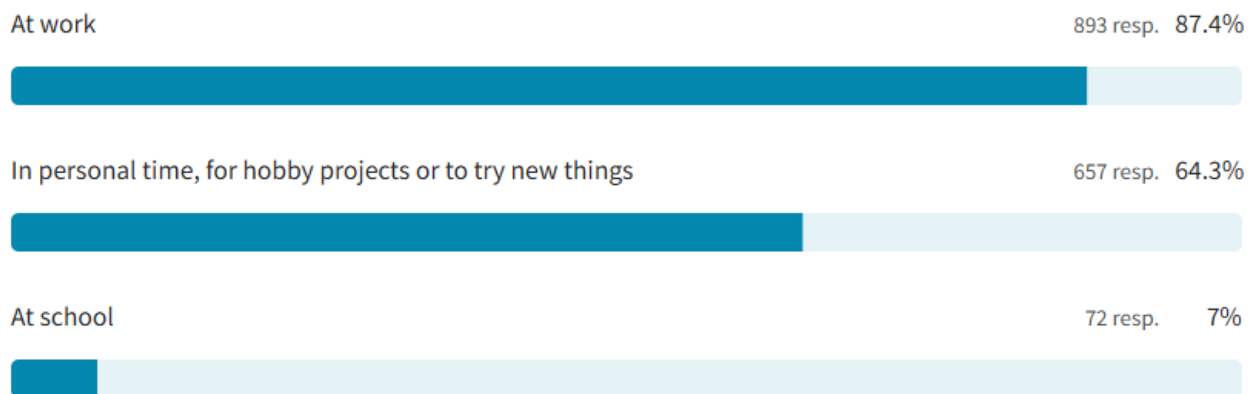


2025 Annual C++ Developer Survey "Lite"

1,036 responses

Where do you use C++?

1022 out of 1036 answered



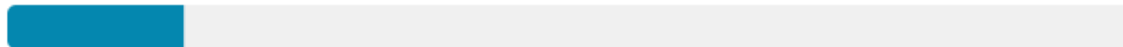
How many years of programming experience do you have in C++ specifically

1026 out of 1036 people answered this question

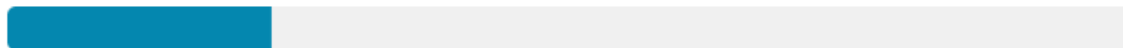
1-2 years 60 resp. 5.8%



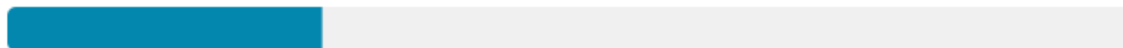
3-5 years 161 resp. 15.7%



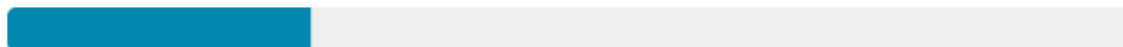
6-10 years 241 resp. 23.5%



10-20 years 287 resp. 28%



>20 years 277 resp. 27%



How many years of programming experience do you have overall (all languages)?

915 out of 1036 people answered this question

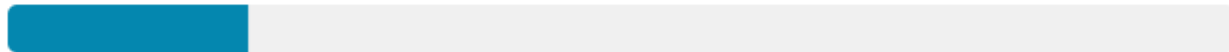
1-2 years 6 resp. 0.7%



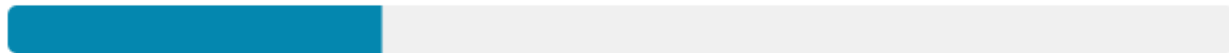
3-5 years 81 resp. 8.9%



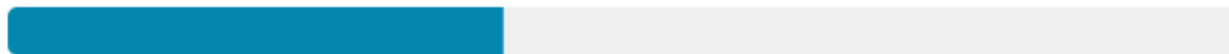
6-10 years 179 resp. 19.6%



10-20 years 279 resp. 30.5%

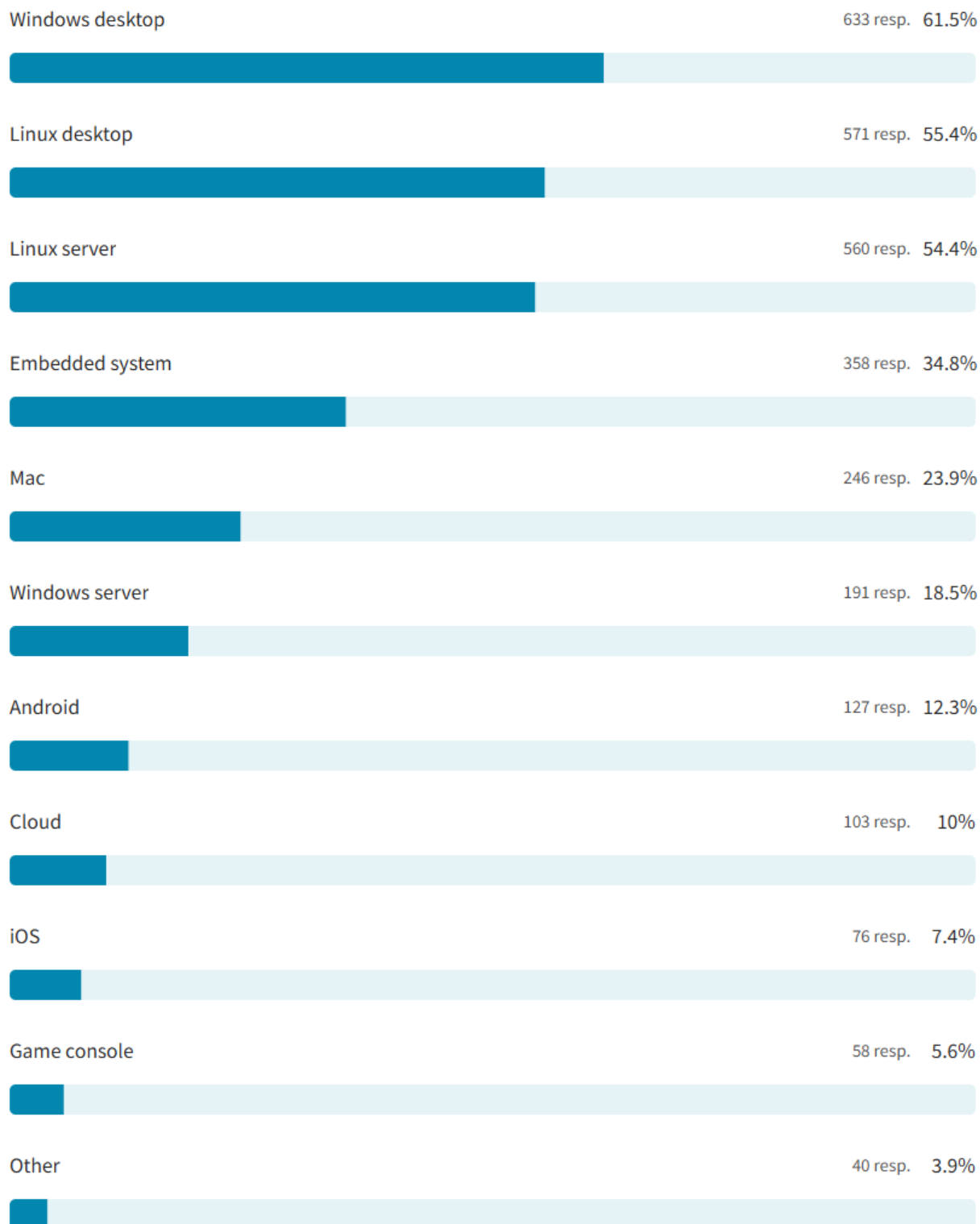


>20 years 370 resp. 40.4%



What platforms do you develop for?

1030 out of 1036 answered



What types of projects do you work on?

1022 out of 1036 answered

Hardware/IoT (e.g., embedded systems, home automation) 315 resp. 30.8%



Engineering (e.g., avionics, power management) 313 resp. 30.6%



Developer tools (e.g., compilers, code editors) 260 resp. 25.4%



Utility apps (e.g., calculators, simple image editors) 222 resp. 21.7%



High performance computing (HPC) 209 resp. 20.5%



Gaming (e.g., console and mobile games) 205 resp. 20.1%



Business (e.g., B2B, B2E) 203 resp. 19.9%



Consumer (e.g., retail websites, mobile apps) 107 resp. 10.5%



Machine learning 105 resp. 10.3%



Artificial intelligence 99 resp. 9.7%



Entertainment (e.g., sports apps, video streaming) 93 resp. 9.1%



Frameworks (e.g., React, Unity)

84 resp. 8.2%



Financial (e.g., trading, mortgage, asset management)

83 resp. 8.1%



Productivity (e.g., budget tracking, note taking)

64 resp. 6.3%



Social and business networking (e.g., Facebook, Twitter)

10 resp. 1%



Other

131 resp. 12.8%



How do you manage your C++ 1st and 3rd party libraries?

1021 out of 1036 answered

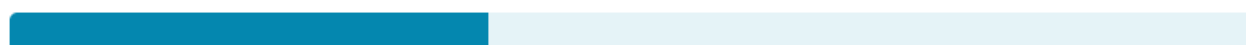
The library source code is part of my build 619 resp. 60.6%



I compile the libraries separately using their instructions 453 resp. 44.4%



System package managers (e.g., apt, brew, ...) 393 resp. 38.5%



Conan 234 resp. 22.9%



I download prebuilt libraries from the Internet 219 resp. 21.4%



Vcpkg 203 resp. 19.9%



Nuget 55 resp. 5.4%



None of the above, I do not have any dependencies 23 resp. 2.3%



Other 110 resp. 10.8%



What build tools do you use?

1016 out of 1036 answered

CMake 855 resp. 84.2%



Ninja 456 resp. 44.9%



Make/nmake 325 resp. 32%



MSBuild 296 resp. 29.1%



distcc/ccache 122 resp. 12%



Autotools 94 resp. 9.3%



QMake 71 resp. 7%



Meson 68 resp. 6.7%



Xcode projects 68 resp. 6.7%



Bazel 58 resp. 5.7%



Gradle 39 resp. 3.8%



Boost Build (bjam) 33 resp. 3.2%



Maven 31 resp. 3.1%



IncrediBuild 30 resp. 3%



Scons 26 resp. 2.6%



FastBuild 9 resp. 0.9%



Goma 1 resp. 0.1%



BuildXL 0 resp. 0%



Other 116 resp. 11.4%



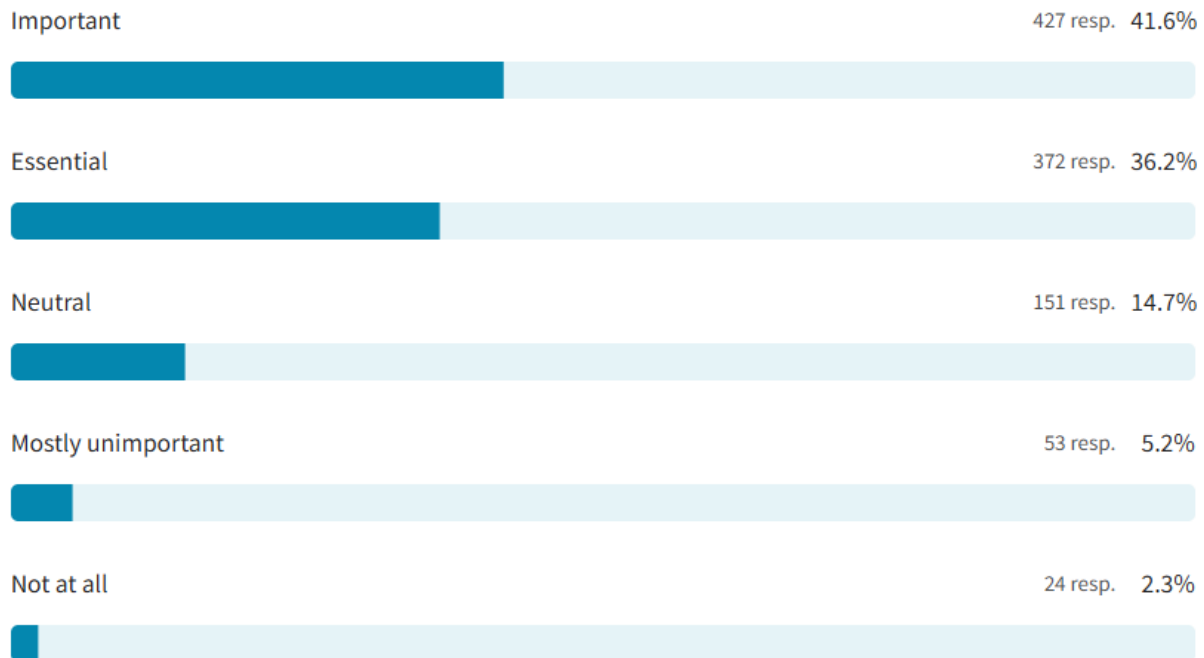
What version(s) of C++ are you allowed to use on your current project (work or school)?

945 out of 1036 answered

	Yes: Pretty much all	Partial: Limited features/usage	No: Not allowed
C++98/03 (e.g., exceptions, templates, RTTI)	84.7%	8.6%	6.7%
C++11 (e.g., auto, move semantics, =delete/=default,...)	91.3%	5.8%	2.9%
C++14 (e.g., generic lambdas, auto return types, general...)	87.1%	9.2%	3.7%
C++17 (e.g., if constexpr, if/switch scoped variables,...)	81.2%	13.3%	5.5%
C++20 (e.g., concepts, coroutines, modules)	43.2%	32.2%	24.6%
C++23 (e.g., expected, mdspan)	26.9%	22.5%	50.6%

How important is memory safety in how you view your future use of C++?

1027 out of 1036 answered



What tools do you rely on for managing code safety in C++?

993 out of 1036 answered

Use new C++ standards (e.g., smart pointers, C++20) 861 resp. 86.7%



Static analysis tools (e.g., clang-tidy, C++ Core Checkers) 819 resp. 82.5%



Coding standards (e.g., Core Guidelines, Google, internal) 639 resp. 64.4%



Sanitizers (e.g., Asan, TSan) 620 resp. 62.4%



Hardening options in existing libraries (e.g., STL checked modes) 232 resp. 23.4%



Fuzzing tools (e.g., libfuzzer or OneFuzz) 111 resp. 11.2%



Interop with memory-safe languages (e.g., Rust, Go, C#, Java) 78 resp. 7.9%



Annotation libraries (e.g., SAL, Checked C) 31 resp. 3.1%



Other 45 resp. 4.5%



How often have you utilized AI assistants for the following C++ coding tasks, if at all?

1002 out of 1036 answered

	Almost every day	Often (1-2x/week)	Sometimes (1-2x/month)	Rarely (every few months)	Never
Writing new code	14.3%	16.6%	16.5%	14.7%	38%
Writing tests	8.2%	11.8%	12.3%	13.1%	54.5%
Understanding code	6.8%	10.1%	13.7%	15.8%	53.6%
Debugging code	4.6%	6.9%	10.6%	13.1%	64.8%
Identifying performance issues	2.9%	3.1%	7.6%	12.8%	73.6%
Code review	3.7%	5.8%	8.7%	14.6%	67.3%
Refactoring code	5.6%	8.6%	12.1%	15.1%	58.6%
Learning unfamiliar coding...	8.5%	14.5%	18.5%	14.8%	43.6%
Writing documentation and other...	8.2%	11.9%	14.5%	11.8%	53.6%

What issues or blockers have you faced when using AI assistants for C++ coding?

937 out of 1036 people answered this question (with multiple choice)

AI-generated output is incorrect 704 resp. 75.1%



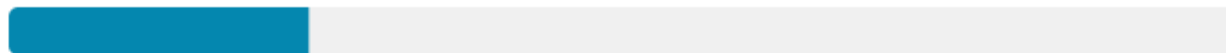
Concerns around data privacy 511 resp. 54.5%



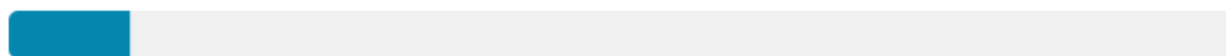
Lack of understanding relevant context 519 resp. 55.4%



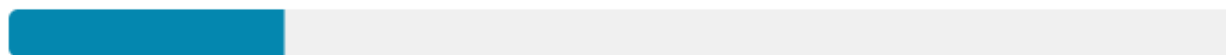
Limited integrations with existing tools 230 resp. 24.5%



Lack of customization 93 resp. 9.9%



Cost of AI tools 211 resp. 22.5%

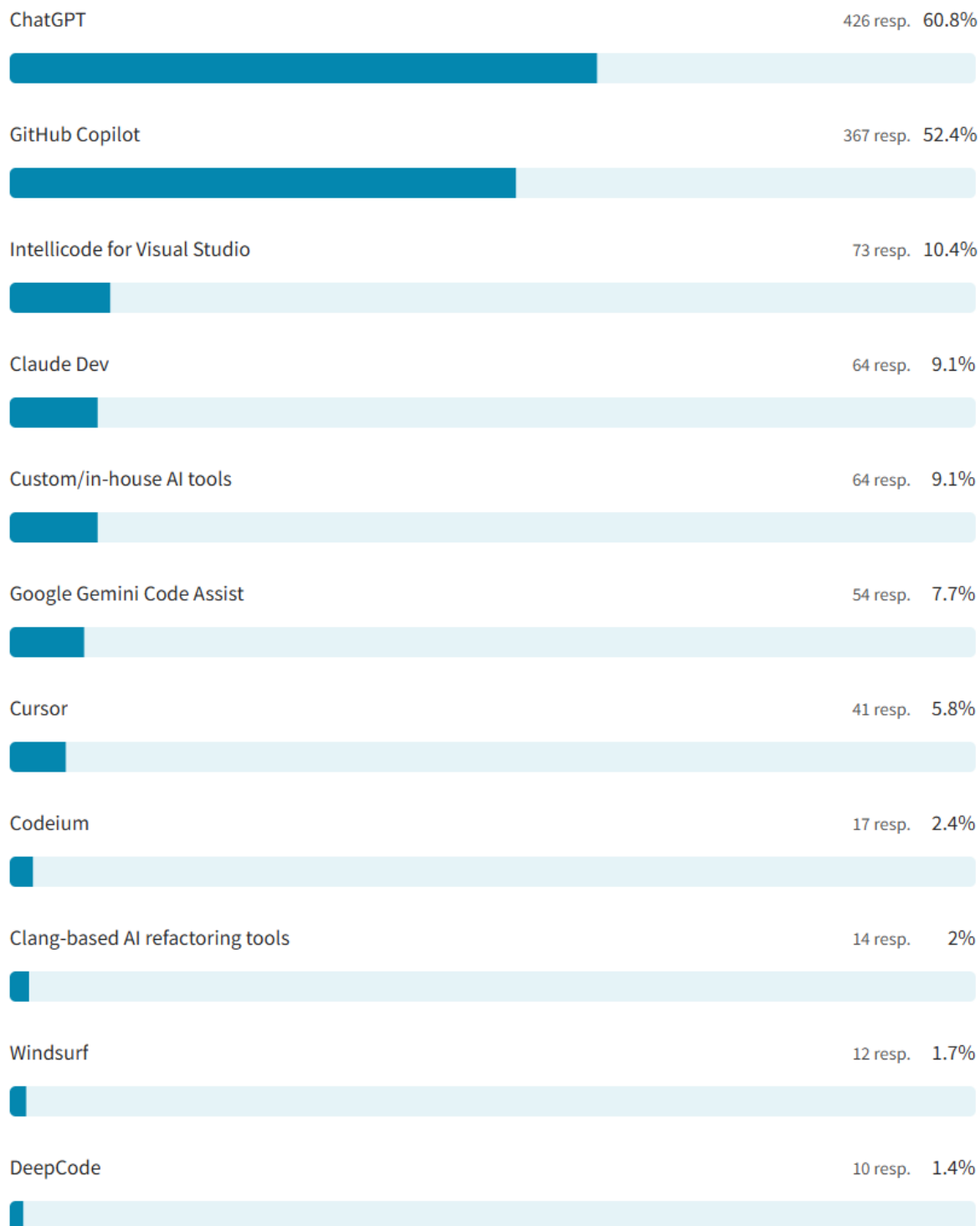


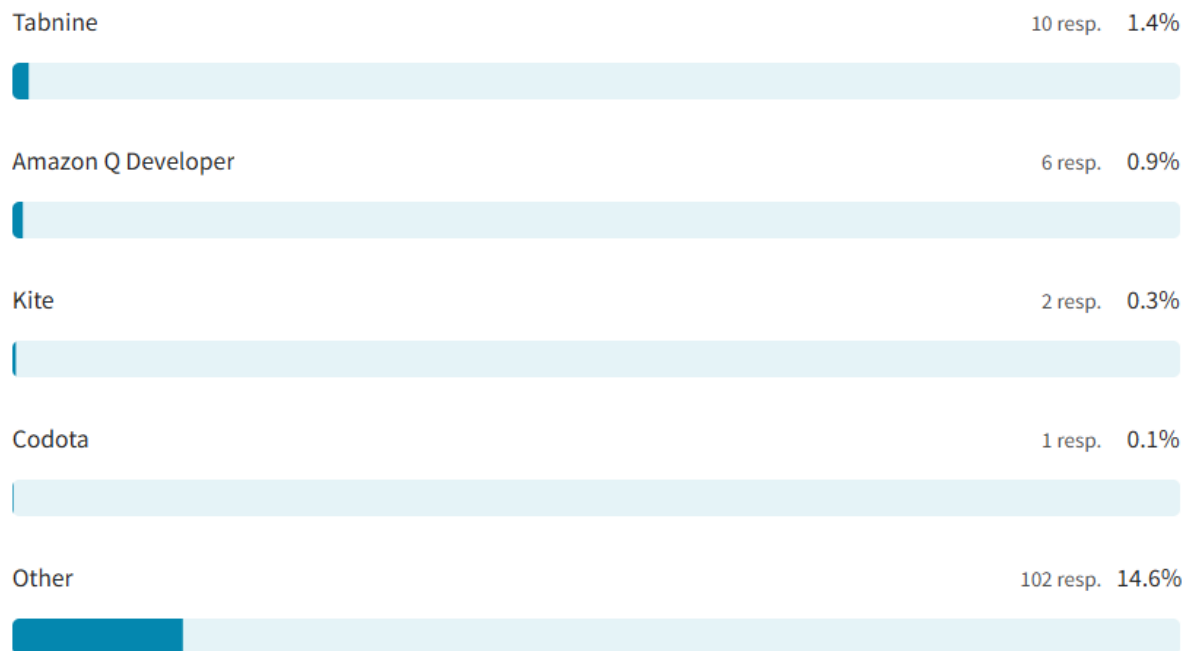
Lack of trust in output 663 resp. 70.8%



What AI-powered code development tools do you use in your current and recent projects?

701 out of 1036 answered





Has AI meaningfully improved your productivity as a C++ developer? If so, how?

646 out of 1036 people answered this question

[AI-generated summary]

Primary Themes (most commonly cited and emphasized)

1. Boilerplate & Repetitive Code Automation

Most common benefit cited. AI helps reduce the tedium of writing repetitive or verbose C++ code (e.g., getters/setters, switch statements, tests). Frequently mentioned as a "smarter autocomplete."

2. Prototyping & Drafting Code

AI aids in quickly generating first versions of code or scaffolding (e.g., for new features, classes, tests). Frequently cited as useful for "getting started", "proof of concept", or "rough outlines".

3. Documentation & Learning

AI helps developers understand new C++ features, standard library usage, or unfamiliar APIs. Used to **replace or supplement Google/StackOverflow**, often described as "faster" or "more interactive".

4. Understanding Existing Code

AI assists in explaining complex or legacy code, improving codebase comprehension. Useful in onboarding or exploring large codebases.

5. Search Acceleration & Navigation

Used as a **natural language interface** for API lookups, library discovery, or configuration examples. Often described as a better search engine that reduces context-switching and sifting through irrelevant posts.

Secondary Themes (less frequent but notable)

1. Unit Test Generation

Several developers mention using AI specifically for writing or scaffolding unit tests.

2. Debugging Assistance

AI helps explain compiler errors or runtime issues (e.g., C++ template errors, platform-specific issues).

3. Code Review Aid / Brainstorming Partner

Some use AI as a "rubber duck" for idea validation, early feedback, or preliminary code review.

4. Syntax Recall / Code Snippet Generation

Cited as helpful when developers forget how to express a certain construct (e.g., `std::ranges`, `pybind11`, `regex`).

5. Speeding Up Grunt Work

AI helps accelerate low-value work so developers can focus on more complex logic or design.

Honorable Mentions (sporadic, low frequency)

1. Refactoring Support

Some noted AI helps with mechanical refactoring, though trust is limited due to subtle errors.

2. Design & Architecture Exploration

Used to brainstorm or evaluate design approaches in early-stage projects.

3. CI/CD and Build Scripts (e.g., CMake)

Helpful in creating or debugging build configurations.

4. Improved IDE-like Experience

Several developers equate AI with having an advanced assistant in the IDE—especially useful where native IDE support lacks.

5. Language Interoperability

Occasionally cited in tasks involving Python bindings, Lua scripting, or reverse-engineering API interactions.

🚫 Widespread Non-Adoption or Negative Feedback

- **Majority of responses were negative or neutral.**

Many explicitly say "No", "Not at all", or describe AI as **distracting, untrustworthy, or ineffective** for serious C++ work. Common concerns: Incorrect or subtly flawed code. Wasted time reviewing or correcting AI output. Overreliance leading to skill atrophy. Corporate/security restrictions limiting use.

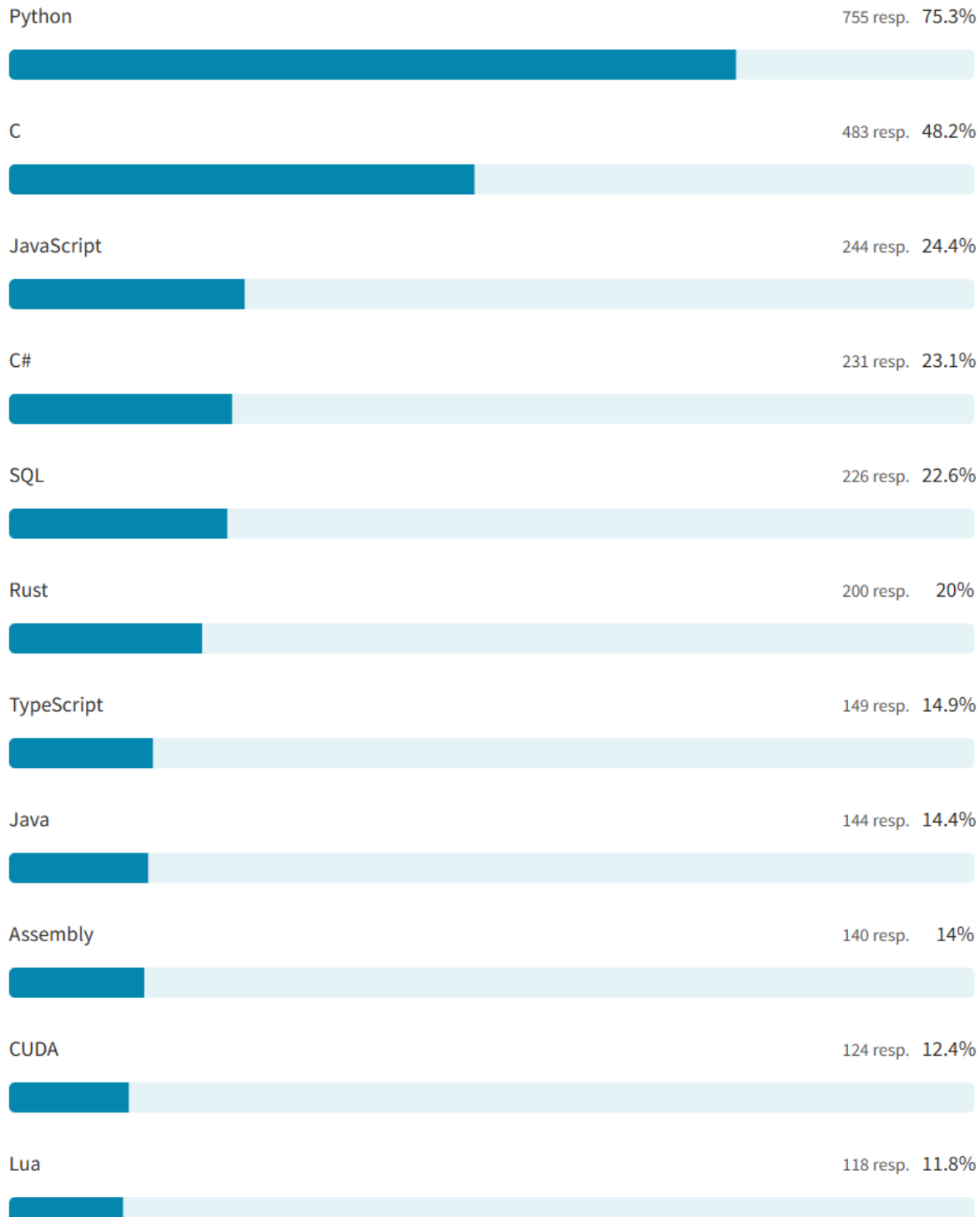
- **AI-generated code is often seen as passable only for trivial tasks**, but not suitable for production-grade C++.

✅ Takeaways for Committee and Product Stakeholders

- **AI excels at scaffolding, not at correctness**—improving tool integration for **safe and debuggable code generation** (e.g., smart templates, compile-time checks) would reduce error-prone outputs.
- **Standardization or libraries that reduce boilerplate** could address the most complained-about C++ pain point.
- Consider **better support in IDEs** (e.g., Copilot-style assistants, context-aware explanations) for both code generation and code comprehension.
- **Improved educational tooling and documentation** (e.g., auto-doc, error explainers) would reinforce AI's perceived strengths.

Besides C++, what programming languages/environments do you use in your current and recent projects?

1002 out of 1036 answered



Node.js 99 resp. 9.9%



Go 94 resp. 9.4%



Objective-C 63 resp. 6.3%



Kotlin 56 resp. 5.6%



OpenMP 54 resp. 5.4%



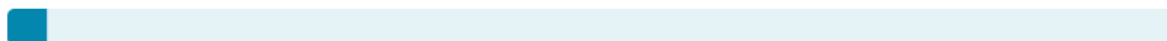
PHP 42 resp. 4.2%



Swift 37 resp. 3.7%



Perl 34 resp. 3.4%



OpenMPI 30 resp. 3%

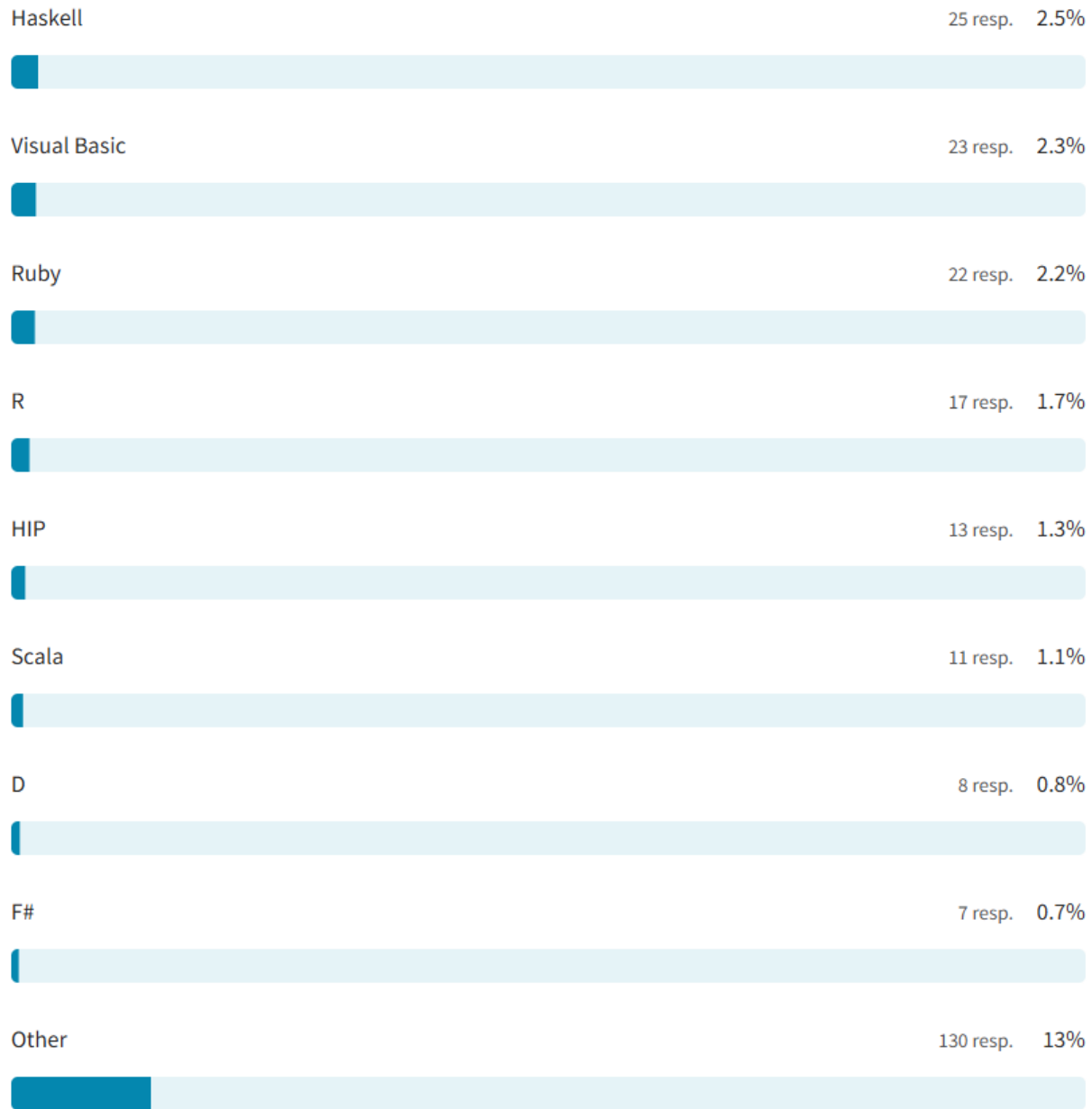


Groovy 27 resp. 2.7%



SYCL 27 resp. 2.7%





Which development environments (IDEs) or editors do you use for C++ development?

731 out of 1036 answered

	Primary	Secondary	Occasional
Android Studio	4.8%	15.4%	79.8%
Atom	13.9%	5.6%	80.6%
CLion	48.8%	18%	33.2%
Code Insight	4.3%	0%	95.7%
CodeLite	6.5%	19.4%	74.2%
Code::blocks	6.2%	18.8%	75%
Eclipse	15.1%	14%	70.9%
Emacs	57%	16.4%	26.6%
IntelliJ IDEA	14.7%	16.2%	69.1%
Kate	13.2%	32.4%	54.4%
KDevelop	9.1%	18.2%	72.7%
Netbeans	10%	3.3%	86.7%
Qt Creator	30.1%	25%	44.9%
Rider	11.8%	25.5%	62.7%
Source Insight	5%	0%	95%
Sublime	19.5%	30.5%	50%

Vim	42.2%	31.6%	26.2%
Visual Studio	57.3%	24.4%	18.3%
Visual Studio Code	54.7%	32%	13.3%
Webstorm	9.7%	16.1%	74.2%
XCode	16.9%	36.3%	46.8%

Which compilers do you use for C++ development?

902 out of 1036 answered

	Primary	Secondary	Occasional
Clang	46.7%	41.3%	12%
GCC	66.5%	26.2%	7.3%
Intel (ICPC, ICPX)	12.5%	25%	62.5%
MSVC	62.6%	21.1%	16.3%
NVIDIA (NVC++, NVCC)	21.4%	37%	41.6%
Custom	33.3%	18.1%	48.6%

What organizations come to your mind the most when you think about C++? Why?

550 out of 1036 people answered this question

[AI-generated summary]

Here is a summary of the most frequently mentioned organizations in the survey responses, in order of frequency. Each summary includes common reasons why the organization was mentioned.

1. ISO C++ Committee (WG21 / ISO / Standard Committee / Standard C++ Foundation / isocpp.org)

The ISO C++ Committee was the most frequently mentioned organization, recognized as the body that defines and evolves the C++ standard. Many respondents respect its crucial role in language design and future direction. However, there is a substantial segment of developers frustrated with the perceived complexity, slow pace, or misguided priorities of the committee. Some view it as disconnected from real-world developer needs, while others appreciate the transparency and documentation of its processes.

2. Microsoft

Microsoft was highly mentioned due to its central role in the C++ ecosystem, primarily through Visual Studio (MSVC), compiler support, and contributions to the standard library. Many developers use their tools daily and appreciate the ongoing involvement in conferences and language evolution. While generally seen positively, there are occasional criticisms about compiler feature lag or legacy code issues, though these concerns were not widely repeated.

3. Google

Google was frequently cited due to its massive C++ code-base, open-source projects like Chromium and Abseil, and previously active contributions to the C++ standards process and tooling (e.g., sanitizers, clang involvement). Many developers follow Google's coding guidelines and use its libraries and tools. Some responses expressed disapproval of the style guide or perceived decline in standards involvement, but overall sentiment was mostly respectful and positive.

4. Bloomberg

Bloomberg is well-known in the community for sponsoring C++ conferences, employing active committee members, and funding language work. Respondents often mentioned its visible support for events like CppCon and contributions to experimental implementations. While some developers appreciate its engagement, others feel Bloomberg's influence on the language is excessive or misaligned with broader developer needs.

5. LLVM / Clang / GCC (LLVM Foundation, Apple, Red Hat, etc.)

LLVM and Clang and GCC were often praised for their tooling, compiler innovation, and influence on modern C++ development. Developers mentioned their role in open-source

compiler advancement and support for cutting-edge C++ features. Many associate Apple and Red Hat with them due to their contributions. Sentiment was overwhelmingly positive regarding the quality and responsiveness of LLVM-based tools.

6. JetBrains

JetBrains, known for its CLion IDE, was frequently acknowledged for high-quality development tools. Users value the productivity gains, strong integration, and regular updates. Its active presence in the C++ community through talks and sponsorships also boosted recognition. Mentions were mostly positive, centered on tooling and usability.

7. Epic Games / Unreal Engine

Epic Games was highlighted for its Unreal Engine, one of the most widely used C++-based game engines. It exemplifies real-world, high-performance C++ usage in AAA game development. Respondents admire how C++ enables complex systems like Unreal Engine, and view Epic as a flagship user organization for C++ in gaming.

8. Boost

Boost was frequently recognized as a source of high-quality, foundational C++ libraries that often influence standardization. Many respondents see Boost as a proving ground for future standard features and value its long-standing role in extending the capabilities of the language. It is seen as a hallmark of good, reusable C++ design.

9. NVIDIA

NVIDIA was noted for its use of C++ in high-performance computing (e.g., CUDA) and driver development. Many developers associate NVIDIA with advanced C++ applications in graphics and AI. The company's visibility in conferences and standards discussions also contributed to its frequent mention.

10. Qt / The Qt Company

Qt was associated with GUI development and large-scale application frameworks in C++. Respondents appreciated the robustness and scope of the Qt libraries. The company is viewed as a key contributor to making C++ more accessible for cross-platform and desktop applications.

11. Yandex

Yandex was repeatedly mentioned, particularly by respondents in or familiar with the Russian tech community. The company is recognized for its open-source projects, high-load systems, and active use of modern C++. Sentiment was positive, emphasizing practical, large-scale deployment

If you could wave a magic wand and change one thing about any part of C++, what would it be, and how would that change help your daily work?

734 out of 1036 people answered this question

[AI-generated summary]

Primary Themes (most frequent and emphasized)

1. Standardized Package Management and Build System

Problem: Users are overwhelmingly frustrated with inconsistent, complex, and fragmented tooling.

Proposed Change: Introduce a **standard package manager** and **build system** akin to Rust's Cargo or Python's pip.

Impact: Streamlines dependency management, reduces onboarding friction, simplifies cross-platform builds.

2. Break Backward Compatibility (Introduce "Epochs" or Editions)

Problem: Legacy features, bad defaults, and ABI compatibility block modern improvements.

Proposed Change: Introduce **epochs/editions** to isolate old code and allow cleaner new code.

Impact: Enables safer defaults, memory safety, simpler syntax, and removal of legacy complexity.

3. Reflection and Introspection

Problem: Manual boilerplate and lack of runtime/compile-time metadata.

Proposed Change: Add **native reflection and code generation capabilities**.

Impact: Reduces boilerplate, enables safer and more efficient tooling and data processing.

4. Memory Safety and Better Defaults

Problem: C++ is too error-prone; unsafe defaults (e.g., raw pointers, mutable by default).

Proposed Change: Default to **const**, **noexcept**, **nodiscard**, memory-safe constructs; introduce **ownership/lifetime models** similar to Rust.

Impact: Improves safety and reliability, especially for critical or large-scale codebases.

5. Fix Compilation Times

Problem: Long build cycles reduce productivity.

Proposed Change: Fully working **modules**, simpler header model, template optimization.

Impact: Enables faster iteration and debugging, improves developer experience.

6. Improve Compiler and Tool Error Messages

Problem: Error messages, especially with templates and lambdas, are unreadable.

Proposed Change: Adopt **Rust-style diagnostic messages**, better context and fix suggestions.

Impact: Reduces learning curve and debugging time.

7. Remove Implicit Conversions and Undefined Behavior (UB)

Problem: Silent bugs and confusion from implicit type casts and undefined behavior.

Proposed Change: Make conversions and UB **explicit and opt-in**.

Impact: Prevents subtle bugs, aids static analysis and tool diagnostics.

Secondary Themes (less frequent but still prominent)

1. Simplify the Language and Syntax

Concerns about verbosity, complexity, and inconsistent grammar.

Desire for Python-like simplicity or Rust-style clarity.

2. Safe/Restrictive Subset or Mode

A stricter "safe C++" mode or dialect to limit dangerous features and encourage best practices.

3. Reform Template and Metaprogramming Model

Make templates easier to read/write/debug; possibly replace with a new compile-time meta language.

4. Better Standard Library and Feature Availability

Add modern facilities (networking, strings, ADTs, better containers); ship implementations sooner.

5. Better Defaults for Language Constructs

Flip defaults to safer, more modern choices (e.g., **const** by default, explicit constructors, no raw pointers).

6. Improve Module System and Adoption

Make modules simpler, truly cross-platform, and integrate with toolchains and editors.

7. Fix Legacy APIs and C Compatibility

Remove or isolate C-style constructs (e.g., raw pointers, **goto**, **malloc**) into unsafe blocks or dialects.

Honorable Mentions (not common, but notable insights)

- **Better Debugging Support**
Language constructs or tools to improve observability and introspection during debugging.
- **Runtime Features like Fibers or Static Interfaces**
Desire for language-native fibers or static interface mechanisms (e.g., Rust traits).
- **Better Documentation and Learning Resources**
Simpler official documentation; clearer guidance, especially for industry newcomers.
- **Integrated Testing Support**
Built-in testing framework or annotations similar to Rust or Python.
- **Governance Reform**
Frustration with ISO and the slow, conservative pace of standardization; suggestions for more agile or open models.

☒ Takeaways for Committee and Product Stakeholders

There is strong consensus around modernizing the **tool-chain**, improving **safety and usability**, and embracing **breaking changes** where necessary. Standardized **package/build systems**, memory safety mechanisms, and reflection are the clearest high-impact opportunities. Implementing these would directly address user pain points and significantly enhance daily productivity, maintainability, and adoption of C++.

(optional) Finally, in the past year what things in the C++ world (in standards, projects, conferences, articles, etc.) ...

... are you most excited about / make you feel positive about C++ and its future? Why?

616 out of 1036 people answered this question

[AI-generated summary]

Primary Themes (Most Frequent and Strongly Emphasized)

1. Reflection (especially static/compile-time reflection in C++26)

Why it matters: Seen as a game-changer; enables removal of boilerplate, improves metaprogramming, allows safe introspection, facilitates serialization/codgen, and modernizes idioms.

Actionable insight: Ensure robust design and wide applicability of reflection. Deliver with good ergonomics. Support tooling and libraries that leverage it.

2. Memory Safety and Safety Profiles

Why it matters: Strongly positive reception to the standard's increasing focus on safety: reducing UB, adding safer defaults, contracts, and introducing opt-in profiles.

Actionable insight: Accelerate standardization of safety profiles and safety-enhancing features (e.g., contracts, safe modes). Demonstrate tangible safety benefits clearly to industry.

3. Modules

Why it matters: Highly anticipated due to promises of better build times, clearer interfaces, and modern project organization.

Actionable insight: Encourage full compiler and tooling support. Promote guidance and tooling for integrating modules into large legacy codebases.

Secondary Themes (Noted Frequently but with Slightly Less Emphasis)

4. Concepts and Constraints

Why it matters: Aids readability, improves error messages, enhances template expressiveness.

Actionable insight: Improve documentation and learning materials. Showcase real-world use cases where concepts improve code clarity and safety.

5. Contracts

Why it matters: Many see contracts as overdue. They're viewed as essential for correctness, documentation, and runtime safety.

Actionable insight: Expedite reintroduction into the standard. Ensure good runtime/compile-time flexibility and tooling integration.

6. Coroutines, Executors, and `std::execution`

Why it matters: Key to writing safer and more scalable async/concurrent code. Interest is growing with related libraries (senders/receivers).

Actionable insight: Prioritize standardization and real-world examples. Improve interoperability with existing async paradigms.

7. General Language Evolution and `constexpr` Expansion

Why it matters: Users appreciate the pace and direction of evolution, especially “constexpr everything”, better syntax, and STL improvements.

Actionable insight: Continue broad `constexpr` adoption. Communicate rationale behind language complexity trade-offs more transparently.

8. Tooling, Education, and Community Engagement

Why it matters: Vibrant community, educational content, and conferences maintain engagement and adoption.

Actionable insight: Support tool creators, maintain momentum with educational initiatives, and ensure committee proposals are accessible to broader audiences.

Honorable Mentions (Smaller but Notable Clusters)

- **Cppfront / Circle / Experimental Frontends**
Interest in exploring radically improved syntax while preserving interop.
- **Rust Influence and C++/Rust Interop**
Rust is often admired for safety, clarity, and ecosystem; some hope C++ learns from it or interoperates better.
- **Package and Build System Improvements**
Calls for standardized and universal solutions (e.g., CPM, Conan, vcpkg, Common Package Spec).
- **Ranges, Views, and Functional Style**
Enthusiasm for a more expressive and composable standard library.
- **Networking and Parallelism**
Interest in more high-level support for structured concurrency and heterogeneous compute.

☑ Takeaways for Committee and Product Stakeholders

- **Top Priority:** Deliver static reflection and safety features effectively; these are overwhelmingly the most anticipated changes.
- **Keep Momentum:** Concepts, modules, and contracts are seen as foundational; continue pushing them to maturity.
- **Show Value Clearly:** Users want tangible improvements in day-to-day code quality, safety, and ergonomics—not just abstract power.
- **Toolchain Coordination is Critical:** Many users are frustrated by delayed compiler support; standard features must be matched with ecosystem readiness.
- **Acknowledge Frustrations:** A significant minority feel C++ is stagnating or too complex—counteract this with better defaults, tooling, and simplification where possible.

(optional) Finally, in the past year what things in the C++ world (in standards, projects, conferences, articles, etc.) ...

... are you most concerned about / make you feel negative about C++ and its future? Why?

636 out of 1036 people answered this question

[AI-generated summary]

Primary Themes (Most Frequent and Emphasized)

1. Excessive Language Complexity

Why it matters: The increasing complexity—both syntactic and conceptual—makes C++ harder to learn, teach, and maintain. It discourages newcomers and makes experienced developers feel overwhelmed.

Actionable insight: Prioritize simplification and refactoring over adding new features. Introduce opt-in epochs or editions with simplified, modern defaults and deprecate legacy constructs.

2. Lack of Progress on Memory Safety

Why it matters: Memory safety concerns are driving adoption of Rust and contributing to negative press, regulatory warnings, and organizational mandates to move away from C++. The current "profiles" approach is widely viewed as insufficient or flawed.

Actionable insight: Adopt a clear, cohesive memory safety roadmap. Support safer language subsets, integrate static analysis tooling (like clang-tidy) into the compiler, and reevaluate or revive proposals like Safe C++ with transparent technical discussion.

3. Backwards Compatibility Preventing Progress

Why it matters: An insistence on ABI and source compatibility prevents fixing long-standing language and library issues, adding friction to evolution and modernization efforts.

Actionable insight: Define and implement a strategy to allow controlled, opt-in ABI breaks (e.g., epochs, build modes), and begin actively deprecating and phasing out harmful legacy features.

4. Slow Standardization and Toolchain Lag

Why it matters: Major features (e.g., modules, coroutines, reflection) take too long to reach production-ready compilers and tools. This erodes trust in the committee and increases reliance on better-working third-party solutions.

Actionable insight: Improve collaboration with compiler vendors, require working prototype implementations before standardization, and focus on fewer but higher-quality features per release cycle.

5. Poor Tooling and Missing Ecosystem Infrastructure

Why it matters: The absence of a standard package manager, project description format, or build tool makes C++

difficult to adopt in modern development workflows compared to Rust (Cargo) or Python (pip).

Actionable insight: Officially standardize (or endorse) a complete project/package ecosystem—e.g., unify CMake and Conan or adopt a single standard toolchain interface akin to Cargo.

Secondary Themes (Frequent but Slightly Less Emphasized)

6. Reputation and External Perception Issues

Why it matters: Government advisories, media narratives, and industry influencers increasingly label C++ as unsafe or outdated, leading to reduced adoption and hiring hesitancy.

Actionable insight: Promote success stories of modern, safe C++ usage. Provide official safety guidance, clear communication around profiles, and transparent rebuttals to misleading narratives.

7. Unusable or Overengineered Language Features

Why it matters: Features like coroutines, modules, concepts, and senders/receivers are often considered too complex, poorly documented, or hard to use in practice.

Actionable insight: Design with usability in mind. Require implementation experience and user feedback before standardization. Provide canonical usage examples and training material alongside new features.

8. Failure to Fix Broken or Incomplete Existing Features

Why it matters: Contracts, modules, and even core standard library components (e.g., `std::vector`, `std::regex`) are seen as broken or half-baked, with no improvement path.

Actionable insight: Allocate committee time specifically to reviewing and fixing existing flaws. Consider a "bugfix" release focused on stabilization, not feature expansion.

9. Disconnect Between Committee and Developers

Why it matters: Many developers perceive the standards committee as insular, opaque, and more focused on niche use cases than the needs of working developers.

Actionable insight: Improve transparency, seek broader community input (especially from industry), and introduce moderation or public representation roles in the standards process.

Honorable Mentions (Notable but Less Widespread)

- **Reflection and Pattern Matching Delays**

Why it matters: Many developers were counting on these for modern code generation and cleaner syntax.

Actionable insight: Accelerate delivery and ensure minimal syntax overhead for core reflective and pattern features.

- **Loss of Language Integrity / Coherent Vision**

Why it matters: C++ is seen as becoming a patchwork of inconsistent, overlapping features with no cohesive direction.

Actionable insight: Establish and publish a long-term design vision for the language, including clear priorities and compatibility policies.

- **Concerns about Community Culture and Governance**

Why it matters: Community toxicity, gatekeeping, and political drama damage participation and morale.

Actionable insight: Adopt formal codes of conduct, address community controversies transparently, and ensure inclusive participation pathways.

- **Competition from Rust and Other Modern Languages**

Why it matters: C++ is losing mindshare among students, startups, and developers who prefer safer or easier languages.

Actionable insight: Learn from and adapt strengths of these languages (e.g., borrow checking, tooling, packaging), while preserving C++'s advantages.

☒ Takeaways for Committee and Product Stakeholders

The majority of C++ developers who responded are not abandoning the language—they're frustrated by stagnation, complexity, and lack of practical progress. They are asking for **simplification, safety, modern tooling**, and a **more responsive, focused standards process**. Addressing these themes could significantly boost trust, adoption, and productivity across the ecosystem.