07, attributes the lineage: "The 'std lib is where libraries go to die' was invented by Python".

The official Rust blog canonized the aphorism in a 2017 post.

> *There's a countervailing mindset which, in its harshest terms, says "the standard library is where code goes to die" (a complaint sometimes heard about batteries-included standard libraries). Because the standard library is coupled to the language itself, it is released at the same frequency as the language is, and making breaking changes means breaking the whole language.*

The passage appears in the Rust blog post "The Rust Libz Blitz", published 2017-05-05.

Amber Brown restated the aphorism at the 2019 Python Language Summit, and the Python Software Foundation's official report preserved her wording.

> *According to Brown, "the standard library is where code sometimes goes to die," because it is difficult and slow to contribute code there.*

The quote comes from the PSF blog report of Amber Brown's "Batteries Included, But They're Leaking", published 2019-05. The verified variants are only the talk title and the phrase "Python's batteries are leaking", and no public primary source exists for an "acid" variant.

By 2025 the aphorism had become standard vocabulary in cross-language standard-library criticism.

> *Perhaps standard libraries' biggest critique is that they tend to atrophy ("the standard library is where modules go to die") ... you can tell from looking at a library which era it was added to the standard library in.*

The example comes from Alex Gaynor's essay "Standard Libraries and their Discontents", published 2025-05-19.

Rust

Rust's accepted RFC 1242, from 2015, states the minimalism policy for the standard library and the reason for it.

> *the stability promises made in std are tied to the versioning of Rust itself, as are updates to it, meaning that the standard library has much less flexibility than other crates enjoy. While we do plan to continue to grow std ... we still plan to take a minimalistic approach.*

The policy appears in Rust RFC 1242, accepted in 2015.

Rust's substitute for a large standard library, the crates.io ecosystem, is measurable, and vocabulary coordination happens outside std by design.

By July 2025, crates.io carried 189,000+ published crates and 151 billion cumulative downloads, according to a Socket report, and the Rust Foundation technology report 2025 records 50.2 billion downloads in 2024 alone. The http crate states its own role: "It's intended that this crate is the 'standard library' for HTTP clients and servers without dictating any particular implementation". The serde crate plays the same role for serialization as a third-party crate.

Python

Python's PEP 594, created in 2019, removed 20+ modules from the standard library and states that the package ecosystem changed the calculus.

> *With the introduction of PyPI (nee Cheeseshop), setuptools, and later pip, it became simple and straightforward to download and install packages. ... Any additional module increases the maintenance cost for the Python core development team. ... The removal of an unmaintained stdlib module increases the chances of a community-contributed module to become widely used.*

The proposal is PEP 594, authored by Heimes and Cannon, created in 2019 and implemented in Python 3.11-3.13. The burden has a two-decade paper trail, since PEP 3108 (2007) and PEP 206 (2000) are both quoted in PEP 594.

PEP 2, Python's standing acceptance procedure for new modules since 2001, states the perpetual-cost rule that C++ never wrote down.

> *Due to the visibility and importance of the standard library, it must be maintained thoughtfully. As such, any code within it must be maintained by Python's development team which leads to a perpetual cost to each addition made.*

The rule appears in PEP 2, "Procedure for Adding New Modules", created 2001-07-07. During the PEP 594 debate, core developer Victor Stinner quantified the invisible burden with the example of sporadic nntplib CI failures where "nobody managed to come with a fix ... in 6 years", an account LWN reported on 2019-06-12.

Zig

The Zig project actively removes code from its standard library and maintains an official orphanage repository for the removed code.

> *This is code that used to be in the standard library, but there wasn't any reason to keep maintaining it there when it could function just fine as a third party package. Feel free to start your own source code repository and take over the duties of maintaining any of this code!*

The invitation appears in the ziglang/std-lib-orphanage README, a repository in the official ziglang org that has been active c. 2019-2025. The community states the operative scope rule in a 2025 Ziggit thread: "Zig's std makes much more sense when you think of it as 'just enough to get the compiler/build system to work.' So no CSV ... or regex". According to Andrew Kelley in ziglang/zig #943, the package manager is the delivery mechanism.

Go: the counterexample, and why it does not transfer

Go pairs a large curated standard library with a single implementation and no ISO process, and on the official Go blog in 2023 Russ Cox stated the source-level compatibility guarantee that Go's leadership calls its most important decision.

> *There will not be a Go 2 that breaks Go 1 programs. Instead, we are going to double down on compatibility, which is far more valuable than any possible break with the past.*

Cox wrote this in "Backward Compatibility, Go 1.21, and Go 2", published 2023-08 on the official Go blog. The same post documents the mechanisms: compatibility is source-level only ("When you update to a new version of Go, you do have to recompile your code"), every release is tested against all of Google's internal Go code, and GODEBUG preserves old behaviors per-module. None of these mechanisms (single implementation, recompile-the-world, centralized testing, per-release opt-outs) is available to an ISO-standardized, multi-implementation, ABI-frozen library.

Node.js

Node.js core contributors stated the small-core philosophy in operational terms in a 2017 essay for the Node.js Collection.

> *The longer you work with Node.js core, supporting, bug fixing, and documenting the existing APIs, the more you wish that Node.js had been more conservative in accepting new APIs in the past, and that those APIs had been implemented as npm modules that could be independently maintained and versioned.*

The essay is "Keeping the Node.js core small", published in the Node.js Collection in 2017. Companion formulations appear elsewhere: Mathias Buus wrote that "Innovation shouldn't come from Node core. It should come from modules … Core's job is to be boring", and Mikeal Rogers told the Changelog that "things that should be in Node Core are things that cannot be done effectively outside of Node Core".

11. Counterargument inventory

This section states each opposing claim in its strongest form, attributes it precisely to its source, and then presents the countervailing evidence. The section renders no verdicts.

The claim that batteries should be included: a larger useful core drives adoption, and C++ has not reached critical mass

Guy Davidson, co-author of the 2D graphics proposal, states the batteries-included case in a 2018 essay.

> *I think the library is impoverished and needs fleshing out to make C++ a stronger contender in the development arena. ... There is clearly a "critical mass" that can be achieved, and while those languages have erred on the other side of that mass, C++ has yet to reach it. I would rather we overshot a little and reaped the benefit of greatly increased adoption. ... Would you like to open a socket? Nope, can't do that. ... It's still the 1970s in C++ land.*

Davidson made the argument in "Batteries not included: what should go in the C++ standard library?", published 2018-02-16, and the full text survives via the archive. His strongest non-convenience argument is that "not all code shops will permit the use of third party code beyond a language and its library. There are commercial, industrial and military environments that insist all code is created in a 'clean room' environment."

The same case was argued on the Python side during the 2019 debates over PEP 594.

> *Many Python users don't have the privilege of being able to install arbitrary, unvetted packages from PyPI. They get to use only packages from approved vendors, including the stdlib, what they write themselves, and nothing else. Please don't dismiss this part of the Python community.*

The speaker is Steven D'Aprano, writing on python-dev, as reported by LWN on 2019-06-12. Barry Warsaw said in the same venue: "Let's not underestimate the value that this has for our users, and the contribution such a stdlib has made to making Python as popular as it is."

The countervailing record opens with Titus Winters, the strongest sympathetic reader of the batteries case, who rejects the standard as the mechanism.

> *The section of Guy's "Batteries Included" piece that resonates with me is this: it isn't that the standard has to provide these things, but that there needs to be some mechanism to readily distribute libraries and dependencies in the C++ community. It is far past time that we as a community find an answer that is somewhere between "this is chaos" and "this is in the standard".*

Winters wrote this on the Abseil blog on 2018-02-27. The same essay takes the Java/Python comparison head-on and asks: "Given this demonstration of prioritizing efficiency and stability, why would something as high-level and changeable as Graphics be a good fit for the standard?"

The batteries-included language's own summit heard in 2019 that the batteries stifle the ecosystem.

> *Brown's most controversial opinion, in her own estimation, is that adding modules to the standard library stifles innovation, by discouraging programmers from using or contributing to competing PyPI packages. "We should embrace PyPI," she exhorted.*

The quotation is from the PSF report of Amber Brown's 2019 Language Summit talk, dated 2019-05, and Python subsequently removed 20+ modules via PEP 594, a removal that section 10 of this file records. The counter to the restricted-environments argument is contained in the reach data: the populations that cannot download code at all are not surveyed as a majority anywhere in the ISO survey series, while section 5 shows that the top pain point for the whole population is dependency management, not missing components.

The claim that C++ has no package manager, so the standard library must carry more

The argument as actually made appears in its strongest public forms in r/cpp comments from 2022 and 2024.

> *a successful modern language should either have a really good package manager (eg. NodeJS) to make up for its shitty standard library; or a really good standard library to make up for a shitty package manager. However, C++ is almost unique in having a shitty standard library that you cannot build networked software with, and no standard package manager.*

The comment is from r/cpp user Dworgi, posted in 2022. A companion form came from r/cpp user ghlecl in 2024: "C++ has no package management, installing external libraries and/or getting them approved is not always easy, incentivizing putting things in the standard library, which is always installed".

The countervailing evidence from Rust shows that where dependencies are frictionless, the availability argument evaporates entirely.

> *When all 3rd party code is just as easy to depend on as the standard library, there's no longer any special availability advantage to being in the standard library. The idea that any "already solved problem" needs to be "standardized" simply evaporates into a non-problem, because once it's on crates.io, then "everyone has it" in practice already. Moving it into std does not increase the availability any further. ... in the specific case of an HTTP server, there is an obvious vocabulary types problem, but the ecosystem ended up agreeing on a 3rd party crate for that and AFAIK no one's ever asked for it to be moved into std.*

The passage comes from the Rust internals forum thread "Expansion of standard library", posted in 2019.

The countervailing analysis of the blogger cor3ntin holds that the stdlib is a bad implementation of a package manager even on its own terms.

> *Fundamentally, that makes the standard library a bad package manager. If the only motivation for something to be in the standard is to palliate to the lack of good package managers, then it's probably not a good motivation. (Especially as the standard library is super bad at availability...) ... The number one requirement for a package manager of any sort is to be authoritative. It doesn't even necessarily need to be good.*

The argument spans two cor3ntin essays, "What is the standard Library?" of 2020-09-20 and "A cake for your cherry" of 2018-02-21. The availability point is empirical in this file: section 6 shows standardized components arriving one to six-plus years late per vendor and sometimes never, and section 9 gives the implementer's version in Lavavej's line that "The Standard Library is the worst package manager".

The claim that standard blessing drives adoption and awareness beyond what downloads achieve

Matt Bentley, the author of the hive proposal, argues in his 2022 paper that generalized components spread awareness and ease communication.

> *Reports from other fields suggest that, because most developers aren't aware of containers such as this, they often end up using solutions which are sub-par for iterative performance such as std::map and std::list in order to preserve pointer validity ... introducing this container would both create a convenient solution to these situations, as well as increasing awareness of better-performing approaches in general. It will also ease communication across fields...*

The passage is from P0447R19, which Bentley published in 2022.

The linalg authors, Hoemmen et al., present their form of the claim in the 2023 paper P1673: the strongest possible existing practice, priorities on record, and sort-like tunability.

> *Linear algebra is like sort: obvious algorithms are slow, and the fastest implementations call for hardware-specific tuning. ... The set of linear algebra operations in this proposal are derived from a well-established, standard set of algorithms that has changed very little in decades. It is one of the strongest possible examples of standardizing existing practice that anyone could bring to C++.*

The passage is from P1673R13, published in 2023.

The countervailing record shows that the coordination this claim invokes was already achieved outside the language standard, and that the blessed version is the one nobody can use. The proposal's own companion section, P1673R13

section 6, concedes that the BLAS solved the Coordination Problem externally: "The BLAS is a standard that codifies decades of existing practice. ... Optimized third-party BLAS implementations with liberal software licenses exist". The same section names AMD BLIS, ARM, Cray, Intel MKL, IBM ESSL, and NVIDIA cuBLAS among those implementations. More than two and a half years after adoption, no standard library ships std::linalg, and section 8 records that the only implementation is the third-party Kokkos reference. Meanwhile the unblessed alternatives hold the field: section 7 shows Eigen powering TensorFlow, Ceres, ROS, and Stan with ~2,765 academic citations, and the ubiquity champions of C++, curl at 20-30 billion installs and NumPy at 21.2% of all developers in the SO 2024 survey, achieved reach through adoption, not standardization.

The claim that standardization worked fine for format

Victor Zverovich, the author of the format proposal, gives the claim its strongest form in his own 2019 testimony.

> *Contrary to the usual "design-by-committee" narrative, the standardization process was tremendously beneficial both for the proposal and the {fmt} library, resulting in numerous improvements...*

Zverovich wrote this in the post "std::format in C++20" of 2019-07-23, where he lists the committee-driven improvements: iterator support, output-size precomputation, constexpr format-string processing, and round-trip floating-point defaults.

The countervailing evidence sits in Zverovich's own timeline and in the maintained comparison. Zverovich recorded in a 2023-12 post that fmt::print existed ~9.5 years before std::print and that "It took extra 3 years for std::print". The shipped C++20 design needed the retroactive DR stack that section 8 documents, and the practitioner comparison lands where the Standardization Penalty model predicts:

> *Our std header moves with the C++ standard ... stable. ... On the other hand the standalone lib {fmt} is constantly being improved with bug fixes, performance fixes; there are updates available. ... if you're in an environment where you do have some sort of package manager or some way to take advantage of lib {fmt}, then I think it wins in this comparison here.*

The comparison is Jason Turner's, delivered in C++ Weekly Ep 341, "std format vs lib {fmt}" in 2022. The format case is the best-executed library standardization of the modern era, field-proven for 6.5 years and author-driven, and it still ships behind the living library on features, performance, and availability. The counterargument's best case is the Standardization Penalty's cleanest exhibit.

12. Source index

The final count is 150 evidence cards. Each evidence card leads with a single bold sentence; the count spans sections 4-11 and includes the summary provenance table and the tabulated ledgers; each card is counted once.

The cards are distributed across the sections as follows:

Section	Cards
4. The committee's own criteria on record	21
5. Coordination problems and reach	20
6. The Complexity Budget, priced	18
7. Case dossiers	29
8. The Standardization Penalty record	27
9. Corroborating voices	11
10. Cross-language records	12
11. Counterargument inventory	12

The evidence draws on the following primary source domains: open-std.org (WG21 papers, working drafts, published minutes), isocpp.org (standing documents, developer survey PDFs, blog), iso.org (catalog records), wg21.link and github.com/cplusplus/papers (paper index and tracker), timsong-cpp.github.io and eel.is (rendered standard sources), gcc.gnu.org (release notes, Bugzilla, mailing lists), llvm.org and libcxx.llvm.org (status pages, commits, cfe-dev), github.com (microsoft/STL, NVIDIA/stdexec, fmtlib, nlohmann, boostorg, kokkos, rust-lang, ziglang, code search API), learn.microsoft.com and devblogs.microsoft.com, boost.org and listarchives.boost.org, lists.isocpp.org (public SG16/SG15/std-proposals archives), herbsutter.com and botondballo.wordpress.com (trip reports), abseil.io, cor3ntin.github.io, quuxplusone.github.io, vitaut.net, accu.org, peps.python.org, pyfound.blogspot.com, lwn.net, blog.rust-lang.org, rust-lang.github.io, internals.rust-lang.org, rustfoundation.org, go.dev, ziggit.dev, docs.rs, daniel.haxx.se, developernation.net, survey.stackoverflow.co, blog.jetbrains.com, stackoverflow.com, reddit.com (r/cpp), youtube.com (conference talks), cppreference.com, netlib.org, sc19.supercomputing.org, docs.carbon-lang.dev, chandlerc.blog, kitware.com, sourceforge.net, engineering.fb.com, leancrew.com, alexgaynor.net, moderncppdevops.com, cppstories.com, schneide.blog, github.com/mattgodbolt (performance benchmarks).

2026-07-08 - compiled from the 2026-07-08 stdlib criteria research