POLYRHYTHM

SEPTEMBER 2026

WHITEPAPER

Best AI for Senior C++ Developers in 2026

PUBLICATION DETAILS

AUTHORS	Polyrhythm Software, LLC
PUBLISHED	September 4, 2026
UPDATED	—
KEYWORDS	cpp, software engineering, delivery, technical debt

POLYRHYTHM

EXECUTIVE SUMMARY

> No public benchmark measures what senior-level C++ work actually is: object lifetime, ABI stability, exception safety, build graphs, and years of architectural consequence. This paper triangulates Senior SWE-Bench, CursorBench 3.2, and Terminal-Bench 4.0, then applies the engineering criteria those leaderboards omit, to rank Claude Fable 5.1, Claude Opus 5, GPT-6 Astra, GPT-5.6 Sol, and Grok 4.6 for the seat that owns a mature C++ system.

Last fact-checked: September 4, 2026

Bottom line: For senior-level modern C++ available today, **Claude Fable 5.1** is our current primary recommendation. It does not win every benchmark, and no public benchmark directly measures the full job of maintaining a mature C++ system. It does, however, have the strongest current combination of senior-repository results, ambiguous multi-file task performance, and independent terminal-agent performance.

Use **Claude Opus 5** as the workhorse on the same Claude Max seat. Treat **GPT-6 Astra**, whose rollout began September 3, as the first serious challenger to retest as soon as it is actually available in your version of ChatGPT, Codex, or the API. Use **Grok 4.6** as a lower-cost independent reviewer, not as the engineer who owns lifetime, ABI, and cross-library consequences.

That ranking is about judgment, not about who dumps the most compiling C++ into a merge request.

Senior-Level C++ is not "write a function"

Most coding benchmarks ask whether a model can fix a ticket, implement a feature, or make tests pass. Those are useful. They are not the job.

A senior-level C++ assistant has to answer questions like:

- Is this the right data structure for *this* cardinality and access pattern?
- Who owns this object, and is that obvious from the signature?
- Does this abstraction reduce complexity, or merely relocate it?
- Are these templates justified?
- Composition, inheritance, a policy, a concept, or nothing?
- What happens to exception safety if this operation fails halfway?
- Does the public API freeze a bad implementation?
- Will this design survive concurrency and ABI stability?
- Is the CMake graph idiomatic, or a pile of global flags?
- Is the vcpkg manifest reproducible on a GitLab runner?
- Does the pipeline test what we think it tests?
- Did the model fix the problem, or only make `ctest` green?

A model that manufactures valid code faster than you can review it is a liability. Accidental complexity, produced at scale, is worse than producing nothing.

There is still no direct benchmark for senior-level C++

We found no credible public benchmark that directly tests mature C++ maintenance across object lifetime, ABI stability, exception safety, CMake, vcpkg, sanitizers, multiple compilers, multiple platforms, and years of architectural decisions.

There are now better proxies than there were even a few months ago, but each measures a different slice of the problem. The honest way to rank models is to triangulate across them, then apply engineering judgment.

Senior SWE-Bench is the closest general proxy

Senior SWE-Bench is the most relevant current public benchmark for senior software-engineering judgment. It uses 100 realistic, often underspecified tasks drawn from 12 real repositories. Its stricter **Tasteful Solve** gate checks more than functional completion. It also evaluates task-specific rubrics, unnecessary code growth, codebase conventions, and relative implementation quality.

That is much closer to senior work than a benchmark that asks only whether a patch passes tests.

It is still not a C++ benchmark. Its repositories use Python, TypeScript, Go, Rust, Elixir, and SQL. It does not test C++ lifetime, templates, ABI, CMake, or sanitizer-driven debugging.

Current pass@1 results:

Model	Tasteful Solve	Basic Solve
Claude Fable 5.1	34.7%	57.9%
Claude Opus 5	34.7%	62.1%
Claude Fable 5	34.7%	53.7%
GPT-5.6 Sol	34.7%	53.7%
Grok 4.6	26.3%	51.6%

Two conclusions matter.

First, Senior SWE-Bench does **not** establish a unique quality winner among Fable 5.1, Opus 5, Fable 5, and GPT-5.6 Sol. They tie on the stricter Tasteful Solve score. Second, Opus 5 currently has the highest

Basic Solve rate in that group. Anyone using this benchmark alone to claim that Fable 5.1 decisively beats Opus 5 is overstating the evidence.

GPT-6 Astra did not yet have a public Senior SWE-Bench result when this article was checked.

CursorBench 3.2 favors Fable 5.1

CursorBench 3.2 is another useful proxy. Cursor built it from ambiguous, multi-file tasks modeled on real Cursor sessions, including planning, refactoring, review, and changes that must preserve downstream behavior.

Model	CursorBench 3.2	Benchmark cost per task
Claude Fable 5.1, max	73.4%	$9.64
Grok 4.6, extra high	70.8%	$2.81
Claude Fable 5, max	70.5%	$17.32
Claude Opus 5, max	70.0%	$8.23
GPT-5.6 Sol, max	67.2%	$5.69

Fable 5.1 leads this snapshot. Grok 4.6 is closer than its reputation and considerably cheaper in Cursor's benchmark run. Sol trails on this harness.

The caveats matter. CursorBench is a first-party benchmark from the company operating the harness. The cost column is observed benchmark cost, not subscription pricing. Small score differences should not be treated as proof of a meaningful real-world gap unless the benchmark publishes enough uncertainty information to support that conclusion.

Terminal-Bench measures the agent and model together

Terminal-Bench 4.0 evaluates real terminal-based software-engineering and systems tasks. It is useful for CMake failures, toolchain problems, build logs, package migrations, and the configure-build-debug loop, although it is not C++-specific.

The current independent Snorkel leaderboard reports agent-model systems, not isolated model intelligence:

Agent and model	Terminal-Bench 4.0
Claude Code + Claude Fable 5.1, max	57.9% ±3.8

Claude Code + Claude Opus 5, max	51.8% ±3.4
Claude Code + Claude Fable 5, max	44.5% ±3.8
Codex + GPT-5.6 Sol, max	37.3% ±3.8
Grok Build + Grok 4.6, high	20.3% ±3.1

This board favors Fable 5.1, but it also measures Claude Code, Codex, and Grok Build. A result can reflect the model, the agent harness, tool-use policy, effort setting, or their interaction.

OpenAI's GPT-6 Astra launch material contains a separate cross-vendor table:

Model	Terminal-Bench 4.0	DeepSWE v1.1	FrontierCode 1.1 Main
GPT-6 Astra	57.9%	74.1%	53.3%
Claude Fable 5.1	55.8%	67.4%	50.9%
Claude Opus 5	52.3%	73.7%	53.4%
GPT-5.6 Sol	37.3%	72.7%	47.5%

Those are **OpenAI-reported comparisons**, not independent leaderboard results. OpenAI also reports that Astra scored 63.9% against Fable 5.1's 57.8% on an internal database-migration evaluation. On FrontierCode 1.1 Main, however, Astra does not lead Opus 5, and Fable 5 also scores slightly above Astra in OpenAI's full table. This is a competitive cluster, not a clean sweep.

The most defensible current conclusion is:

- **Fable 5.1** has the strongest combined public case for difficult, ambiguous, long-horizon work that you can broadly use today.
- **Opus 5** remains extremely competitive and is the more economical volume model.
- **GPT-6 Astra** has strong vendor-reported coding results, but independent Senior SWE-Bench and Terminal-Bench results are still needed.
- **Grok 4.6** is valuable as an inexpensive dissenting reviewer, but its independent senior-repository and terminal results do not support making it the primary seat.

Why Fable 5.1 fits senior-level C++

The advantage is not prettier `for` loops. It is holding the shape of the problem.

Take a refactor that looks one-line:

> *Remove shared ownership of `SimulationContext`.*

Changing `std::shared_ptr<SimulationContext>` to `SimulationContext&` is the easy part. The work is constructors, factories, worker lifetime, asynchronous callbacks, test fixtures, bindings, plugins, and destruction order. A junior agent starts editing. A senior agent traces the ownership graph first and is willing to say the proposed model is invalid.

Anthropic positions Fable 5.1 for demanding reasoning and long-horizon agentic work. Its own guidance is to start with Opus 5 for most workloads, then use Fable 5.1 when Opus at high effort falls short. That is sensible product guidance, and it also suggests the right economic split for a senior C++ workflow.

Anthropic says Fable 5.1 is designed to avoid superficial shortcuts and pursue root causes. That is a vendor claim, not independent proof, but it matches the class of work where the model is most useful: cross-codebase reasoning, rare failures, difficult reviews, performance work, and refactors whose consequences are not local to the edited file.

Fable 5.1 also exposes a one-million-token context window and up to 128,000 output tokens through supported interfaces. Those limits help with large repositories and long sessions, but context capacity is not architectural understanding. A model can read a million tokens and still choose the wrong ownership model.

At standard API rates, Fable 5.1 costs $10 per million input tokens and $50 per million output tokens; Opus 5 costs $5 and $25. For senior-level C++ specifically, we would use Fable 5.1 on the hard problems and keep Opus 5 as the volume model.

Data structures: stop saying O(1)

A model that recites `std::unordered_map` is not useful. A senior assistant notices that expected cardinality is twelve, mutation is rare, iteration dominates, locality matters, and a `std::vector` with a linear scan is probably the better representation.

The discussion you want includes:

- expected and worst-case cardinality
- mutation frequency
- iteration shape
- cache behavior
- allocation behavior
- iterator and reference invalidation
- ordering and determinism
- concurrency and synchronization

- serialization format
- what the public API forces on callers

"Use a hash map because lookup is O(1)" is freshman analysis. The model should also tell you when *your* proposed structure is wrong.

Fable 5.1 is the model we would trust most to sustain that argument across a subsystem. Opus 5 is close and cheaper. Astra's coding results suggest it belongs in the evaluation immediately, but benchmarks such as DeepSWE and FrontierCode do not directly test C++ data-layout judgment. Grok 4.6 is good enough to disagree with Claude without consuming Fable usage.

Architecture: use the model as an adversary

You do not need the Observer pattern explained. You need an attacker.

Give the model a design and tell it to break the design:

- Where is responsibility unclear?
- Which abstraction is premature?
- What has more than one reason to change?
- Where does ownership cross a boundary?
- Which interfaces leak implementation?
- Where are we paying complexity and getting no flexibility?
- What could simply be deleted?
- Which pattern am I forcing onto the problem?

Useful replies sound like this:

- You already have observer behavior. Formalizing it as a framework would make this worse.
- This class does three jobs because those jobs share state. Split by behavior, then remove the cycle.
- This factory hierarchy exists to construct one concrete type. Delete it.
- This interface does not isolate volatility. It only hides a one-line call behind another virtual dispatch.

Opus 5 is excellent in this role for daily review. Fable 5.1 is the model we would give the subsystem that has become hard to extend. After Claude answers, give Grok 4.6 the same design without Claude's conclusions. Independent convergence is useful evidence. Correlated agreement from two runs of the same model family is much weaker.

The most important prompt instruction is permission to reject the premise. A model that assumes every requested abstraction belongs in the design will make an experienced engineer less productive.

CMake is a better C++ test than HumanEval

It is easy to generate CMake that works. It is hard to generate CMake that encodes a maintainable target graph.

This is not syntax trivia:

```
target_link_libraries(our_library
    PUBLIC
        fmt::fmt
    PRIVATE
        some_internal_library
)
```

In modern CMake, `PUBLIC`, `PRIVATE`, and `INTERFACE` control whether a link dependency and its usage requirements apply to the current target, propagate to consumers, or both:

- `PRIVATE`: used by the current target, not propagated to consumers.
- `PUBLIC`: used by the current target and propagated to consumers.
- `INTERFACE`: propagated to consumers but not used to build or link the current target itself.

They do not literally mean "dependency direction." They encode scope and transitive propagation. In a well-designed target graph, that propagation reflects architectural boundaries.

A useful model also understands imported and alias targets, generator expressions, build and install interfaces, package configuration files, toolchain files, presets, transitive usage requirements, and why globally mutating include paths and compiler flags produces an unmaintainable monorepo.

CMake bugs are repository bugs. Fixing one package exposes assumptions in six others. That is why terminal-agent performance matters more here than whether the model memorized `FetchContent`.

The same standard applies to **vcpkg**. The answer is not `vcpkg install fmt`. It is manifest mode, an intentionally pinned `builtin-baseline`, overlays, custom ports, triplets, platform-specific dependencies, binary caching, and reproducible runner configuration.

A `vcpkg.json` manifest alone does not make a build reproducible. A serious setup also pins or controls the vcpkg tool version, compiler and SDK versions, runner image, overlays, custom triplets, and any exact-version overrides. `builtin-baseline` establishes the baseline versions available to resolution; it is one part of reproducibility, not the whole story.

If adding one library requires changing global compiler state, the dependency design is wrong. The model should notice.

Testing is strategy, not GoogleTest volume

Almost any frontier model can emit cases for `double calculate_intercept(...)`.

A senior assistant asks:

- What are the invariants?
- Which tolerances are meaningful?
- Where are the discontinuities?
- Which inputs are physically impossible?
- Which properties must always hold?
- What belongs in unit tests versus integration tests?
- Would randomized or property-based tests outperform enumerated examples?
- Should the parser be fuzzed?
- Are ASan, UBSan, TSan, or MSan applicable?
- Are we testing externally meaningful behavior or private implementation details?

Then the agent has to run the loop without weakening the agreed acceptance criteria. Disabling a failing test, loosening an assertion, suppressing a sanitizer finding, or adding a sleep until a race "goes away" is not completion.

Our preferred split is:

- Use **Fable 5.1 or Opus 5** to define invariants, failure modes, and the test strategy.
- Use **Codex** for the repetitive configure-build-test-log-patch loop after the acceptance criteria are explicit.
- Return to the reasoning model when the loop reveals that the design, not the patch, is wrong.

That is a workflow recommendation, not a benchmark result. Agent products change quickly, and the best operator depends heavily on repository setup, permissions, tool feedback, and how aggressively the agent is allowed to modify the tree.

GitLab CI is software engineering written in YAML

A production C++ pipeline may coordinate:

- vcpkg restore and binary caching
- CMake configure and build
- Ninja and MSBuild generators
- GCC, Clang, and MSVC matrices
- Debug and RelWithDebInfo configurations
- GoogleTest discovery and reporting

- sanitizer jobs
- `clang-tidy`
- code coverage
- build artifacts and container images
- release and approval gates

The hard questions are policy, not YAML:

- Clang and GCC on every commit, or one on every commit and the other nightly?
- Windows on every merge request, or on a schedule?
- Separate sanitizer jobs with clean build directories?
- Is the vcpkg binary cache safe to share across branches and trust boundaries?
- Which artifacts survive, and for how long?
- What exactly constitutes a release candidate?
- Which failures block merge?
- Which warnings are actionable enough to enforce?

Fable 5.1 and Opus 5 are better choices for designing a pipeline you would merge. Codex is a strong choice for reading a failed job, reproducing it, applying a bounded repair, and rerunning the loop. Astra inside Codex is the combination to evaluate first when it is available on your account.

Do not let an agent "simplify" CI by deleting a platform, removing a sanitizer, weakening a gate, or turning a required job into `allow_failure: true`. A green pipeline can be less correct than a red one.

The metric most leaderboards still omit: how much junk did it add?

AI made implementation nearly free. Deletion got more valuable.

Senior SWE-Bench is a notable exception because its Tasteful Solve gate explicitly evaluates bloat and codebase practice. Most coding leaderboards still emphasize whether the requested behavior was completed, not whether the repository is better after the patch.

When reviewing generated C++, look for:

- wrapper classes with one job
- factories with one implementation
- interfaces with one consumer and no volatility boundary
- template indirection nobody asked for
- premature genericity
- defensive copies that hide ownership uncertainty

- `shared_ptr` used as a panic button
- duplicate validation in multiple layers
- comments that apologize for unclear code
- functions shattered until control flow is harder to follow
- configuration options with no real second use case
- compatibility layers for versions that never existed

Nine hundred lines that compile are not six times more productive than one hundred and fifty maintainable lines. They are a backlog.

The prompt that most reliably improves C++ output is still simple:

> *Find the smallest maintainable change that implements this feature while preserving the architecture. Do not introduce an abstraction unless it removes existing complexity or has a concrete second use case. Do not weaken tests, diagnostics, or platform coverage to make the change pass.*

That constraint is worth more than switching models.

GPT-6 Astra is the qualifier, not the winner yet

OpenAI announced **GPT-6 Astra** and began rollout on September 3, 2026. OpenAI said access would begin with a limited set of organizations, then expand across eligible ChatGPT and other product surfaces, Codex, the API, and partner platforms over the following days.

As of September 4, access is not universal. At the time of checking, GPT-6 Astra appears in ordinary Chat as **GPT-6 Pro** for Pro $100, Pro $200, Business, and Enterprise accounts; OpenAI's help page explicitly says it is not included with ChatGPT Plus in Chat. Availability can still differ between Chat, Work, Codex, and the API during rollout. Do not assume that having a particular ChatGPT plan means Astra is already selectable in every product surface.

On OpenAI's own coding table, Astra:

- scores 57.9% on Terminal-Bench 4.0 versus 55.8% for Fable 5.1
- scores 74.1% on DeepSWE v1.1 versus 67.4% for Fable 5.1 and 73.7% for Opus 5
- scores 63.9% versus 57.8% for Fable 5.1 on OpenAI's internal database-migration evaluation
- scores 53.3% on FrontierCode 1.1 Main, slightly below Opus 5 at 53.4% and Fable 5 at 53.5%

Those results make Astra a serious contender. They do not establish it as the best senior-level C++ model. They are vendor-reported, the benchmark environments may differ from production products, and the most relevant independent boards do not yet include Astra.

The product may matter as much as the raw score. OpenAI says Astra can experimentally preserve notes across context windows in Codex, with earlier context remaining searchable. That could matter in a TSan investigation, dependency migration, or cross-library refactor that lasts all afternoon. It is

currently described as an experimental feature, not something to assume is enabled in every Codex session.

We would test Astra first on:

1. CMake and toolchain failures that cross package boundaries.
2. Dependency migrations with explicit rollback and compatibility requirements.
3. GitLab CI failures where the agent must preserve all existing gates.
4. Refactors that touch several libraries, tests, and downstream call sites.
5. Sanitizer investigations where preserving failed hypotheses matters.

Do not crown a model you cannot log into, and do not crown one from its vendor's launch table alone.

Grok 4.6

Grok 4.6 at extra-high effort scores 70.8% on CursorBench 3.2, only 2.6 points behind Fable 5.1 Max, at $2.81 versus $9.64 of observed benchmark cost per task. Grok Build is also a real coding environment with plan mode, parallel subagents, worktrees, repository search, terminal execution, code review, and headless workflows.

That is enough to keep it on the desk.

It is not enough to make it the primary senior-level C++ model. Grok 4.6 trails the top group on Senior SWE-Bench's Tasteful Solve score and trails substantially on the current independent Terminal-Bench 4.0 board. Its public positioning emphasizes moving quickly across codebases and shipping working software. That is valuable, but it is not the same thing as proving superior judgment on lifetime, ABI, exception safety, build graphs, and long-term maintenance.

Use Grok for independent review, competing designs, cheap exploratory drafts, and adversarial disagreement. Hide Claude's answer before asking. The value is model-family diversity, not another vote after you have revealed the conclusion.

One pricing correction matters. The public x.ai pricing page currently lists **SuperGrok at $30 per month** and **SuperGrok Plus at $100 per month**. SuperGrok Heavy remains listed, but the public pricing page checked for this article did not display a monthly price. We would not repeat an older $300 figure as a current fact without a dated invoice or an official price page that still shows it.

How to spend the $200-$300 tier

Relevant current individual pricing:

Seat	Price	Use for senior-level C++
Claude Max 20x	$200/month	Best single buy today. Use Opus 5 for daily review and implementation; spend Fable 5.1 on ownership, concurrency, large refactors, and build-graph changes. The Max plan has five-hour session limits and weekly usage limits. Up to 50% of weekly plan usage can be spent on Fable models at no extra cost.
ChatGPT Pro 20x	$200/month	Best second seat if you live in Codex. It is the first seat to reevaluate once Astra is available in the exact product surface you use. OpenAI also offers a $100 Pro 5x tier. Model-specific and product-specific limits still apply.
SuperGrok	$30/month	Inexpensive independent architecture review and dissenting analysis. Grok Build itself is available to try free; this plan raises Grok usage limits.
SuperGrok Plus	$100/month	Buy for higher usage, not because the public evidence establishes better senior-level C++ judgment.

Our current value recommendation is **Claude Max 20x plus SuperGrok for $230 per month**. That gives you the strongest current primary model, the more economical Opus workhorse, and a genuinely different model family for review.

The alternative is **ChatGPT Pro 20x plus SuperGrok for $230 per month** if your workflow is dominated by unattended Codex execution and Astra is already available to you. We would not switch purely on OpenAI's launch benchmarks. Run the same repository tasks against both stacks first.

Available on September 4, 2026

1. **Claude Fable 5.1**: hardest design work, ownership, large refactors, and cases where Opus loses the thread.
2. **Claude Opus 5**: daily senior work, review, API design, CMake cleanup, test strategy, and smaller diffs.
3. **Codex with GPT-5.6 Sol**: red pipelines, repetitive repair loops, mechanical ports, and well-specified repository work.
4. **Grok 4.6**: independent review with the first model's answer hidden.

Rolling out now

GPT-6 Astra: the first model to test against Fable 5.1 on your own C++ repository as soon as it is actually available. Its vendor-reported coding results qualify it for the top tier, but the independent evidence is not complete.

The ranking that should survive the next launch

Which model writes the most C++ will keep moving. Fable 5.1 launched on September 1, and Astra rollout began on September 3. A numerical ranking can become stale within weeks.

The standard should not. Implementation got cheap. The scarce work in modern C++ is still:

- deciding whether the template belongs in the tree for ten years
- deciding whether developers can build on Windows and Linux after the CMake rewrite
- deciding whether two hundred tests encode the right behavior
- deciding who actually owns the object after the model reaches for `shared_ptr`
- deciding whether the smallest patch is actually the safest long-term design
- deciding what the agent must not change merely to obtain a green result

Evaluate assistants against that bar.

Today: Claude Fable 5.1 in the primary seat, with Opus 5 as the workhorse on the same Max plan.

Qualify next: GPT-6 Astra in Codex against your own hardest repository tasks.

Keep: Grok 4.6 as the model that never saw the first review.

Frequently asked questions

What is the best AI for senior-level C++ developers in 2026?

Claude Fable 5.1 is our current primary recommendation for difficult senior-level C++ work. Claude Opus 5 is the better everyday workhorse because it remains highly competitive and costs half as much per API token. No public benchmark directly proves either model is best at mature C++ engineering, so repository-specific evaluation still matters more than a generic leaderboard.

Is GPT-6 Astra better than Claude Fable 5.1 for C++?

Not yet established. OpenAI reports that Astra beats Fable 5.1 on several coding evaluations, but those are vendor-reported results. Astra did not yet have public results on the independent Senior SWE-Bench and Terminal-Bench leaderboards when this article was checked. Treat it as a qualified challenger, not a settled winner.

Which AI is best for CMake and GitLab CI?

Use Fable 5.1 or Opus 5 to reason about target boundaries, usage requirements, platform matrices, release policy, and what must remain invariant. Use Codex for the repetitive configure-build-test-debug loop after the constraints are explicit. The best result often comes from separating design judgment from mechanical execution.

Is there a C++ benchmark for ownership, ABI, CMake, and sanitizers?

We found no credible public benchmark that covers that complete combination. Senior SWE-Bench, CursorBench, and Terminal-Bench are useful proxies, but none directly measures mature C++ maintenance across ownership, templates, ABI, CMake, vcpkg, sanitizers, compilers, and platforms.

Should I use one model or several?

Use one primary model and one independent reviewer from a different model family. Showing the second model the first answer creates anchoring and reduces the value of the second opinion. For consequential architecture or lifetime decisions, independent convergence is much more useful than repeated agreement from the same family.

Methodology and sources

This ranking separates three kinds of evidence:

1. **Independent benchmark results,** where an organization other than the model vendor operates the leaderboard.
2. **First-party harness benchmarks,** such as CursorBench, which are useful but reflect the operator's task selection and agent environment.
3. **Vendor-reported launch results,** which are relevant evidence but should not be treated as independent validation.

Prices and availability were checked against current first-party product pages. Benchmark scores are dated snapshots and will change.

Primary and independent sources:

- Senior SWE-Bench leaderboard and methodology
- CursorBench 3.2 leaderboard
- CursorBench methodology
- Terminal-Bench 4.0 independent leaderboard
- Anthropic: Claude Fable 5.1 announcement
- Anthropic model documentation for Fable 5.1
- Anthropic Claude plan comparison
- Anthropic Fable usage on paid plans
- OpenAI: GPT-6 Astra announcement and benchmark tables
- OpenAI: GPT-6 availability in ChatGPT products
- OpenAI: ChatGPT Pro tiers
- SpaceXAI: Grok 4.6 announcement
- SpaceXAI: Grok Build
- SpaceXAI pricing
- CMake documentation
- CMake build-system and usage-requirements documentation
- Microsoft vcpkg manifest mode
- Microsoft vcpkg versioning
- Microsoft reference