ChatGPT

# Rust Code Base Quality vs. Quantity: Implications for LLM Training

## Rust's Code Base Size Compared to C++/C

Rust is a much younger language than C or C++, and accordingly its open-source code corpus is significantly smaller. Estimates in 2022 put the Rust developer community at around 2.8 million people – less than one-quarter the size of the C/C++ community [1]. This disparity is reflected in the volume of available Rust code. For example, in *The Stack* open-source code dataset used to train code models, Rust accounts for only about 40 GB of code, whereas C++ contributes roughly 193 GB [2]. Other model training corpora show a similar pattern: Rust code is on the order of only a few gigabytes, versus tens or hundreds of gigabytes for languages like C++, Java, or Python [2]. In short, Rust's accessible code base is an order of magnitude smaller than those of older, more entrenched languages.

This smaller corpus means that out-of-the-box large language models (LLMs) have had less Rust data to learn from. Indeed, evaluations of GitHub Copilot (powered by OpenAI Codex) found it **performed much worse for Rust** than for C++ or Java, largely because "Rust…does not have such a huge codebase" in the training data [2]. In contrast, C++ and Java have been around for decades and dominate open-source repositories, so models see far more examples of those languages [2]. All else being equal, **sheer data quantity has given C/C++ an advantage in current coding AI**, while Rust's relative scarcity has been a limiting factor.

However, raw volume is only one side of the story. The hypothesis we're examining is that **Rust's smaller code base might be higher in quality on average**, potentially offsetting some of the quantity disadvantage. To assess this, we need to look at what "quality" means in this context and how Rust code might differ from C/C++ code in practice.

## Quality Characteristics of Rust Code

Several factors suggest that Rust code tends to be **high-quality** or at least more consistent in certain best practices:

- **Language Safety and Bug Prevention:** Rust was explicitly designed taking lessons from C and C++ to avoid common programming errors [3]. The language's strict compiler and ownership model make it *harder to write buggy code*. Memory safety issues (like buffer overflows, use-after-free, null pointer dereferences) are largely eliminated in safe Rust. Microsoft noted that **~70% of all their security vulnerabilities (CVEs) stemmed from memory bugs**, a problem Rust "is famous for not allowing" [4]. This built-in safety net means a lot of the *"low-hanging fruit" of bad code (memory leaks, invalid pointer access, etc.) simply doesn't compile in Rust*. A developer with experience in both languages observed that when using Rust, they encounter *"much fewer bugs (no random segfaults or undefined behavior)" and their components end up "better designed"* [5]. In a real-world rewrite of a C++ codebase to Rust, the team reported they **no longer worry about out-of-bounds array accesses or arithmetic overflows**, issues which had been common in the C++ version [6]. (Rust panics on out-of-bounds and checks integer overflow in debug mode by

default.) In short, Rust's language design steers developers toward safer, more robust code, which should raise the overall quality of code that makes it into public repositories.

- **Modern Best Practices and Consistency:** Because Rust is a newer language (first stable release in 2015) with an active community, its ecosystem has benefited from modern best practices from day one. There is a strong culture of writing *idiomatic Rust*, aided by tools like `rustfmt` (for standardized formatting) and Clippy (a linter that catches common mistakes and enforces idioms). The compiler's strictness also forces handling of errors and edge cases that C/C++ might ignore. For example, one report noted that during a C++→Rust rewrite, developers were prompted to add checks (like validating UTF-8 string data) that the old C++ code "did no such check" for – **Rust refused to let them ignore these potential issues** [7]. Similarly, Rust will warn about unused variables or dead code; the rewrite project found that Rust's aggressiveness in flagging unused code led them to **eliminate 30–50% of the old code as dead logic** that had been lurking in the C++ codebase [8]. These factors mean that the Rust code found in repositories is more likely to be pared down to what's actually needed, with less bit-rotted or legacy cruft, simply because the language makes such cruft harder to ignore. In contrast, C and C++ code bases (especially older ones) often accumulate decades of inconsistent styles and leftover code that no one got around to deleting.

- **Experienced and Careful Developer Community:** Rust has a reputation for a steep learning curve, which has arguably resulted in a community skewing a bit more toward experienced developers and those who care deeply about code quality. Surveys have shown that many Rust users have prior professional programming experience in other languages [9] [10], and Rust is perennially the "most loved" language in Stack Overflow surveys (developers who try it often want to keep using it) [11]. One reason cited is that Rust "delivers on its promises" of reliability and performance [12]. The ecosystem also encourages best practices like rigorous testing and documentation – Rust's built-in test framework is easy to use (compared to C++ testing frameworks that "took so much time to even compile" [13]), lowering the barrier to writing tests for new code. On average, the **signal-to-noise ratio** of Rust's open-source code may be higher: it's often written by developers aiming for correctness and robustness from the start. Even in package management, the Rust community tends to be more conservative. As one discussion noted, the typical Rust project doesn't pull in "1000+ dependencies you've never heard of" as sometimes happens in Node.js/JavaScript; Rust developers are generally *more careful about adding dependencies*, partly due to the different (systems programming) mindset [14]. This cultural difference can translate to more review and scrutiny on the code that *is* published. The Rust crate registry (crates.io) has had far fewer incidents of malware or supply-chain attacks compared to npm; observers attribute this to Rust's smaller, more cautious developer base and the fact that Rust packages often run in less publicly exposed contexts [14]. Overall, these community factors mean Rust code available for training an LLM is more likely to be **serious, production-grade code** (systems tools, libraries, etc.) rather than throwaway scripts or trivial experiments.

- **Learning from C++'s Mistakes:** Many patterns that are now considered "bad practice" in older languages simply never took root in Rust. For instance, Rust has no preprocessor, so you won't find the kind of macro-based obfuscation that plagues some C/C++ projects. Manual memory management is rare (confined to `unsafe` blocks), so you avoid a whole class of error-prone pointer arithmetic patterns. The result is a more **uniform coding style** across projects. Idiomatic Rust favors clarity and safety (e.g. using `Result` for error handling, avoiding global state, leveraging the type system for invariants). By design, there's less variability in how fundamental tasks are done compared to C++, where one codebase might use raw pointers and another smart pointers, one uses exceptions for errors and another uses return codes, etc. This

consistency in Rust's "way of doing things" could be an advantage for LLMs – the model doesn't have to learn as many disparate patterns to cover the bulk of Rust code.

Of course, **"high quality" doesn't mean Rust code is perfect**. Rust doesn't prevent logical bugs, algorithmic inefficiencies, or poor architectural choices. One can still write messy or overly complex Rust code, especially when fighting the borrow checker – e.g. by littering `unsafe` blocks or cloning data gratuitously to sidestep lifetime issues. And not every Rust project is exemplary; beginners learning Rust may still produce less-than-idiomatic code (though the compiler and community norms push them to improve). However, on balance, the safeguards and culture of Rust do raise the floor for code quality. There is evidence that when the same project is implemented in Rust versus C++, the Rust version ends up simpler and more robust. For example, developers rewriting a network application from C++ to Rust observed that the Rust code was "much, much simpler and easier to reason about" even though the line count was similar [15]. The slight increase in Rust LOC came from explicitly handling errors that the C++ code had silently ignored – a net gain in correctness [15]. This illustrates how Rust's insistence on handling edge cases and errors leads to more reliable code, even if it's a bit more verbose.

In summary, **Rust's code corpus, while smaller, is arguably of higher average quality** in terms of modern coding standards: memory-safe, well-structured, with fewer outright broken snippets and outdated practices.

## C/C++ Code Base: Volume and Legacy Issues

To put Rust's quality advantage in perspective, consider the typical state of C and C++ code "in the wild." These languages dominate open source by sheer volume – C and C++ together have hundreds of gigabytes of code on GitHub [2] and millions of files [16]. But a large fraction of that corpus is **old, unmaintained, or low-quality code**. One analysis found that when attempting to compile millions of random C/C++ files from GitHub, **fewer than 1%** could compile successfully without extensive fixes [17]. In other words, an enormous amount of C/C++ code out there is effectively "junk" – incomplete projects, experimental fragments, or just broken code that never worked. Even many compiled projects contain significant dead code or backwards-compatibility hacks. A long-running C++ codebase might carry baggage from the 1990s up through C++20, with multiple generations of style intermixed. This can be *noise* for an LLM: the model has to ingest both good and bad examples, from elegant modern C++14 STL usage to archaic manual memory management with `malloc/free`.

Furthermore, C and C++ give programmers enough rope to hang themselves, and plenty of code bases have accumulated questionable patterns or subtle bugs. Memory unsafety in C/C++ is well-known – and indeed, those languages have caused countless security bugs (leading Google, Microsoft and others to begin migrating components to Rust for safety [4]). The presence of so much unvetted C/C++ code in training data can even degrade an AI model's outputs. It's been documented that code models like Codex sometimes regurgitate known insecure or bad practices that were present in the training set [18]. In one study, Copilot's suggestions were vulnerable or buggy ~40% of the time, likely because it had learned from the "ton of…junk" mixed into public code [19] [18]. In contrast, a smaller Rust corpus might act as a **curated set of code written with a more uniform philosophy of safety**, containing far fewer instances of, say, buffer overflow exploits or dangerous patterns.

Another point is that C++'s great flexibility results in highly heterogeneous code. There are many ways to do the same thing (multiple inheritance vs templates vs macros, etc.), and no enforced paradigm – one project might be strictly procedural C-style, another heavily object-oriented, another template-metaprogramming-heavy. This diversity is powerful but means an LLM has to generalize across very dissimilar coding styles under the umbrella of "C++". Rust, by contrast, has a more opinionated set of

idioms (for example, **ownership/borrowing is unavoidable**, functional error handling with `Result` is encouraged, etc.). That potentially makes it easier for a model to learn the core patterns despite having less data, because the Rust data is internally consistent to a greater degree.

In short, C/C++ offer *quantity* but with a lot of noisy data, whereas Rust offers *quality* with more homogeneous, up-to-date practices. The hypothesis is that for training AI, a high-quality dataset might punch above its weight compared to a larger, messier dataset.

## Evidence in LLM Performance and Behavior

How do these factors actually play out in today's coding assistants? The record is mixed. On one hand, current general-purpose LLMs still clearly perform better in languages where they saw more training examples. As mentioned, Copilot and Codex had noticeably lower success rates solving coding tasks in Rust than in C++, Java, or Python [2]. Rust was simply underrepresented in their training relative to those languages. Additionally, Rust's unique concepts (the borrow checker, lifetimes, etc.) pose challenges that an LLM might not fully grasp without very extensive training. Many users have observed that AI models often generate Rust code that **fails to compile due to borrow checker errors** or incorrect lifetimes. In fact, one research project specifically built a "RustAssistant" to use LLMs to *fix* Rust compilation errors, because even advanced models frequently introduce borrow/lifetime mistakes in Rust code generation [20]. The model can sometimes get stuck in a loop tweaking lifetimes and still not satisfy the compiler [20]. This shows that Rust's stricter correctness requirements can trip up an AI that hasn't perfectly internalized the borrowing rules. It's easier for an AI to generate a plausible-looking C++ snippet (even if it might have a subtle bug) than to generate Rust that the compiler will accept, because Rust has an extra layer of constraints to satisfy.

On the other hand, there are promising signs that **Rust's high-quality corpus imparts good practices to those models that do train on it**. Anecdotally, when a competent model like GPT-4 *does* produce Rust code, it often follows idiomatic patterns (for example, using `Option`/`Result` types correctly, leveraging iterators, etc., rather than writing Rust in a C-style manner). One user noted that AI helpers "show you the very common Rust pattern" and can handle boilerplate, making it easier to get started in Rust for someone who knows what they want to achieve [21]. Impressively, an LLM was able to **"connect the dots" on a complex and *rare* Rust feature (generic associated types) to meet the user's requirements**, providing a solution that neither Stack Overflow nor the official forums had readily available [21]. This suggests the model had absorbed advanced Rust knowledge from its training data – likely gleaned from high-quality Rust repositories or RFC examples, since GATs are a relatively recent and sophisticated feature. When it comes to more routine coding, models like OpenAI's have been used successfully by Rust developers to speed up mundane tasks. One commenter on Hacker News reported, *"LLMs have made me at least twice as fast at writing Rust code"*, saying that AI autocompletion helped with Rust's verbosity and showed idiomatic usage (though they did have to double-check the results) [22]. These experiences hint that **the Rust code which *is* available to LLMs is rich in "best practice" examples**, so models can learn the *right* way to do things in Rust (when they manage to do it at all).

Furthermore, specialized code models that include Rust are quickly improving. The open-source StarCoder model (15B parameters) was trained on the **The Stack** dataset covering many languages including Rust, and it has reasonably strong Rust capabilities. PolyCoder, another research model, explicitly included a Rust training subset. We are likely to see diminishing returns on simply adding more and more raw code data (especially if much of it is low-quality). Instead, **models curated for higher signal content could narrow the gap**. In other words, Rust's smaller-but-cleaner codebase might yield disproportionate benefits once the model is sophisticated enough to leverage it. A high signal-to-noise ratio means the model spends less capacity memorizing weird edge-case code or

deprecated patterns, and more on idiomatic Rust approaches. This could be one reason that, as of 2023, we started seeing models that, while not perfect, can produce non-trivial Rust code that adheres to Rust's strict rules and style. As Rust's popularity grows (it's one of the fastest-growing communities [23]), the volume of Rust in training corpora will also grow, hopefully continuing the trend of better Rust performance.

It's also worth comparing how **LLM outputs in Rust vs C++** might differ qualitatively. Given Rust's focus on safety, an AI writing Rust will naturally use safe libraries and idioms that avoid entire classes of bugs. By contrast, an AI writing C or C++ might, if guided only by frequency in training data, output code that compiles but is *not* good practice (e.g. forgetting to check a pointer for null, or using `gets()` for input because it saw that in old C code). In one study, Copilot was **prone to suggesting insecure C code** because it had seen so much of it in public repos [18]. Rust simply doesn't have an equivalent of `gets()` – the language and its standard library steer both humans and AIs toward safer constructs. So even if an LLM has less Rust data, the data it does have acts as a **collection of modern, vetted coding patterns**. In effect, Rust's design and community have done a lot of curation up front. An AI trained on crates.io code is learning from implementations that often went through rigorous code review and testing, whereas an AI trained on "all of GitHub C++" is learning from many throwaway projects and legacy systems along with the good code.

## Conclusion: Verifying the Hypothesis

**Is the existing Rust code base small but high-quality overall?** The evidence largely supports this hypothesis. Rust's accessible training corpus for LLMs is indeed relatively small – on the order of 1–2% of what a model like Codex or StarCoder sees for C++ or Python [2]. However, that corpus benefits from Rust's youth and the "lessons learned" baked into the language. The Rust code that AI models train on tends to be written with memory safety, correctness, and modern style in mind [3] [6]. Common bug patterns are prevented by the language or caught by the compiler, meaning the training data has far fewer examples of those pitfalls. Additionally, Rust's steep learning curve and systems programming niche mean its open-source contributors are often experienced programmers applying best practices. This all translates to a **higher average quality** of Rust code in the wild compared to the sprawling, uneven landscape of C/C++ code.

That said, we must be careful not to **overstate** the advantage this gives in practice. So far, quantity of data has proven extremely important for LLM performance. The models do not intrinsically "know" that Rust's data is better; they only improve if they see enough examples to generalize. Early results (like Copilot's LeetCode solver study) showed that insufficient Rust training data led to noticeably poorer results, despite Rust's inherent consistency [2]. Quality cannot fully compensate for lack of volume – an LLM can't learn a concept it's rarely or never seen. For instance, if an LLM hasn't encountered many examples of Rust's borrow checker constraints, it will make mistakes no matter how elegant the few examples in the dataset were. In those cases, the model's confidence may exceed its actual competence, leading to hallucinated Rust code that *looks* fine but doesn't compile or handle edge cases.

On balance, however, as models are fed more Rust (and they surely will be, given Rust's rising popularity), we can expect their Rust proficiency to rapidly improve – **and to do so in line with Rust's high standards**. Indeed, developers are already leveraging GPT-4 and other models to generate idiomatic Rust, and while they sometimes hit limitations, the output often reflects the language's best practices. We've seen LLMs propose correct usage of advanced Rust features and help find solutions that adhere to Rust's safety principles [21]. Moreover, the absence of decades of "bad old Rust code" (since Rust is new) means future models won't be bogged down by needing to unlearn antiquated

patterns. In contrast, an AI learning C++ has to reconcile code from C++98, C++11, etc., including lots of legacy anti-patterns, which could be a long-term maintenance burden in its parameters.

In conclusion, **the Rust code used to train coding LLMs can be characterized as small but overall high quality** – a product of Rust's modern design and community ethos. This has positive implications: as LLMs incorporate more Rust, they are imbibing a corpus rich in clean, safe coding practices rather than decades of deprecated habits. In theory, this should enable AI agents to "master" Rust in a way that aligns with what expert human Rustaceans would do. We are beginning to see that in practice, though current models are not perfect Rustaceans yet. The conservative take is that Rust's qualitative advantages *help* but do not wholly overcome the quantitative shortfall; we've witnessed that more data still often beats better data in today's models. Nonetheless, the gap is closing. Given Rust's growth and the demonstrated ability of models to learn even complex Rust idioms from limited examples, it's reasonable to expect that **future coding LLMs fine-tuned on Rust will produce very high-quality Rust code**. The training body of Rust, while modest in size, sets the model up for success by teaching it "the right way" from the start. In the long run, that may prove more valuable than sheer volume, resulting in AI solutions that mirror Rust's famed reliability and modern best practices.

**Sources:**

- BigCode Project, *The Stack* dataset statistics (Rust vs other languages) [2]
- InfoQ report on Rust ecosystem (developer community size and trends) [1] [11]
- Microsoft Security Response Center on CVE causes (memory safety issues) [4]
- KDAB Whitepaper on Rust vs C++ (Rust avoids common C/C++ errors by design) [3]
- Philippe Gaultier, "Lessons from a Rust Rewrite" (experiences of cleaner, safer code after migrating from C++) [6] [7]
- Developer anecdote on Rust having fewer bugs and better design [5]
- Reddit discussion on crates.io vs npm (Rust devs more cautious with dependencies) [14]
- Hacker News discussion on AI with Rust (LLM usage of advanced Rust features like GATs) [21]
- StartupSoft analysis of code corpora quality (open-source code data can be noisy) [19] [17]
- Academic study of Copilot on LeetCode (Rust performance vs C++/Java) [2]

---


[1] [9] [10] [11] [12] [23] Rust Reviewed: the Current Trends and Pitfalls of the Ecosystem - InfoQ
https://www.infoq.com/articles/rust-ecosystem-review-2023/

[2] GitHub Copilot: the perfect Code compLeeter?
https://arxiv.org/html/2406.11326v1

[3] KDAB's Software Development Best Practices: | KDAB Whitepapers
https://www.kdab.com/publications/bestpractices/best-practices-hybrid-rust-cpp-apps.html

[4] [5] Understanding Rust as a C++ developer - DEV Community
https://dev.to/daaitch/understanding-rust-as-a-c-developer-2o28

[6] [7] [8] [13] [15] Lessons learned from a successful Rust rewrite
https://gaultier.github.io/blog/lessons_learned_from_a_successful_rust_rewrite.html

[14] Is using crates more safe than using npm? : r/rust
https://www.reddit.com/r/rust/comments/qdqek2/is_using_crates_more_safe_than_using_npm/

[16] [17] [18] [19] Comparative Guide to Human-Written Code Data Sources for Training LLMs - StartupSoft
https://www.startupsoft.com/guide-to-human-code-data-sources-for-training-llms/

20 21 22 RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code
https://news.ycombinator.com/item?id=43851143