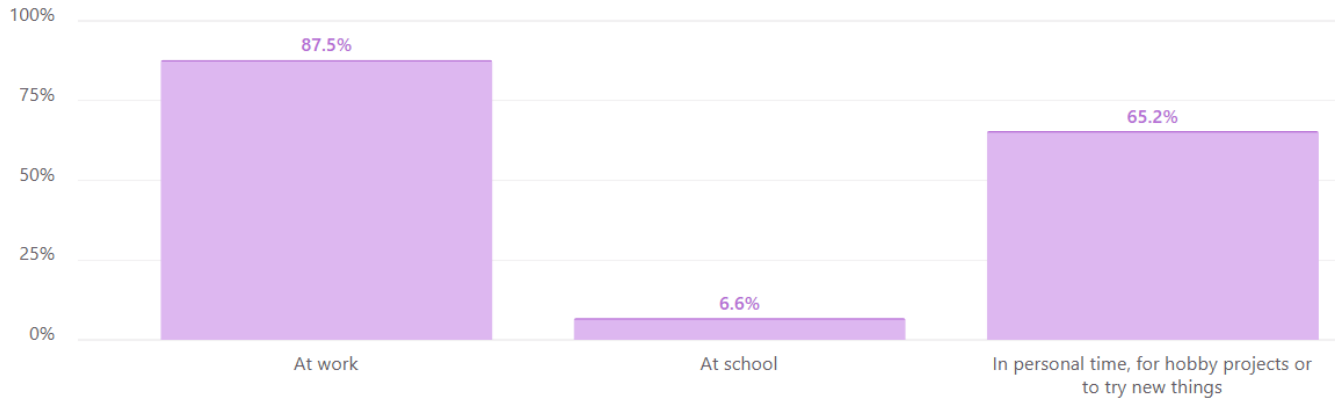
1 2026 Annual C++ Developer Survey "Lite"

1a Where do you use C++?

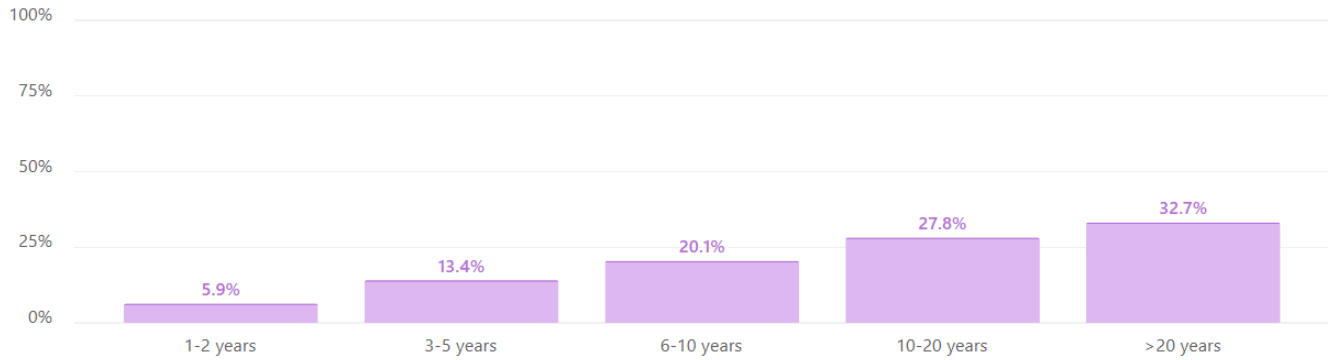
1428 out of 1434 people answered this question.



2 Your experience

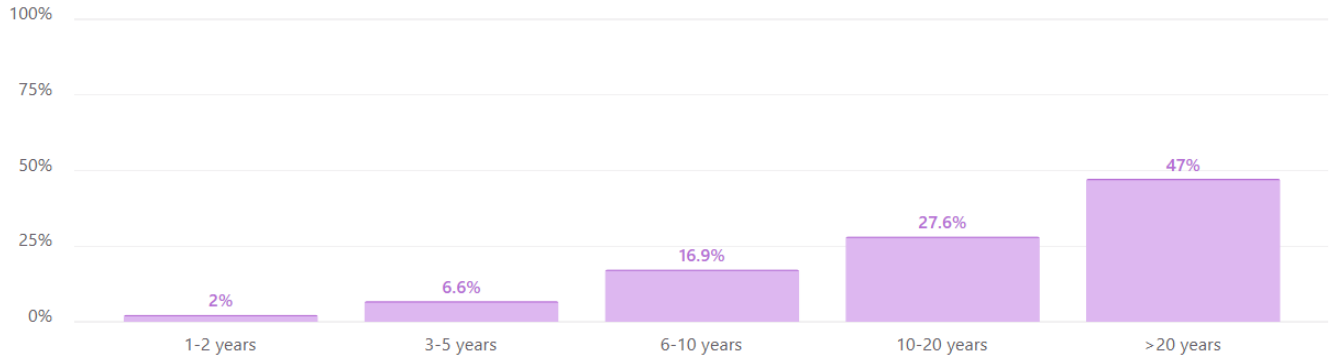
2a How many years of programming experience do you have in C++ specifically?

1421 out of 1434 people answered this question.



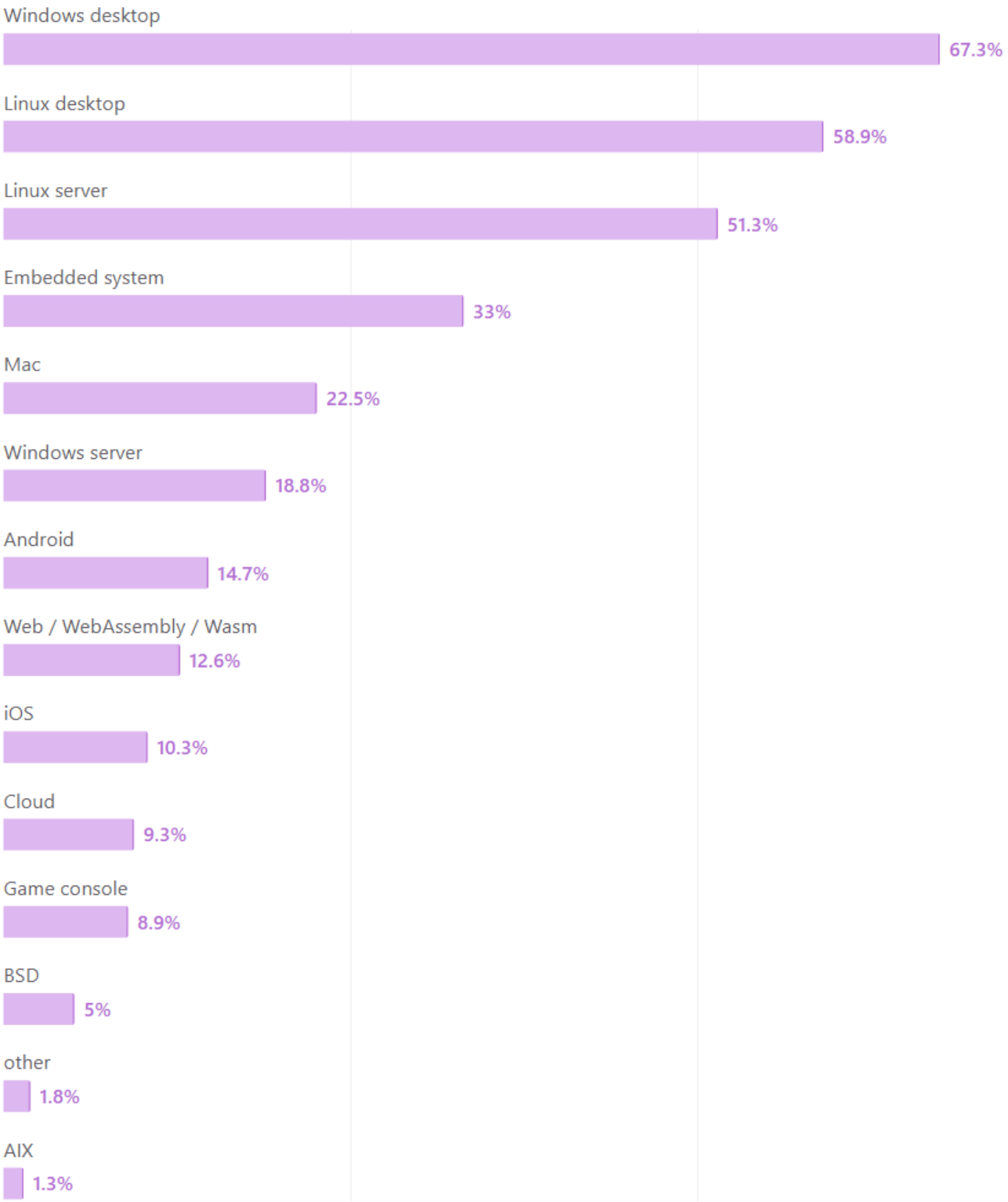
2b How many years of programming experience do you have overall (all languages)?

1277 out of 1434 people answered this question.



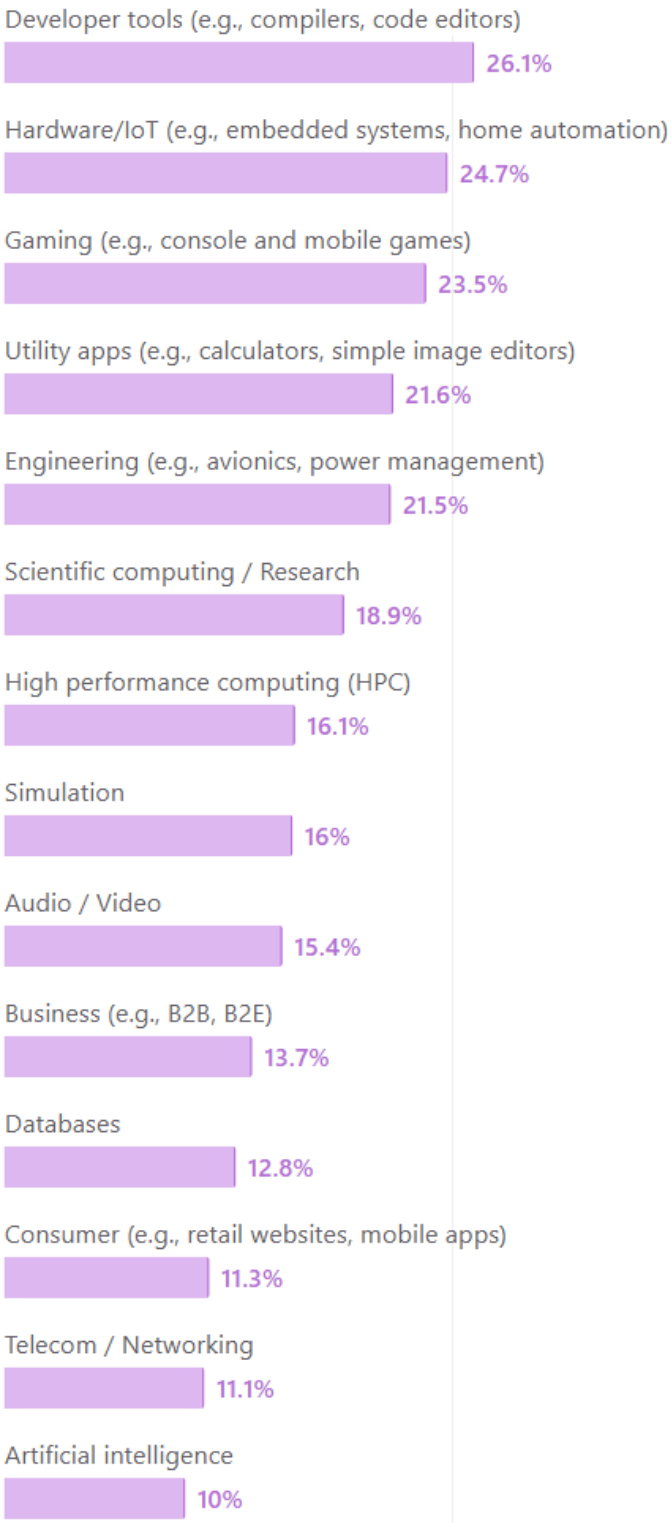
3 What platforms do you develop for?

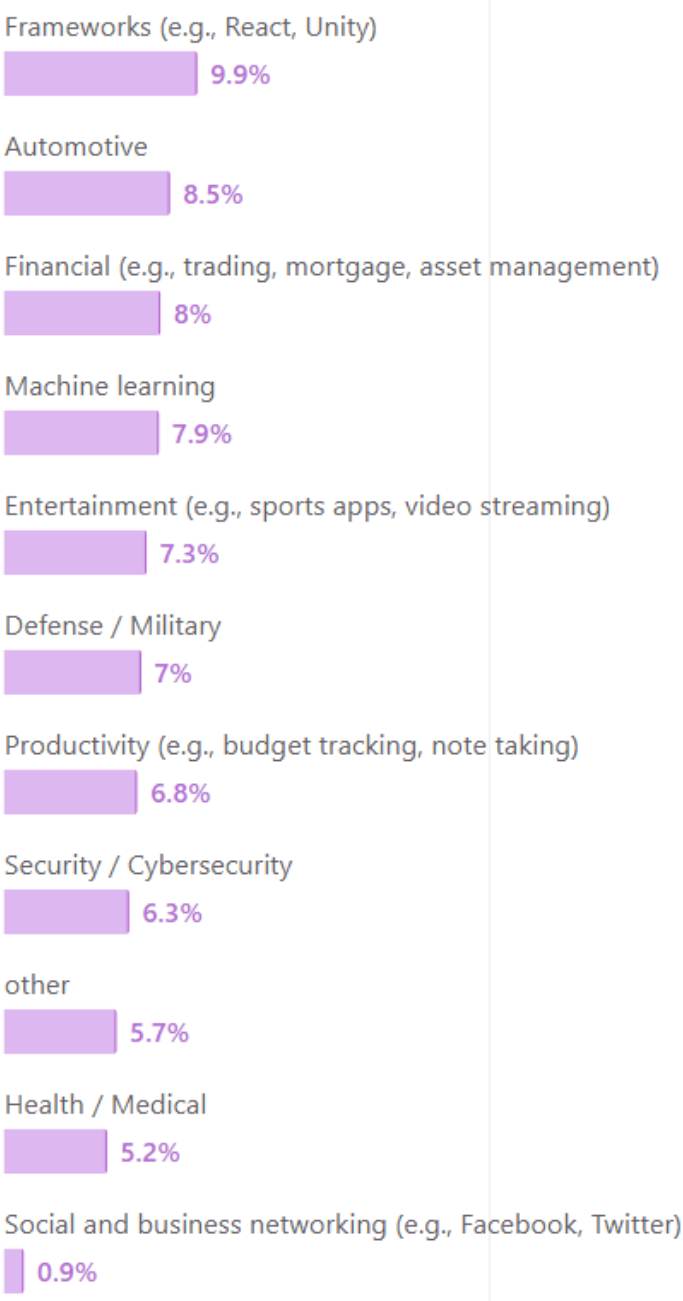
1420 out of 1434 people answered this question.



4 What types of projects do you work on?

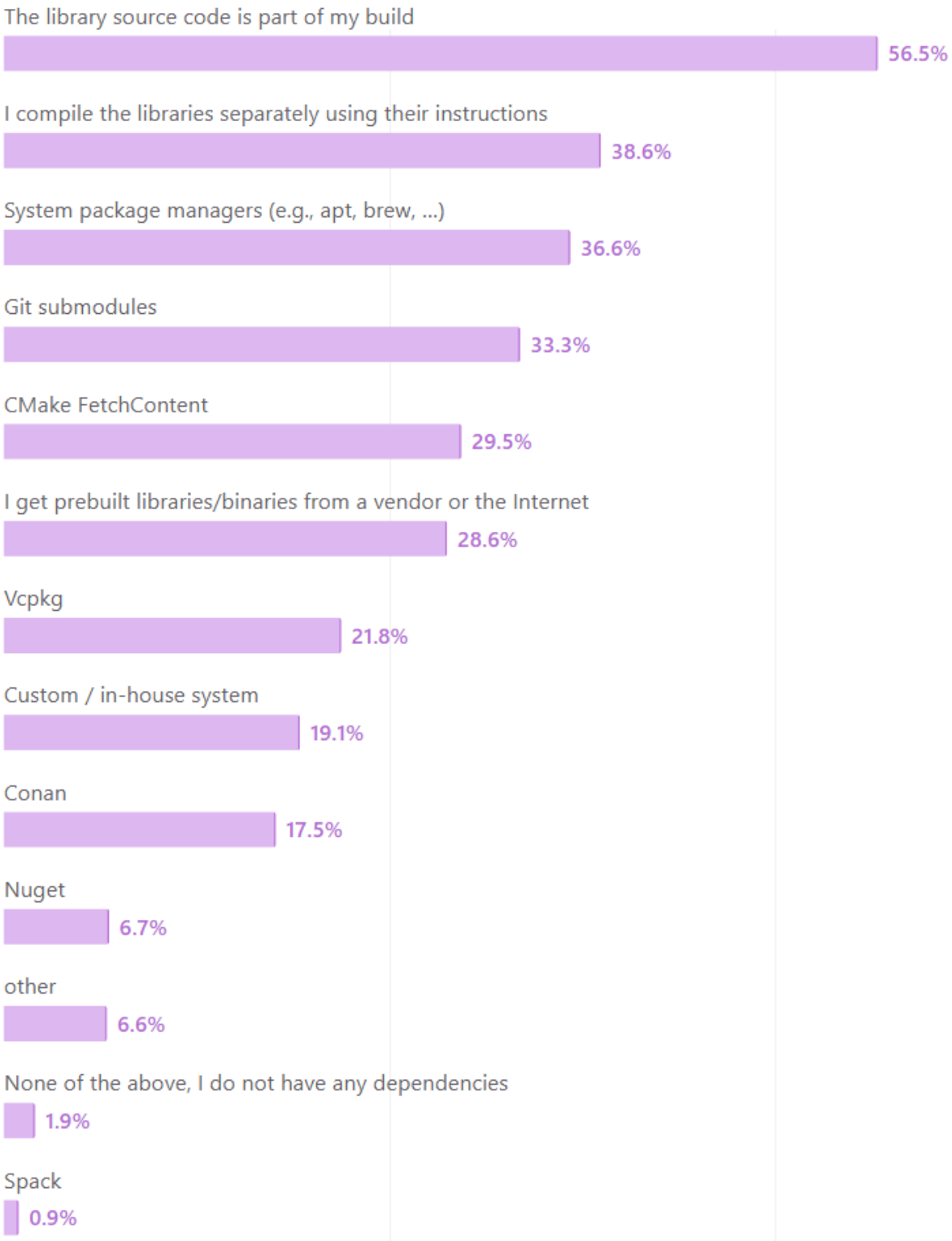
1420 out of 1434 people answered this question.




Frameworks (e.g., React, Unity)
9.9%
Automotive
8.5%
Financial (e.g., trading, mortgage, asset management)
8%
Machine learning
7.9%
Entertainment (e.g., sports apps, video streaming)
7.3%
Defense / Military
7%
Productivity (e.g., budget tracking, note taking)
6.8%
Security / Cybersecurity
6.3%
other
5.7%
Health / Medical
5.2%
Social and business networking (e.g., Facebook, Twitter)
0.9%

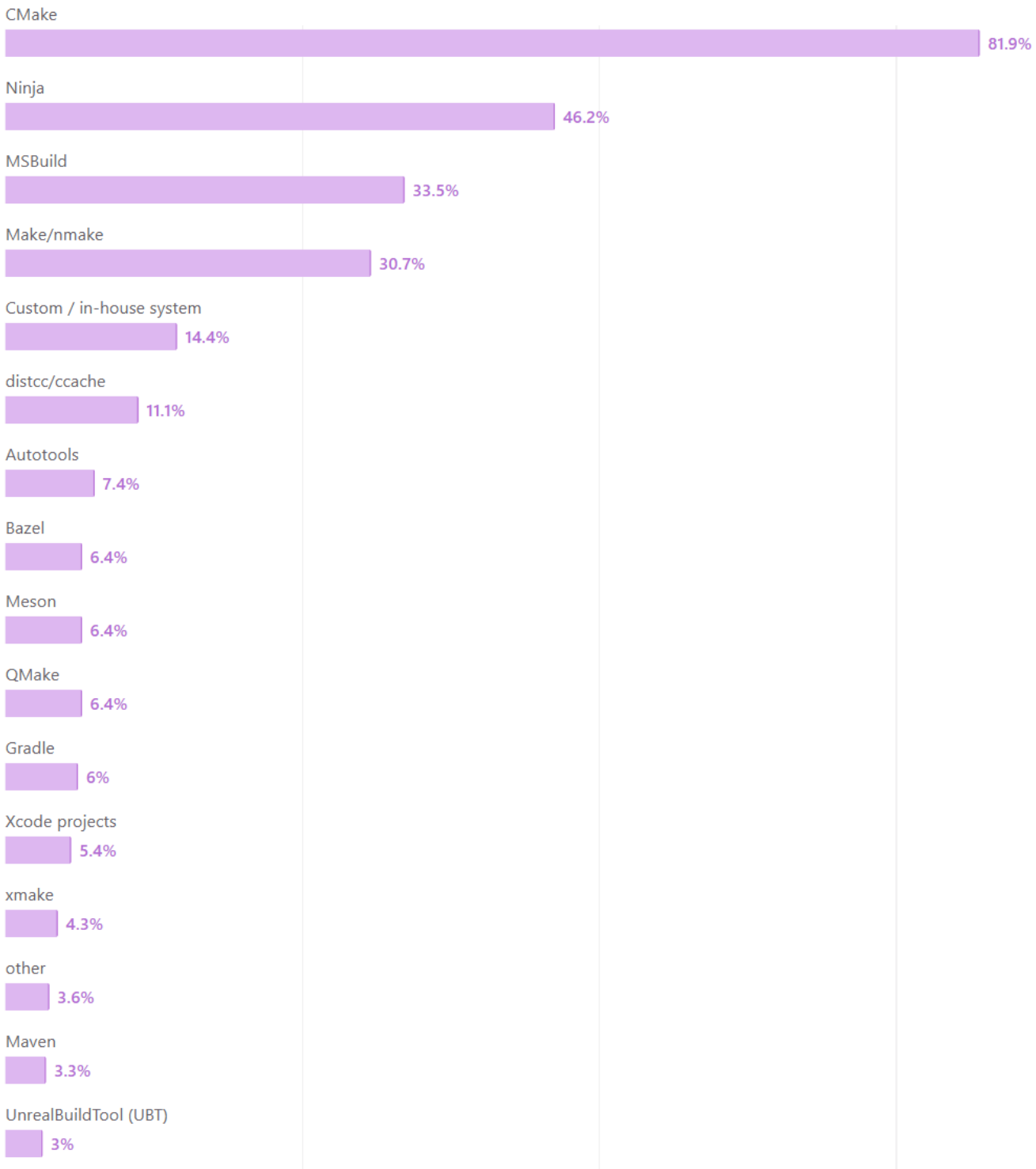
5 How do you manage your C++ 1st and 3rd party libraries?

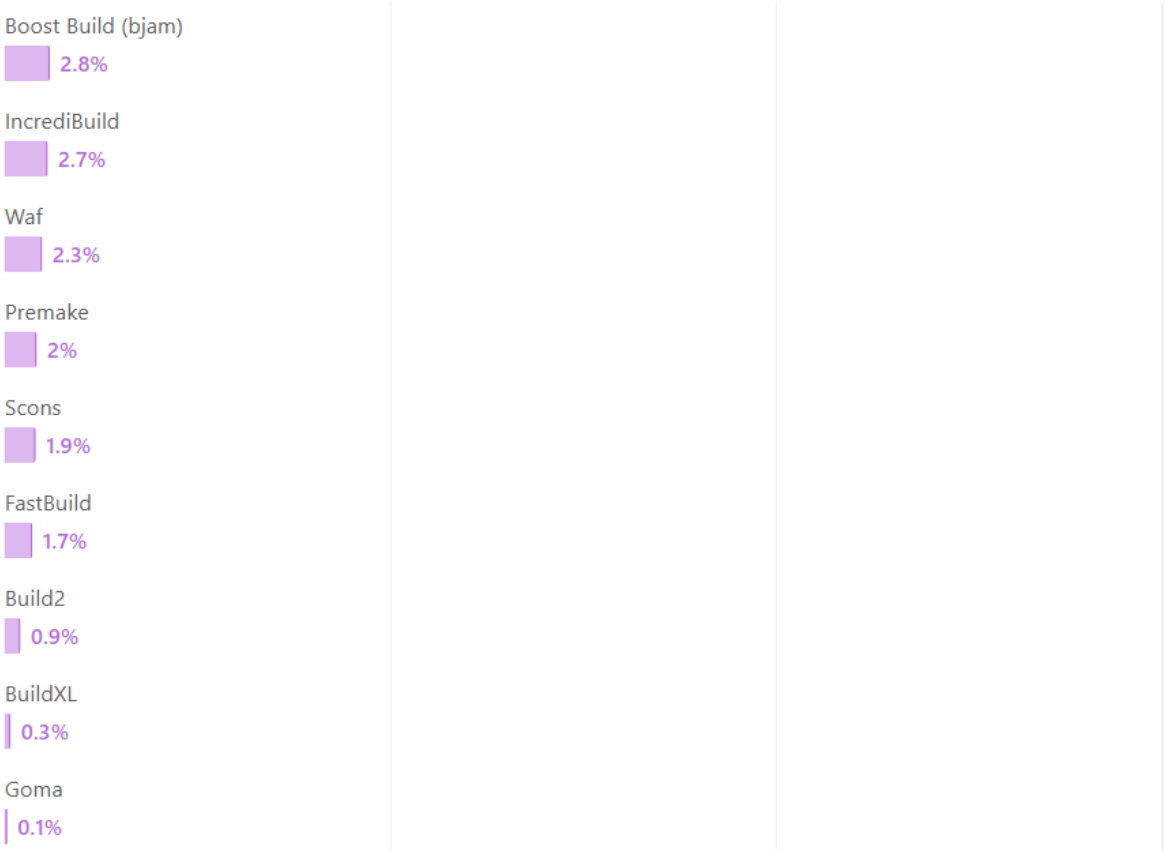
1416 out of 1434 people answered this question.



6 What build tools do you use?

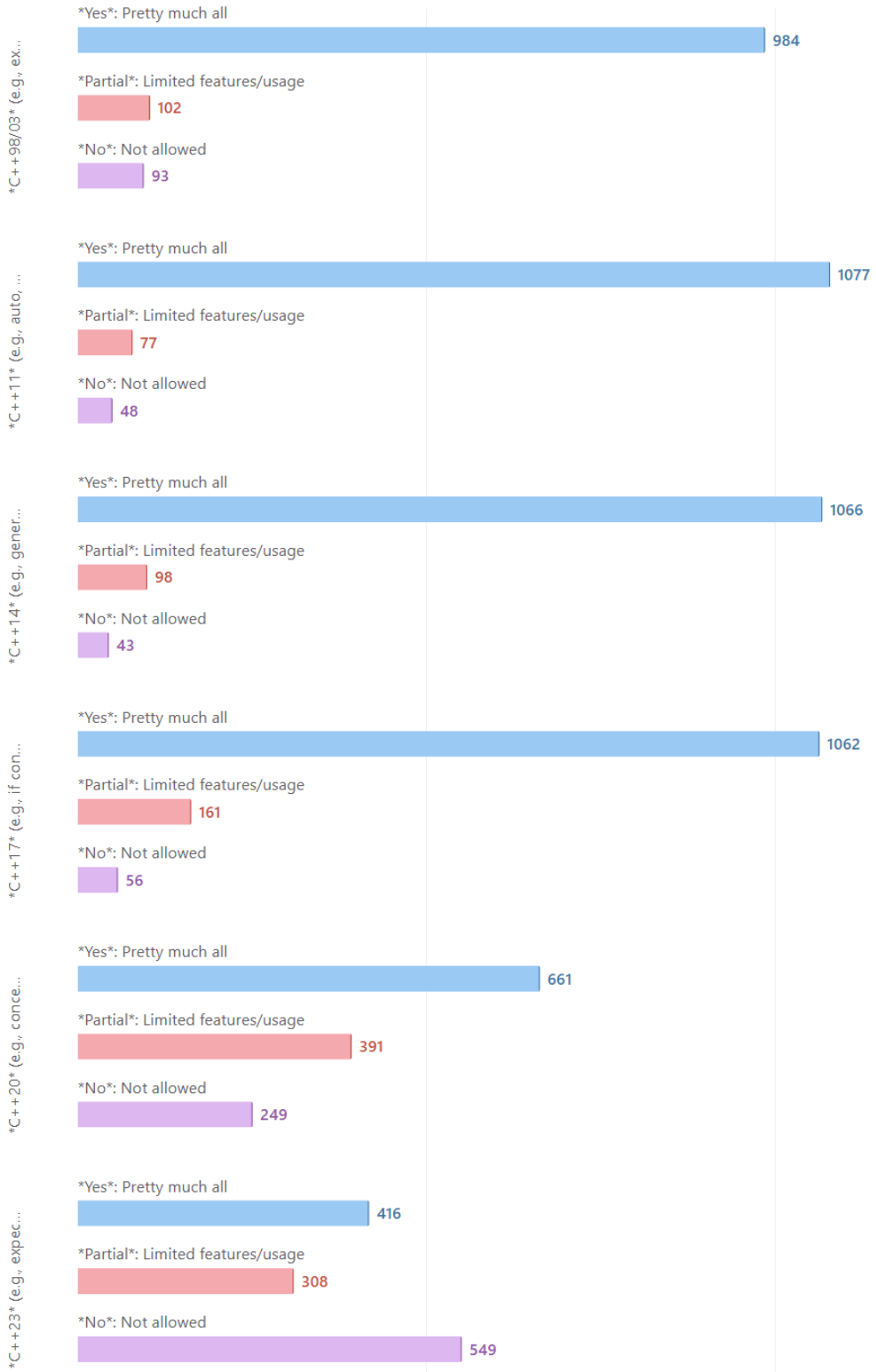
1415 out of 1434 people answered this question.




Boost Build (bjam)
2.8%
IncrediBuild
2.7%
Waf
2.3%
Premake
2%
Scons
1.9%
FastBuild
1.7%
Build2
0.9%
BuildXL
0.3%
Goma
0.1%

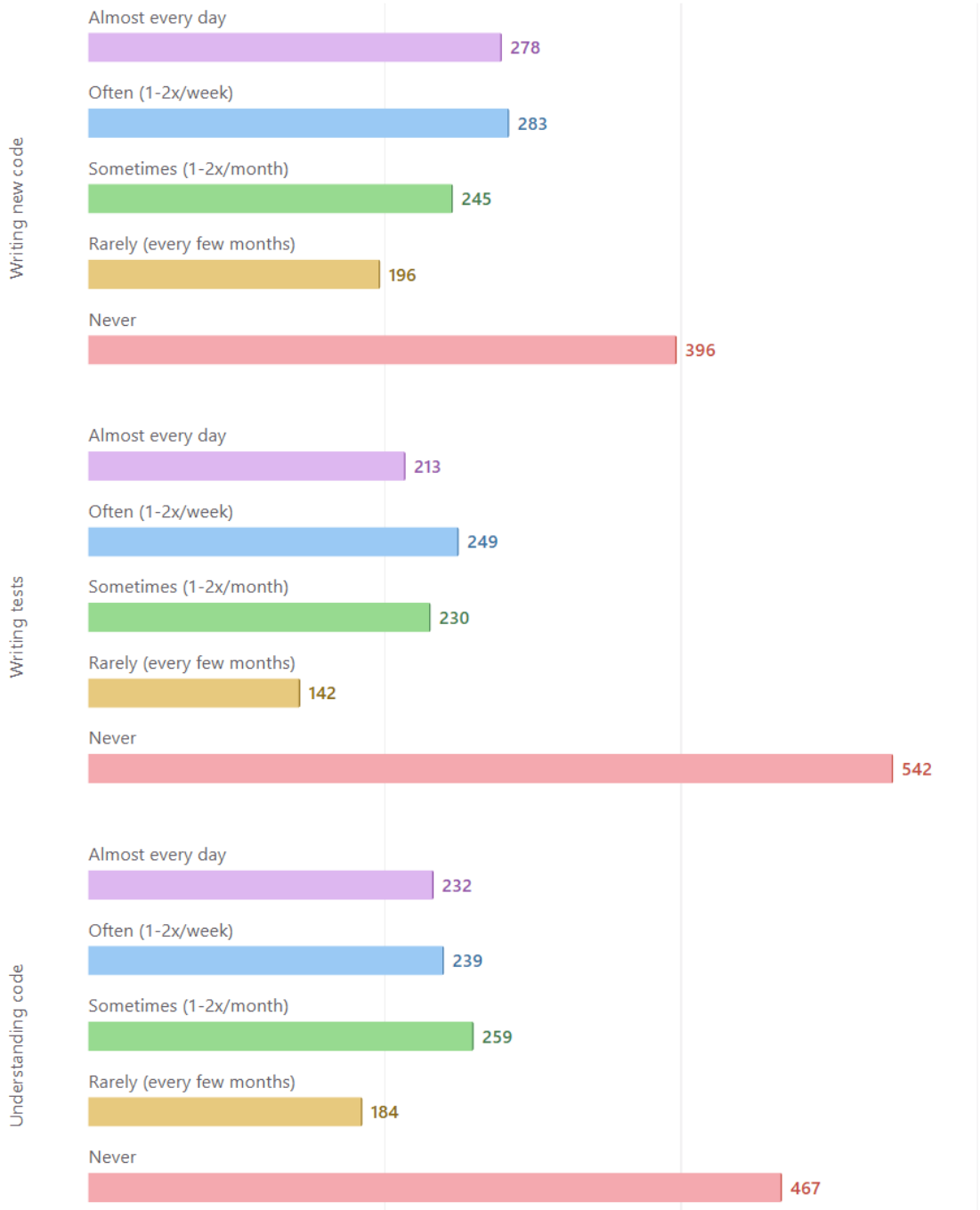
7 What version(s) of C++ are you allowed to use on your current project (work or school)?

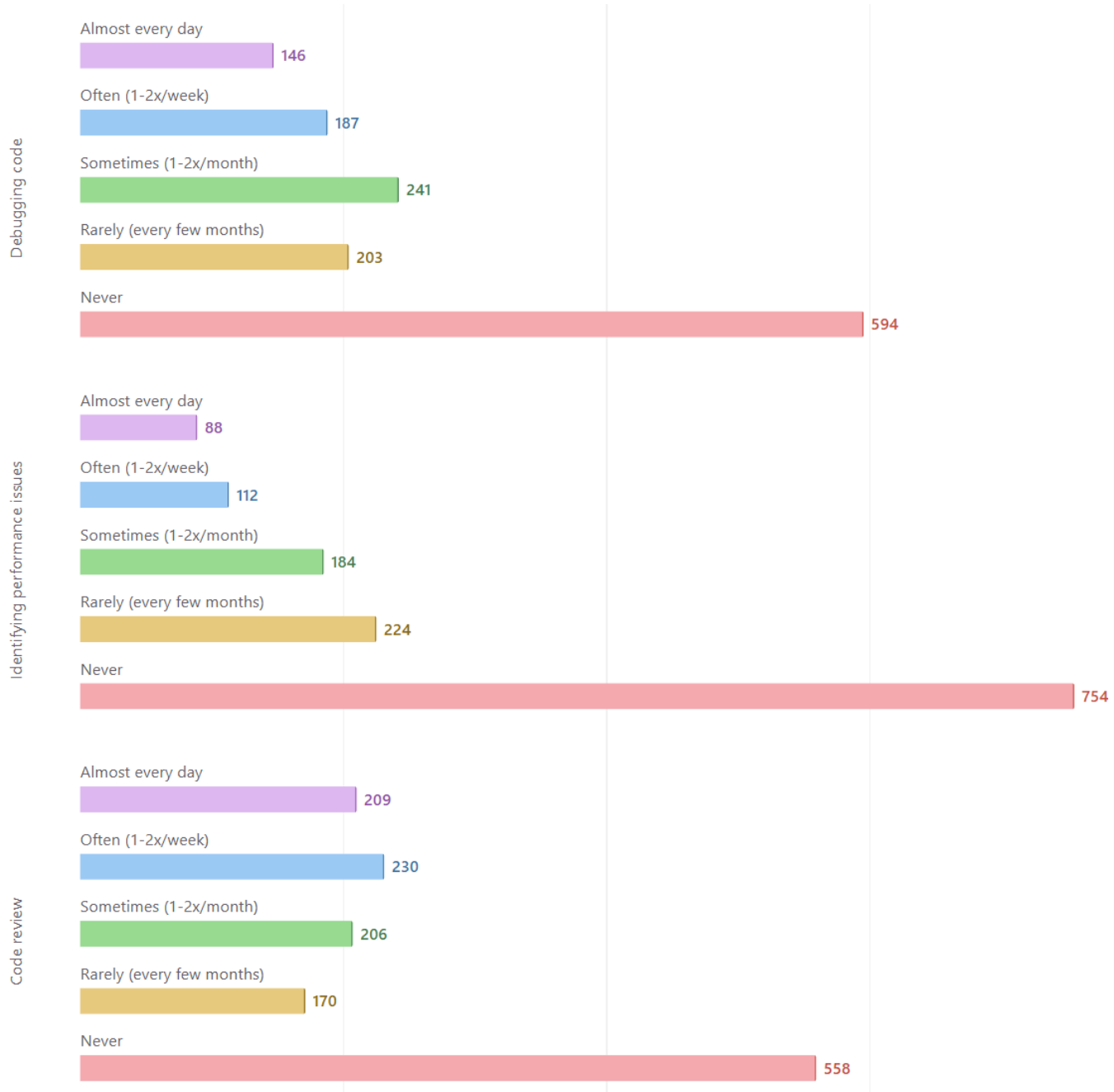
1412 out of 1434 people answered this question.

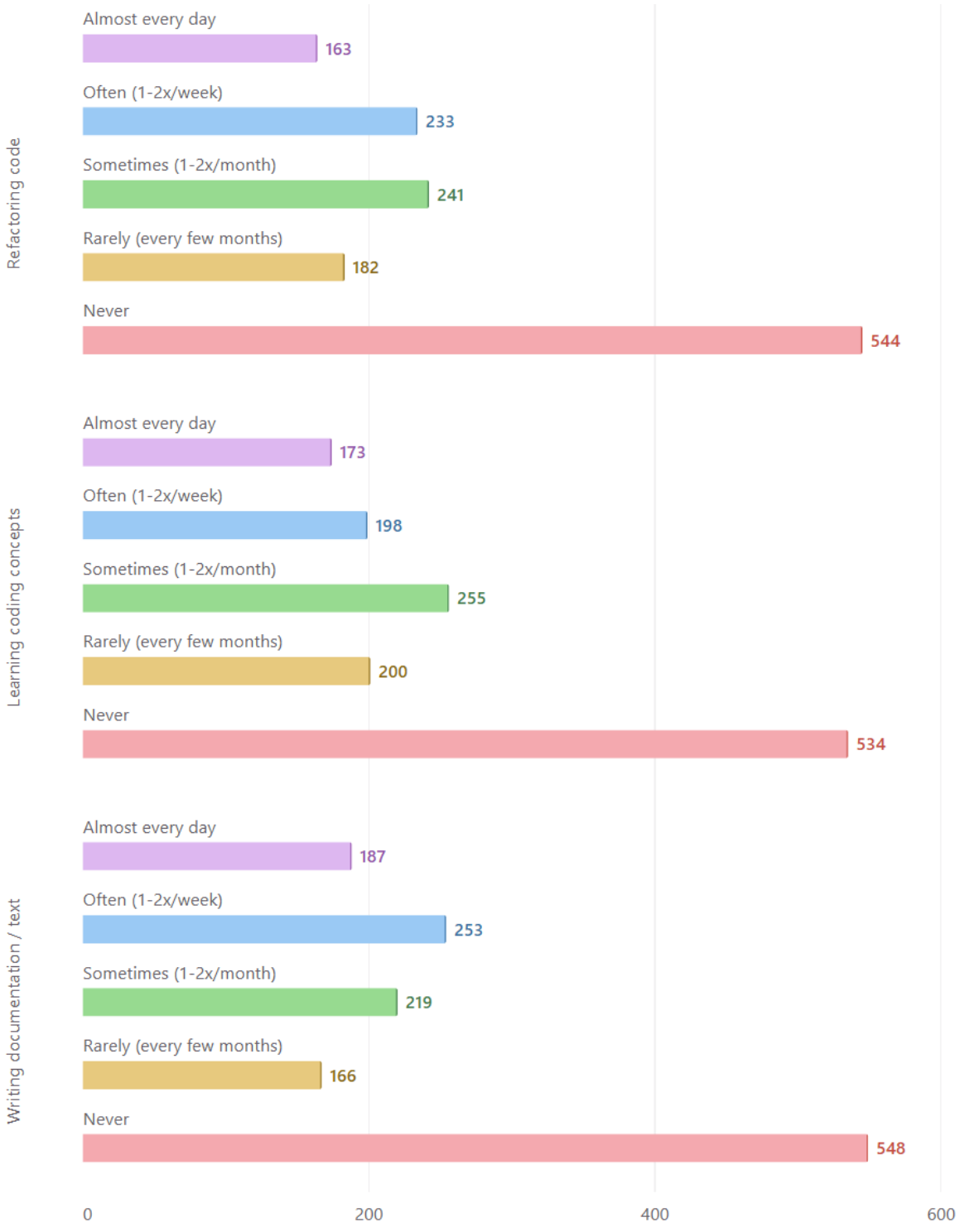


8 How often have you utilized AI assistants for the following C++ coding tasks, if at all?

1408 out of 1434 people answered this question.

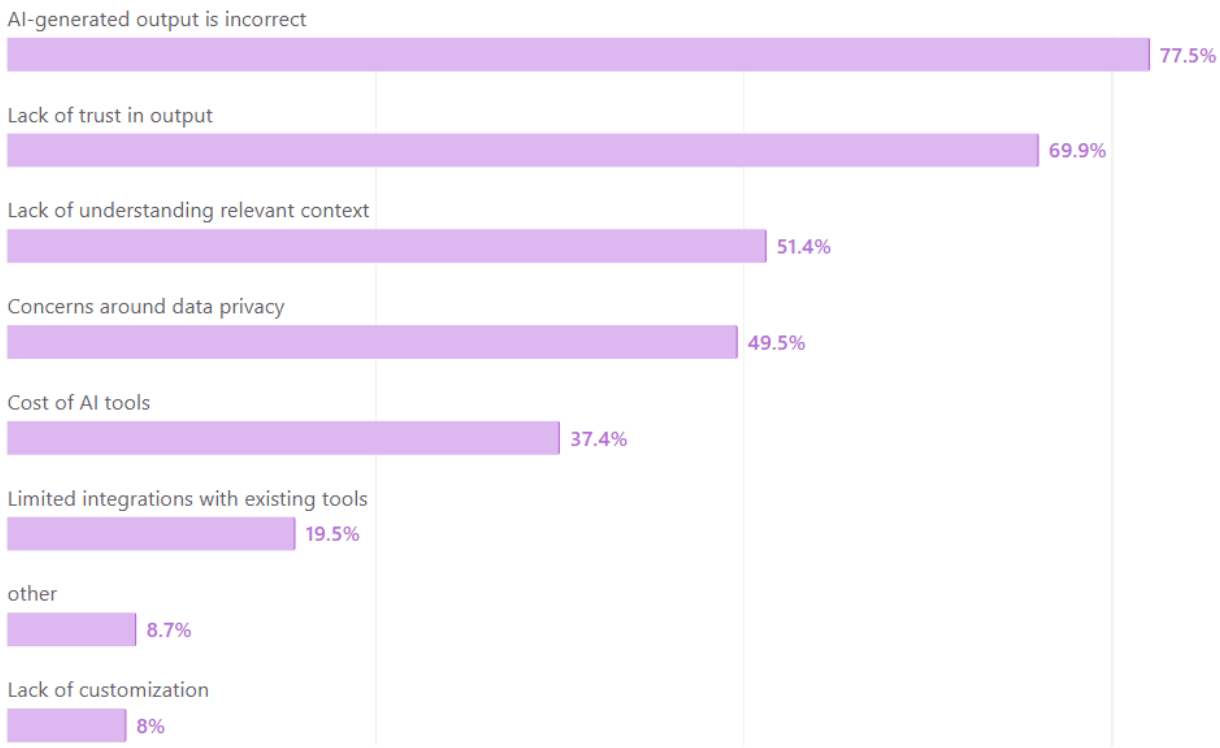



Debugging code
Almost every day
146
Often (1-2x/week)
187
Sometimes (1-2x/month)
241
Rarely (every few months)
203
Never
594
Identifying performance issues
Almost every day
88
Often (1-2x/week)
112
Sometimes (1-2x/month)
184
Rarely (every few months)
224
Never
754
Code review
Almost every day
209
Often (1-2x/week)
230
Sometimes (1-2x/month)
206
Rarely (every few months)
170
Never
558


Refactoring code
Almost every day
163
Often (1-2x/week)
233
Sometimes (1-2x/month)
241
Rarely (every few months)
182
Never
544
Learning coding concepts
Almost every day
173
Often (1-2x/week)
198
Sometimes (1-2x/month)
255
Rarely (every few months)
200
Never
534
Writing documentation / text
Almost every day
187
Often (1-2x/week)
253
Sometimes (1-2x/month)
219
Rarely (every few months)
166
Never
548
0
200
400
600

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

1326 out of 1434 people answered this question.



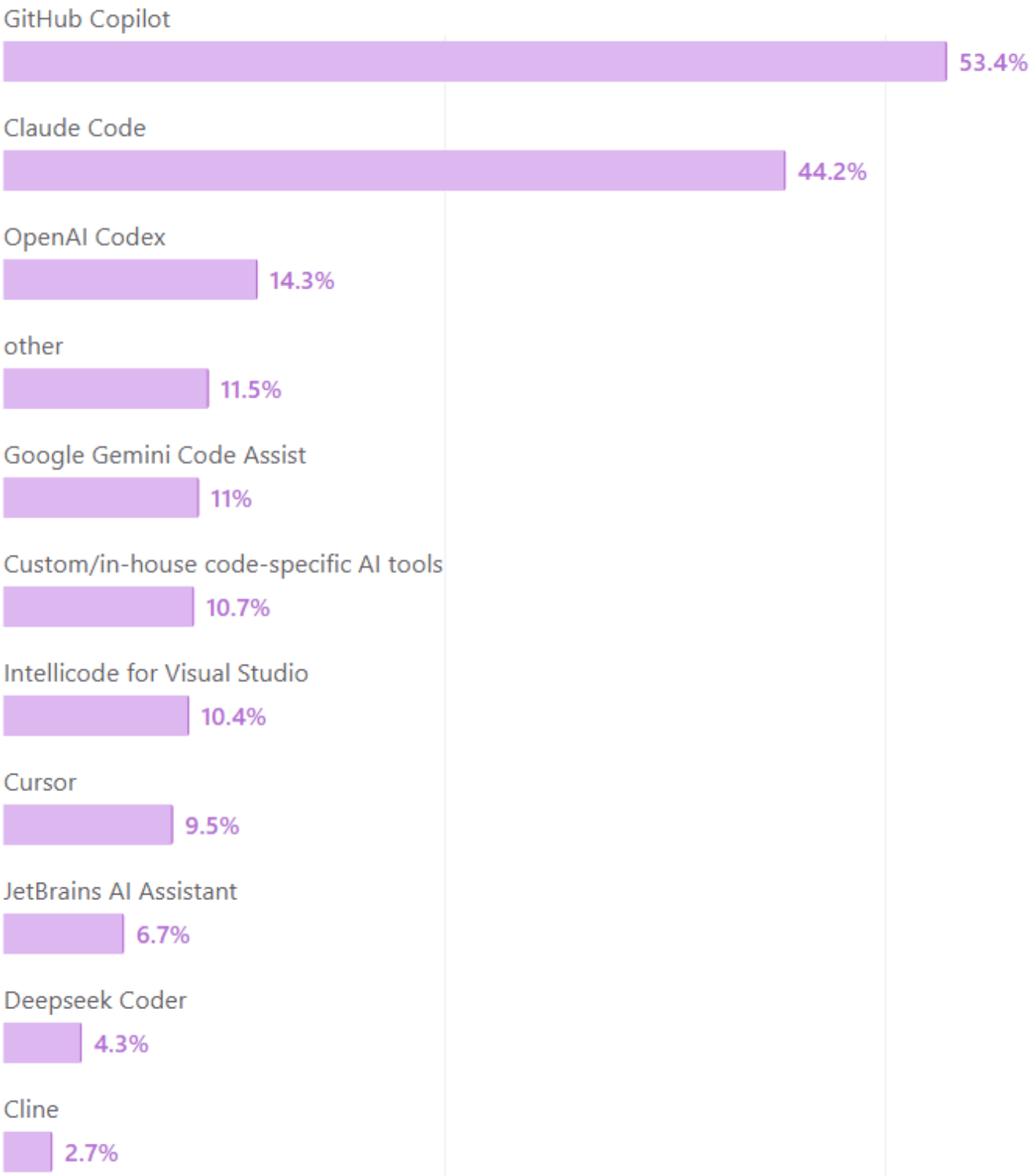
[AI-generated summary]

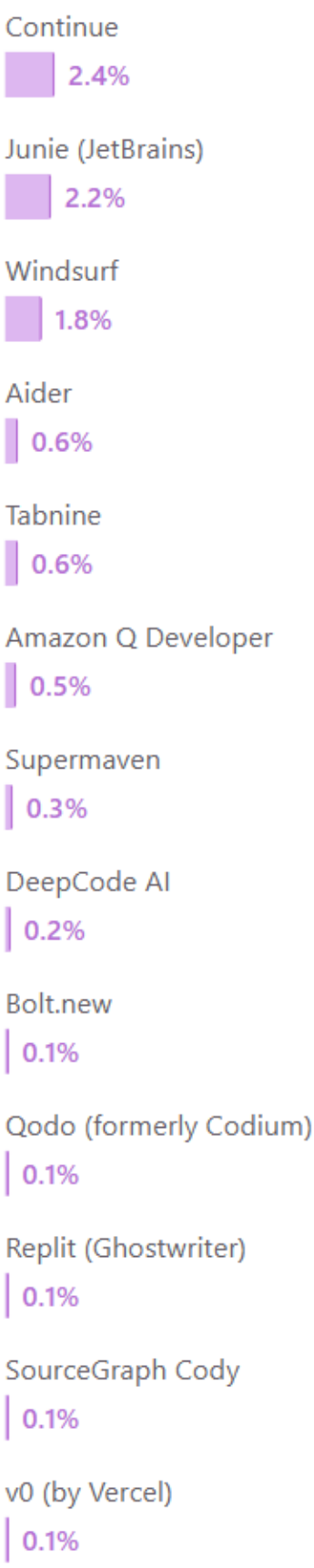
Summary of **"Other"** write-in responses:

- The dominant barriers are **policy/security constraints**, **ethical/environmental objections**, and **lack of trust in output quality**.
- For C++ specifically, respondents repeatedly point to AI tools struggling with **large projects, complex build systems, macros, generated code, correctness, idiomatic modern C++, and the cost of reviewing/fixing generated output**.
- Notable tone: Compared with a neutral "Other" list, **this set contains unusually strong negative sentiment**: "unethical," "garbage," "harmful," "fasc tech," "burning planet," and similar language. That suggests a segment of respondents is not merely unconvinced by AI tools but actively opposed to them.

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

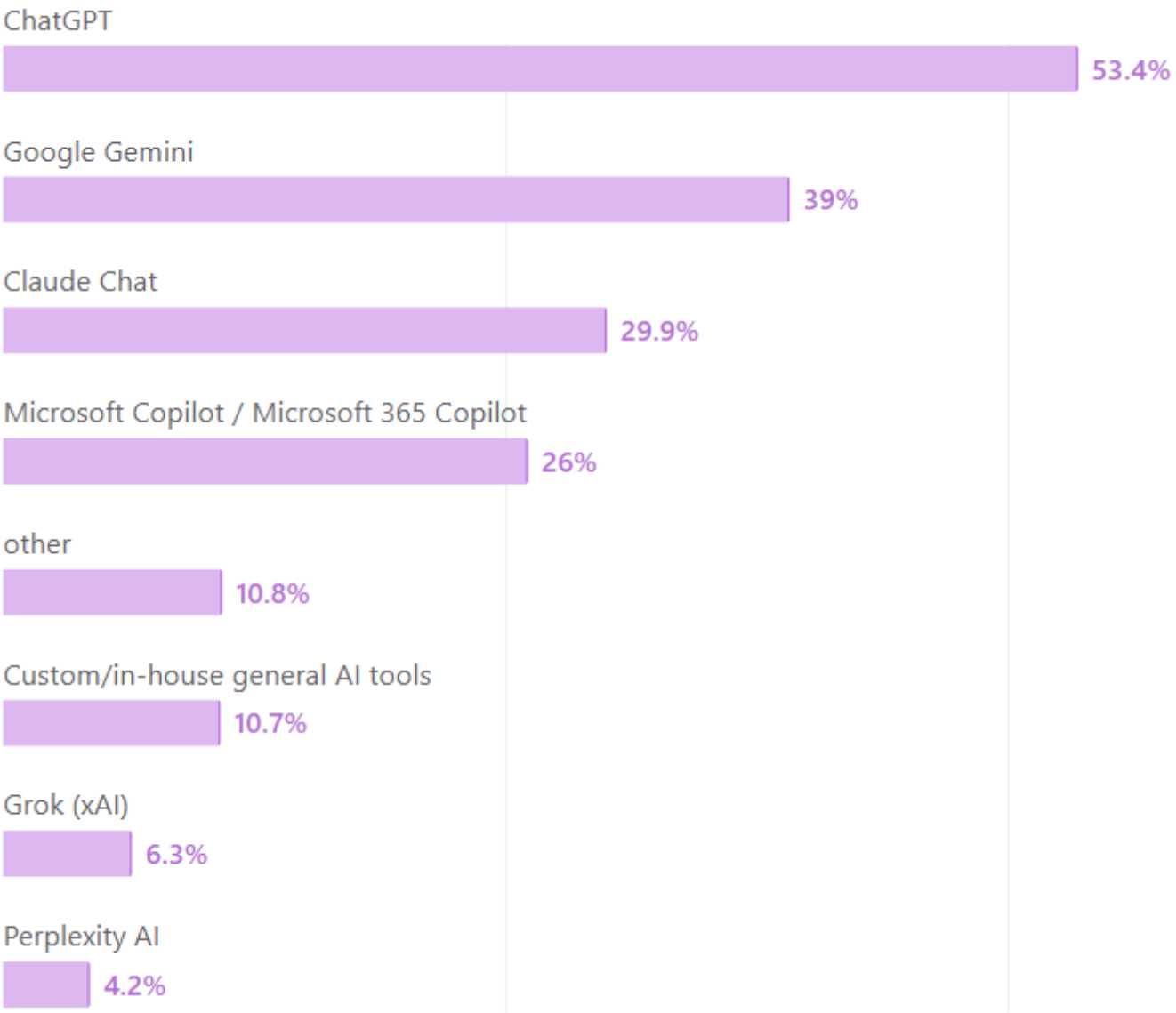
1083 out of 1434 people answered this question.




Continue
2.4%
Junie (JetBrains)
2.2%
Windsurf
1.8%
Aider
0.6%
Tabnine
0.6%
Amazon Q Developer
0.5%
Supermaven
0.3%
DeepCode AI
0.2%
Bolt.new
0.1%
Qodo (formerly Codium)
0.1%
Replit (Ghostwriter)
0.1%
SourceGraph Cody
0.1%
v0 (by Vercel)
0.1%

11 What not-code-specific general AI assistants do you also use for code in your current and recent projects (e.g., pasting code or error output into plain ChatGPT, or asking Grok to write code)?

1046 out of 1434 people answered this question.



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

966 out of 1434 people answered this question.

[AI-generated summary]

Primary themes (most commonly cited and emphasized)

1. **AI is most useful as a faster documentation/search/research front end.**

   Many respondents use it instead of Google/Stack Overflow/cppreference-style lookup, especially for APIs, libraries, compiler flags, standards features, obscure functions, and poor documentation.

   Actionable: improve official C++ documentation discoverability, examples, and AI-consumable structure.

2. **Boilerplate, repetitive code, small utilities, and scaffolding are the clearest coding wins.**

   Users repeatedly cite faster generation of wrappers, getters, skeletons, helper scripts, small functions, test harnesses, and routine edits.

   Actionable: products should target low-risk, easily reviewable, pattern-based code generation.

3. **Tests are a major AI success case.**

   Unit tests, test skeletons, edge-case suggestions, and "more tests than I would write by hand" appear often.

   Actionable: invest in AI-assisted test generation, coverage expansion, and test-data generation integrated with build/CI systems.

4. **Understanding unfamiliar code, libraries, APIs, and legacy systems is a major productivity gain.**

   Respondents value codebase navigation, explanation of old or unknown code, call-path/log analysis, and onboarding to unfamiliar systems.

   Actionable: tools should emphasize code comprehension, semantic search, dependency/context maps, and "explain this subsystem" workflows.

5. **Debugging, compiler-error explanation, and build/tooling diagnosis are frequent high-value uses.**

   C++ template errors, vague compiler diagnostics, build failures, CI-only problems, package/configuration issues, logs, and third-party library interactions are common pain points.

   Actionable: AI should be paired with compilers, sanitizers, static analysis, build logs, and repro/ minimization tools.

Secondary themes (less frequent but notable)

6. **Refactoring and modernization are valued, especially at scale.**

   Users report gains from large mechanical refactors, modernization, repeated pattern changes across files/repos, and edits beyond regex.

   Actionable: combine AI with AST-aware tooling, clang-tidy, compiler feedback, formatting, and safe transactional review.

7. **Code review and bug finding are promising but uneven.**

   Many find AI useful for second-pass review, subtle bugs, PR summaries, thread-safety hints, and neutral review comments; others report false positives and review burden.

   Actionable: AI review tools need confidence levels, evidence links, reproducible reasoning, and suppression controls.

8. **Prototyping and "blank page" acceleration are common benefits.**

   AI helps users get from idea/spec to first working draft, proof of concept, or competing design alternatives faster.

   Actionable: support explicit prototype-to-production workflows with review, hardening, and cleanup stages.

9. **Learning C++ features and concepts is a recurring benefit.**

   Respondents use AI to learn modern C++, standard-library features, coroutines, executors, reflection, SFINAE/templates, platform APIs, and algorithms.

   Actionable: standards and tool vendors should provide authoritative examples and explanations suitable for AI retrieval.

10. **AI as rubber duck / design partner is useful for some users.**

    Users cite brainstorming, architecture tradeoff discussion, naming, "second opinion," and idea exploration.

    Actionable: design-assistant modes should be clearly separated from code-generation modes and should preserve architectural context.

Honorable mentions / significant cautions

11. **A large minority report no meaningful productivity gain, or a net loss.**

    Common reasons: unreliable output, hallucinated APIs, subtle bugs, excessive review time, poor C++/systems understanding, context limits, and AI-generated coworker/OSS "slop."

12. **Trust, review cost, and maintainability are the dominant negative themes.**

    Many say AI is helpful only when outputs are small, local, and easy to verify; complex C++, performance-sensitive, safety-critical, embedded, regulated, or architectural work remains poorly served.

13. **Workplace constraints matter.**

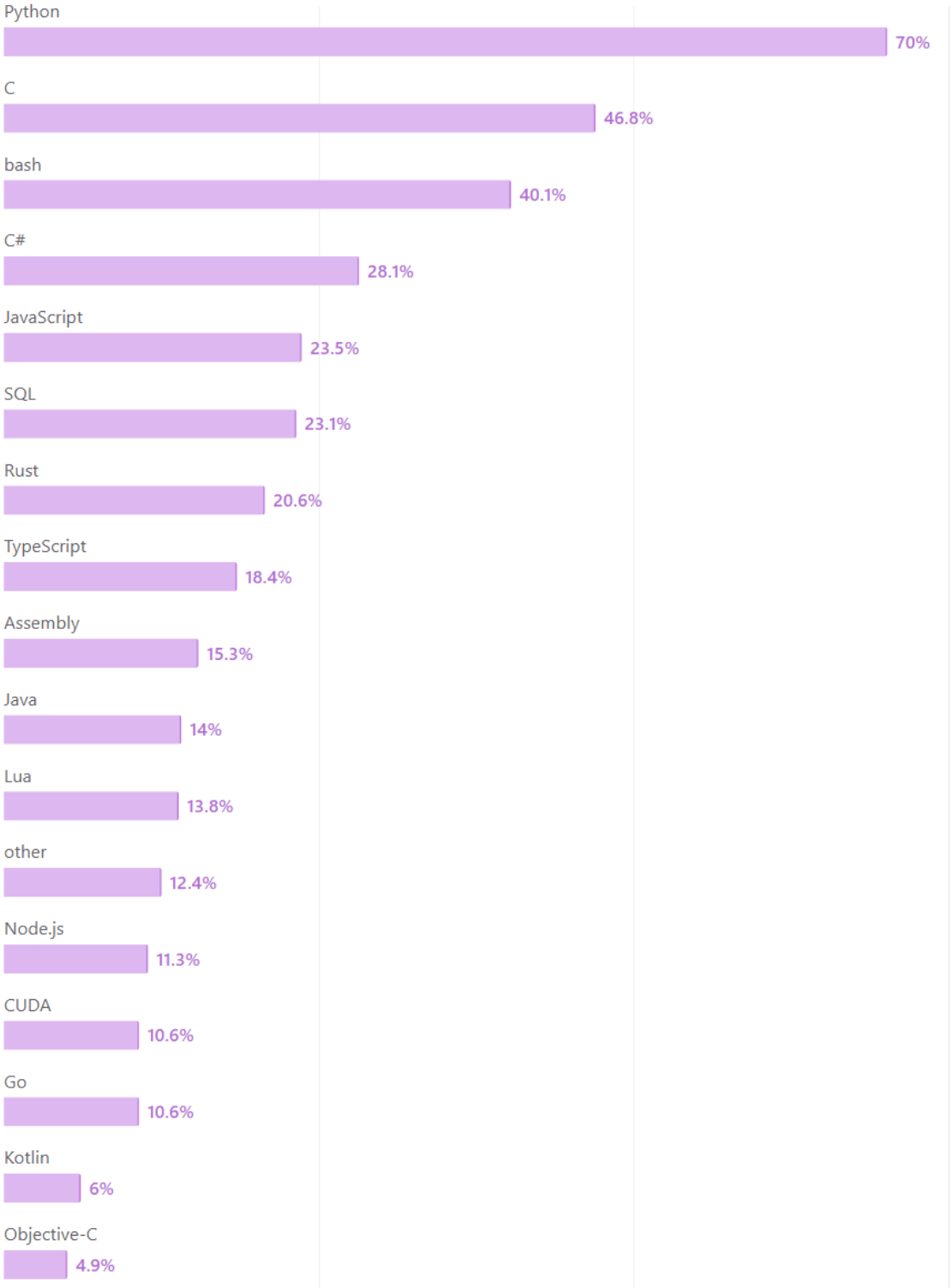
    Some cannot use cloud AI because of employer policy, privacy/IP concerns, air-gapped systems, regulated domains, or lack of approved tooling.

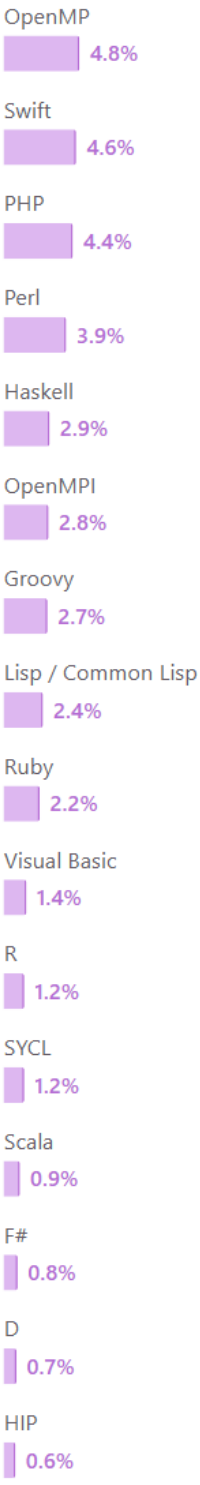
14. **The best pattern is "AI drafts, human reviews."**

    Positive respondents often stress that productivity gains require strong human expertise, tight scope, good prompts, existing tests, compiler/static-analysis feedback, and line-by-line review.

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

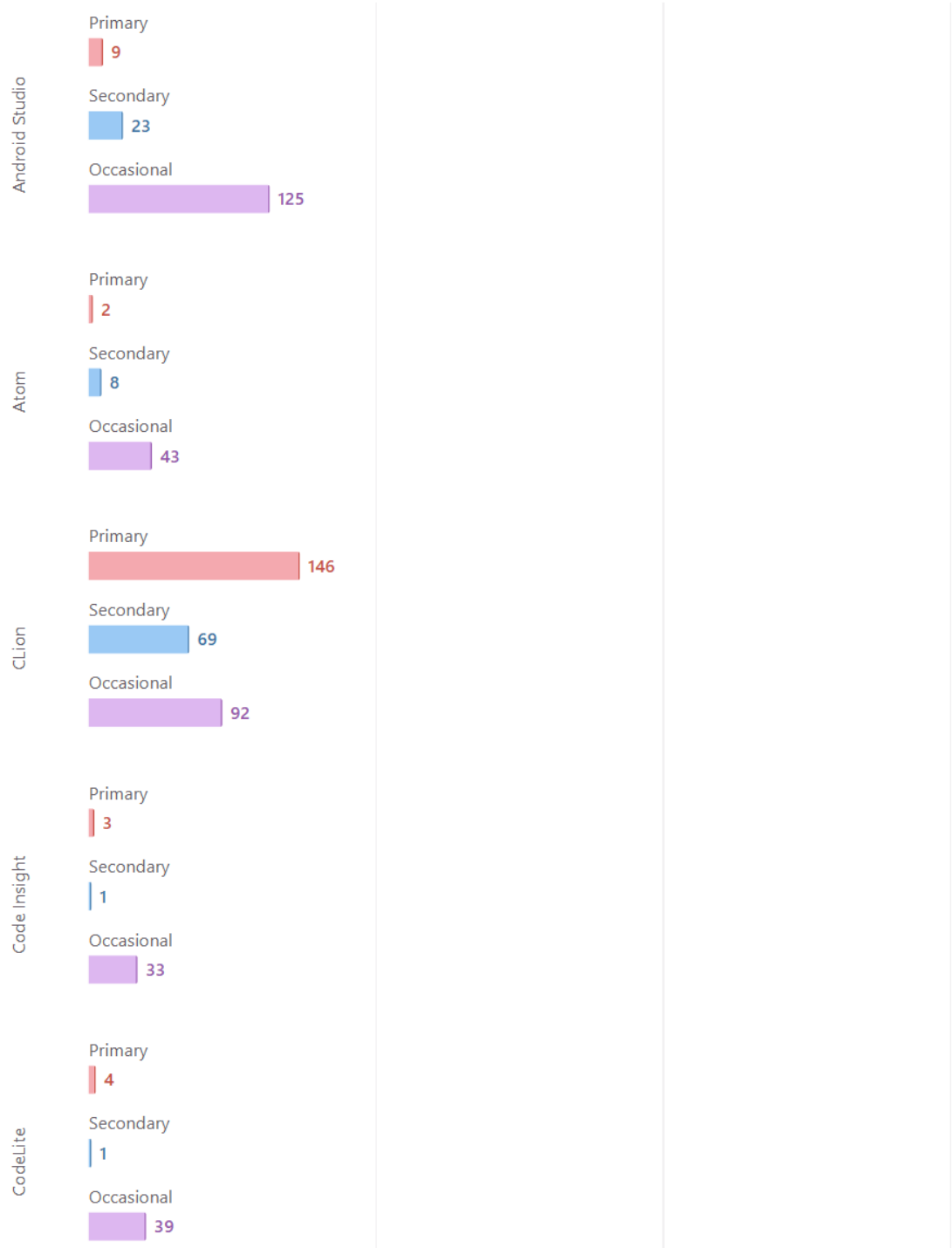
1403 out of 1434 people answered this question.



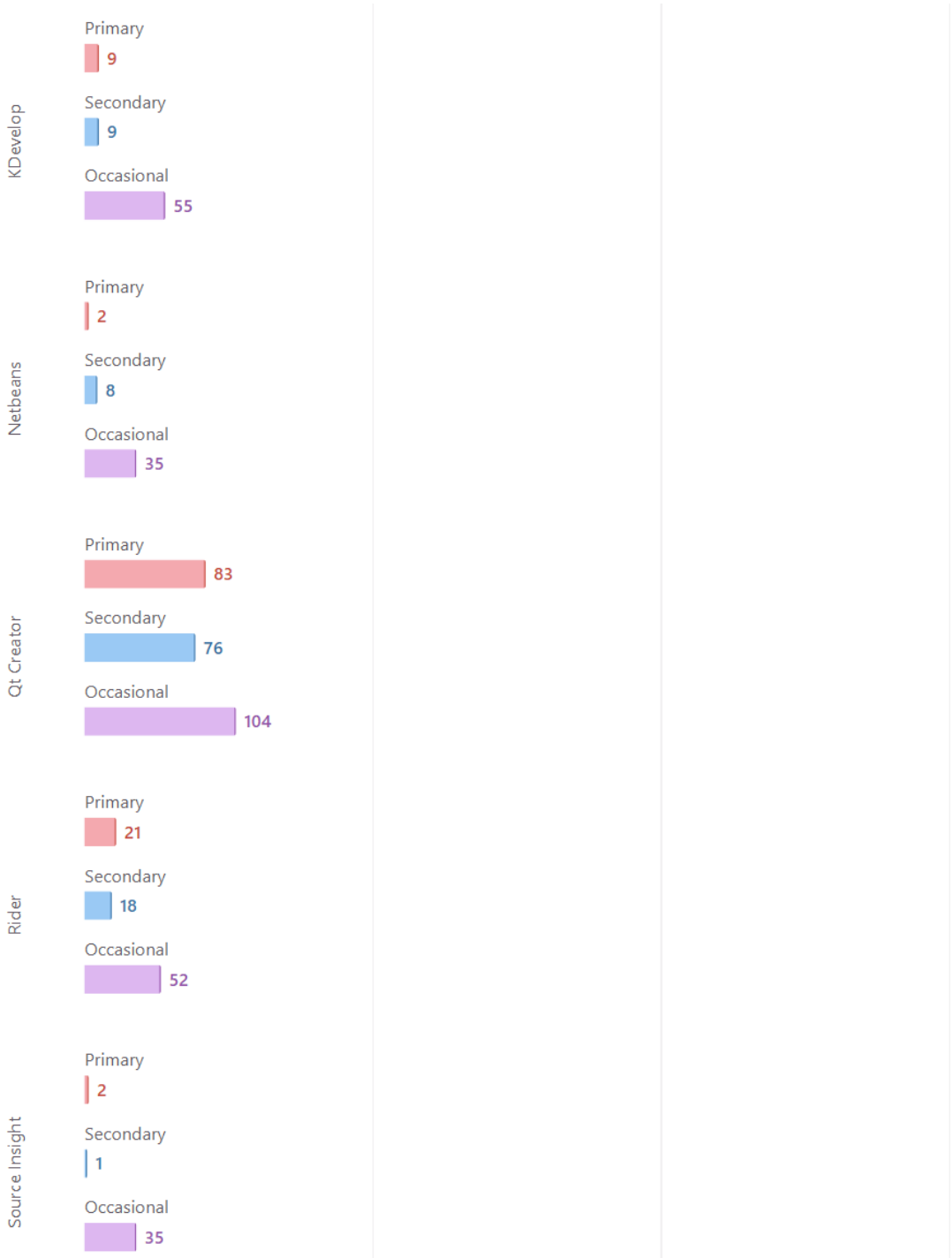

OpenMP
4.8%
Swift
4.6%
PHP
4.4%
Perl
3.9%
Haskell
2.9%
OpenMPI
2.8%
Groovy
2.7%
Lisp / Common Lisp
2.4%
Ruby
2.2%
Visual Basic
1.4%
R
1.2%
SYCL
1.2%
Scala
0.9%
F#
0.8%
D
0.7%
HIP
0.6%

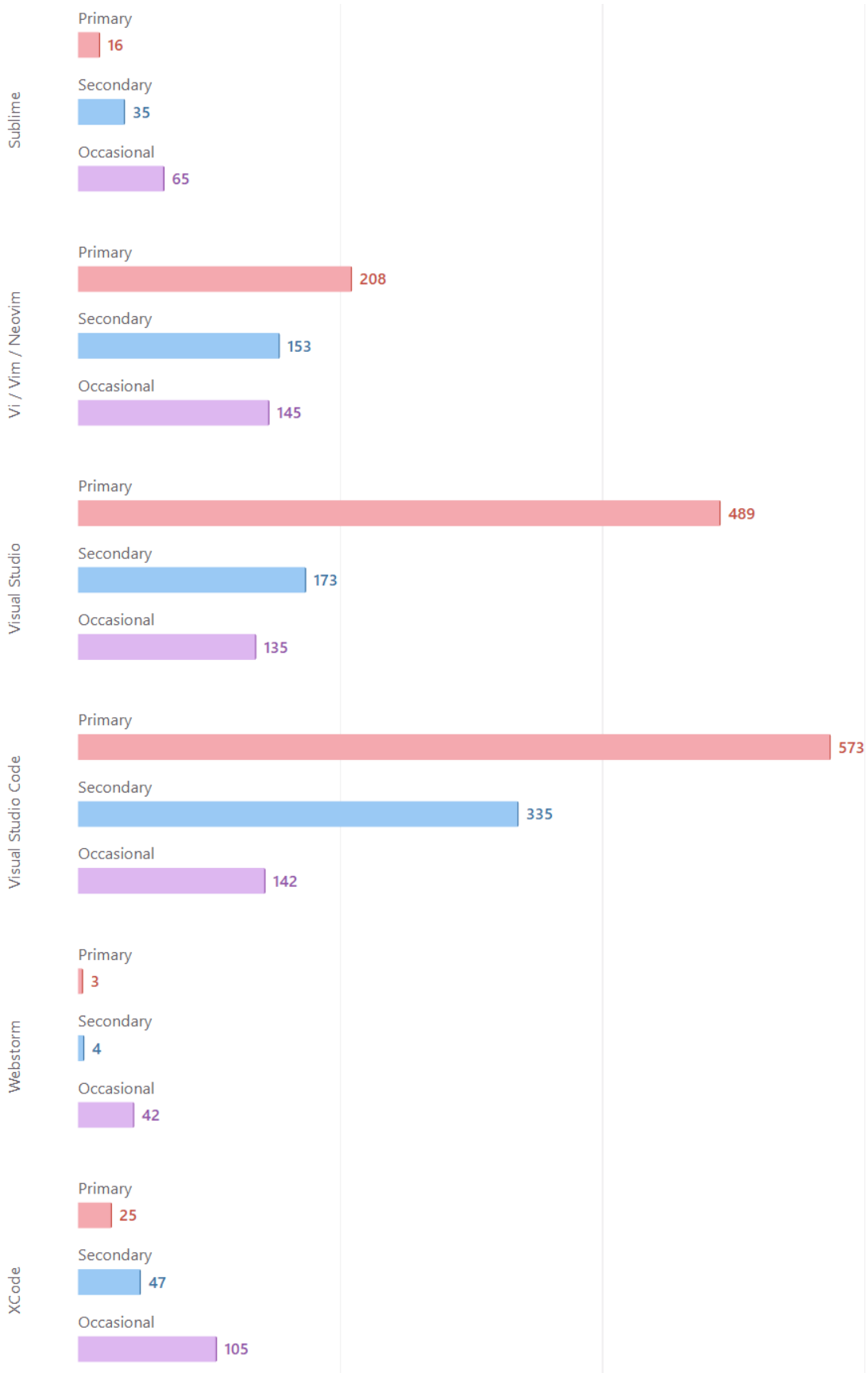
14 Which development environments (IDEs) or editors do you use for C++ development?

1405 out of 1434 people answered this question.



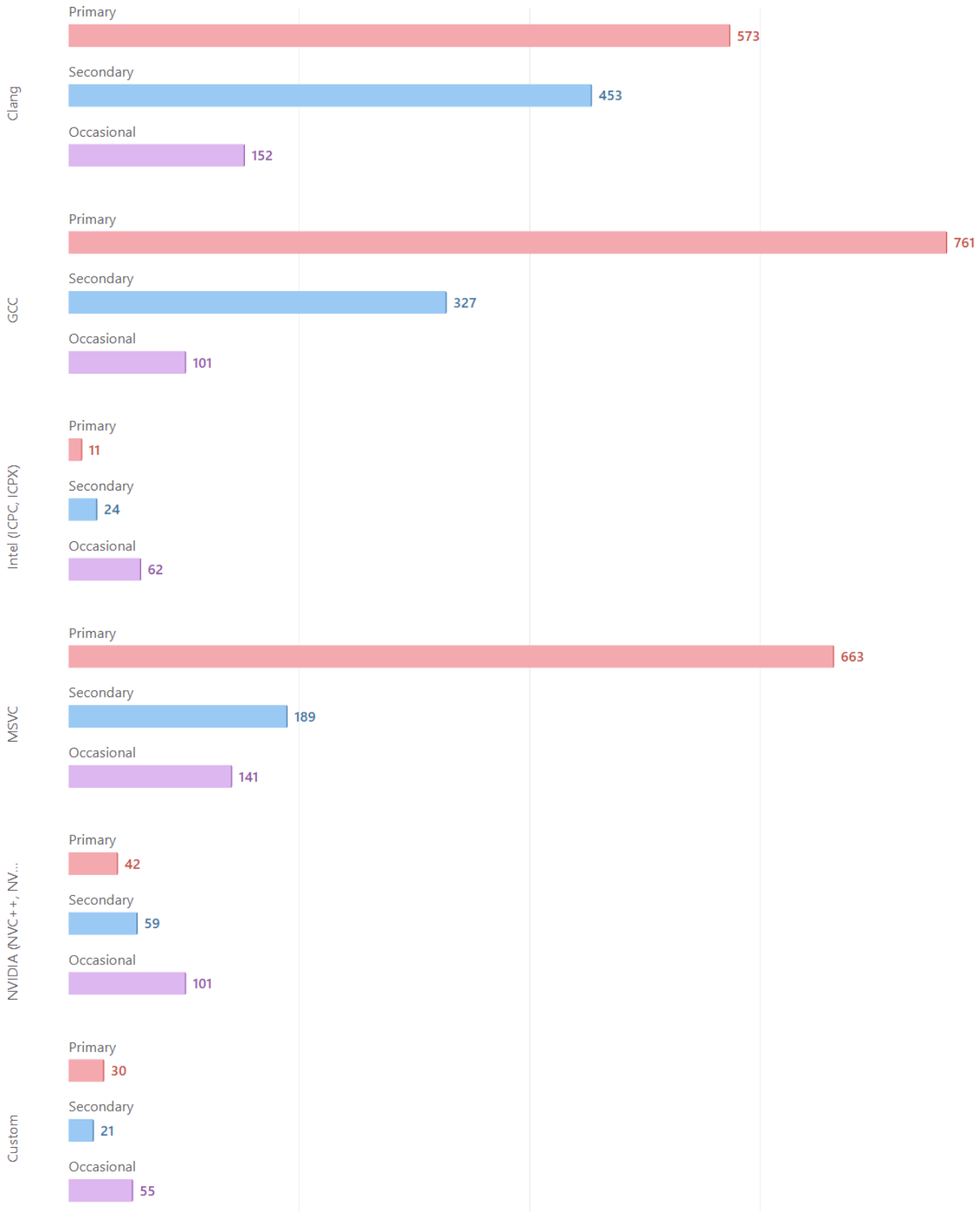

Code::blocks
Primary
8
Secondary
14
Occasional
54
Eclipse
Primary
19
Secondary
27
Occasional
63
Emacs
Primary
64
Secondary
26
Occasional
64
Intellij IDEA
Primary
9
Secondary
23
Occasional
73
Kate
Primary
15
Secondary
37
Occasional
54


KDevelop
Primary
9
Secondary
9
Occasional
55
Netbeans
Primary
2
Secondary
8
Occasional
35
Qt Creator
Primary
83
Secondary
76
Occasional
104
Rider
Primary
21
Secondary
18
Occasional
52
Source Insight
Primary
2
Secondary
1
Occasional
35


Sublime
Primary
16
Secondary
35
Occasional
65
Vi / Vim / Neovim
Primary
208
Secondary
153
Occasional
145
Visual Studio
Primary
489
Secondary
173
Occasional
135
Visual Studio Code
Primary
573
Secondary
335
Occasional
142
Webstorm
Primary
3
Secondary
4
Occasional
42
XCode
Primary
25
Secondary
47
Occasional
105

15 Which compilers do you use for C++ development?

1426 out of 1434 people answered this question.



16a If you had the power to instantly change anything about C++ or its libraries/tools, what would you change and how would it help your daily work?

924 out of 1434 people answered this question.

[AI-generated summary]

Primary Themes (most frequent and emphasized)

1. Standard package/dependency management, simpler build integration

Problem: Disproportionate time on dependency acquisition, linking, CMake/vcpkg/Conan/Bazel fragmentation, and cross-platform setup.

Proposed change: Standardize a Cargo/pip-like package manager, registry/interchange format, and common project/build metadata.

Impact: Faster onboarding, easier library use, fewer bespoke build systems, better enterprise/SBOM/security workflows.

2. Make modules/header replacement actually work

Problem: Headers, includes, macros, TU boundaries, and current modules support remain painful, slow, and inconsistently supported.

Proposed change: Make modules production-ready across compilers, build systems, package managers, IDEs, and standard libraries.

Impact: Reduced include hell, faster builds, cleaner project structure, and fewer fragile macro/export hacks.

3. Reduce build times and improve iteration speed

Problem: Full and incremental builds are repeatedly described as too slow, especially with templates, headers, ranges, and big std headers.

Proposed change: Prioritize compilation performance: better module caching, compiled templates, smaller headers, build-system dependency precision.

Impact: Shorter edit/build/test cycles, less CI waste, and more practical use of modern C++ in large codebases.

4. Allow ABI/compatibility breaks, epochs, or versioned evolution

Problem: ABI stability and backward compatibility are seen as blocking stdlib repair, safer defaults, and removal of legacy baggage.

Proposed change: Introduce epochs/profiles/versioned ABIs or explicit opt-in compatibility breaks.

Impact: Enables fixing old mistakes without freezing the ecosystem around historical constraints.

5. Safety profiles, safer defaults, and fewer footguns

Problem: Bugs from undefined behavior, implicit conversions, unsafe legacy C compatibility, dangling/lifetime issues, and wrong defaults.

Proposed change: Add enforceable profiles/subsets: no C-style casts, stricter conversions, const/explicit/noexcept defaults, lifetime/borrow checking.

Impact: Fewer latent defects, better static checking, easier code review, and stronger confidence in C++ for safety-critical work.

Secondary Themes (less frequent but notable)

6. Simplify the language and standard library

Problem: Users complain of too many features, too many ways to do the same thing, and rising cognitive load.

Proposed change: Consolidate, deprecate, remove obsolete features, slow feature churn, and favor simple language mechanisms over template-heavy libraries.

Impact: Easier teaching, maintenance, review, and long-term mastery.

7. Faster, more complete, more consistent standards implementation

Problem: C++20/23/26 features are unevenly available across MSVC, GCC, Clang, IDEs, platforms, and standard libraries.

Proposed change: Require stronger implementation maturity before standardization; improve conformance and feature parity.

Impact: Developers can use new features sooner and with less per-compiler workaround code.

8. Expand practical standard-library coverage

Problem: Users repeatedly need third-party libraries for common basics: networking, JSON/XML, Unicode, GUI, strings, async tasks, units, UUID/IP, crash reporting.

Proposed change: Add or standardize broadly useful libraries and improve existing weak areas such as regex, strings, chrono, locale, and formatting.

Impact: Less dependency friction, more portable applications, and fewer incompatible ad hoc library choices.

9. Better diagnostics, static analysis, and tooling

Problem: Template/concepts errors, linker errors, false-positive searches, weak LSP behavior, and sanitizer friction waste time.

Proposed change: Improve compiler diagnostics, refactoring, static analysis, sanitizer usability, docs generation, and IDE/LSP correctness.

Impact: Faster debugging, easier learning, better maintainability, and less time spent interpreting tool output.

10. Reflection and code generation

Problem: Boilerplate for serialization, config, GUI binding, database mapping, and type introspection is manual or macro/tool driven.

Proposed change: Deliver static reflection with practical compile-time code generation.

Impact: Less repetitive code, safer refactoring, simpler metaprogramming, and potentially faster compilation than template-heavy workarounds.

Honorable Mentions

Unicode/text handling

Problem: UTF-8/UTF-16 conversion, casing, locale, and char8_t interactions remain inconsistent.

Proposed change: Make Unicode/UTF-8 a first-class, coherent standard-library capability.

Impact: Less platform-specific text code, fewer third-party patches.

Embedded and low-level support

Problem: Locale, exceptions, RTTI, dynamic allocation, hardware addresses, and type punning are often poor fits for embedded/resource-constrained targets.

Proposed change: Provide freestanding-friendly facilities, constexpr hardware-address support, exception-free APIs, allocator control, and low-overhead modes.

Impact: Makes modern C++ more usable where size, determinism, and hardware control matter.

Interop and migration paths

Problem: Users want easier interop with C, Rust, Swift, .NET/CLR, GPU/device code, and binary libraries.

Proposed change: Standardize safer ABI contracts, FFI patterns, and migration/interop support.

Impact: Protects existing C++ investments while reducing pressure to rewrite wholesale.

16b What organizations come to your mind the most when you think about C++? Why?

707 out of 1434 people answered this question.

[AI-generated summary] Summary, in frequency order

1. ISO / WG21 / C++ Standards Committee

This was by far the most frequently mentioned organization cluster. Respondents usually cited it because it defines the C++ standard, controls language evolution, publishes new features, and gives C++ its formal direction. Positive comments described the committee as essential, transparent, and responsible for moving the language forward. Negative comments focused on slow progress, over-complex language design, and frustration with features such as modules.

2. Microsoft / MSVC / Visual Studio

Microsoft was the most frequently mentioned company. Most respondents associated it with Visual Studio, MSVC, Windows development, VS Code, vcpkg, and long-term support for C++ tooling. Positive sentiment emphasized strong IDE support, compiler investment, Windows-native development, and visible standards participation. Negative sentiment focused on MSVC quality, standards conformance gaps, bloat, and the feeling that Microsoft has too much influence over C++.

3. Google / Alphabet

Google was mentioned as a major C++ user, open-source contributor, and standards/community participant. Respondents cited GoogleTest, Abseil, protobuf, Google Benchmark, Chromium, coding guidelines, large-scale infrastructure, and contributions to Clang/LLVM. Positive sentiment centered on useful libraries, massive production C++ codebases, and broad community influence. Negative sentiment focused mainly on dislike of Google's C++ style guide, perceived poor or overly idiosyncratic code practices, and a sense that Google may be reducing its C++ investment.

4. Bloomberg

Bloomberg was strongly associated with C++ conferences, committee participation, funding, sponsorship, talks, trading systems, and language-evolution work. Many respondents mentioned Bloomberg because its engineers are visible at CppCon and WG21 and because the company is seen as a major financial-industry C++ user. Positive sentiment credited Bloomberg with serious investment in C++ and influence on features such as reflection or contracts. Negative sentiment centered on concern that Bloomberg has disproportionate influence and may be unrepresentative of the wider C++ community.

5. LLVM / Clang

LLVM and Clang were commonly mentioned as core compiler and tooling infrastructure. Respondents cited Clang, clang-format, clang-tidy, clangd, libclang, and the broader LLVM ecosystem as tools they use directly. Positive sentiment was very strong: LLVM was described as forward-looking, readable, contributable, and foundational for modern C++ tooling. There was little recurring negative sentiment beyond occasional general complaints about compiler implementation difficulty.

6. Boost / C++ Alliance

Boost was repeatedly described as an essential library ecosystem and a proving ground for future standard-library features. Respondents valued it for filling gaps in the standard library, providing high-quality reusable components, and influencing C++ evolution. The C++ Alliance was usually mentioned in connection with Boost stewardship and ecosystem support. Sentiment was mostly positive, with occasional fatigue around complexity but no broad repeated negative theme.

7. GNU / GCC / FSF

GNU and GCC were frequently cited as one of the pillars of practical C++ development. Respondents mentioned GCC, g++, libstdc++, and the FSF as crucial compiler and open-source infrastructure. Positive sentiment emphasized GCC's importance, availability, and role in preventing compiler tooling from becoming entirely commercial. Negative sentiment was minimal.

8. NVIDIA

NVIDIA was associated with CUDA, GPU computing, high-performance computing, conference presence, and standards work around execution models. Respondents mentioned CUDA, TensorRT, CCCL, std::execution, and NVIDIA engineers' visibility in talks and committees. Positive sentiment focused on serious C++ use in performance-critical domains and visible technical leadership. Negative sentiment was minimal.

9. CppCon

CppCon was frequently mentioned as a central C++ learning and community institution. Respondents associated it with talks, YouTube videos, standards updates, recognizable speakers, sponsors, and visibility for companies such as Bloomberg, Microsoft, Google, Adobe, and NVIDIA. Positive sentiment centered on education, conference culture, and keeping developers aware of modern C++ practices. Some respondents were critical of "ivory tower" tendencies, but the dominant sentiment was that CppCon is a major source of C++ knowledge.

10. Qt / Qt Group / KDE

Qt was mentioned both as a company/ecosystem and as a practical C++ UI framework. Respondents cited daily use of Qt, GUI development, cross-platform desktop work, and KDE-related development. Positive sentiment emphasized Qt as a productive, mature, high-level C++ framework that made C++ useful for application development. Negative sentiment was minimal.

11. Meta / Facebook

Meta/Facebook was usually grouped with Google and other large-scale C++ users. Respondents cited large C++ codebases, open-source libraries such as Folly, infrastructure, and big-company use of C++ at scale. Positive sentiment focused on useful libraries and the credibility that comes from running large production systems in C++. Negative sentiment was minimal.

12. Yandex

Yandex was cited as a major C++ employer, conference organizer, backend C++ user, and source of tools or frameworks such as userver. Positive sentiment focused on C++ visibility, education, talks, internships, and large-scale production use. Negative sentiment was minimal.