ChatGPT

Rust’s Current Gaps and Potential Enhancements

Rust already excels at low-level performance and memory/concurrency safety, but like any language, it omits certain features that other languages provide. In particular, Rust’s *type system* and *language semantics* lack some advanced features (e.g. dependent types, variadic generics, runtime reflection) that could further increase expressiveness. Its **memory model** uses strict ownership but does not natively support alternatives like garbage collection or “generational” references (safe use-after-free pointers) found in some newer languages. Rust has *no runtime reflection or dynamic typing*, so common patterns in Java/Python (introspection, dynamic loading) aren’t built-in. The error model in Rust deliberately excludes exceptions and hidden control flow, which improves safety but means Rust cannot mimic exception-based frameworks in Java/C++. Likewise, Rust lacks built-in **range-checked integer types** (as in Ada) or dependent types, which would let the compiler encode and verify more invariants [1] [2]. In short, Rust’s core language is powerful but conservative; it could borrow features like *range constraints, static taint analysis*, or even *dependent typing* (e.g. for compile-time proofs) from languages such as Ada/SPARK or Idris to catch more errors early [1] [2].

- **Advanced Type System**: Rust recently stabilized *generic associated types* and is working on *const generics*, but it still has no *variadic generics* (needed for arbitrary tuples/ecs queries) or *specialization* (trait overrides) in stable Rust. Dependent types or function contracts (pre/post-conditions) are not native, though a formal effort is underway to add **function/loop contracts** for verification [3]. These would allow encoding invariants (like “length ≥ 0” or “pointer non-null”) checked at compile time. In contrast, languages like **Ada/SPARK** have built-in range types and contracts, and theorem-proving languages (Coq, Idris) use dependent types for formal guarantees [1] [3].

- **Memory Management Variants**: Rust’s ownership/borrowing model avoids garbage collection, but some workloads benefit from optional GC. *Alternative memory schemes* could be added: for example, **generational references** (a form of safe handles) which detect stale pointers, as in Vale [4]. Vale’s design shows how Rust-like performance could be combined with automatic free detection: its “generational references… are able to detect when an object has been deallocated” and thus abort on invalid use rather than cause UB [4]. Rust currently requires unsafe code for raw memory/FFI, so it cannot isolate bugs in unsafe modules. Some propose *“Fearless FFI”* or sandboxing (like Vale’s model) to separate safe and unsafe memory, preventing a bug in one from corrupting the other [5] [6].

- **Concurrency Models**: Rust’s approach (threads or `async`/`await`) guarantees no data races, but still allows locks and panics, so deadlocks and other high-level concurrency bugs are possible. Other languages offer different paradigms: e.g. **Pony** uses an actor model with reference capabilities to eliminate locks entirely. Its tutorial boasts “no locks or atomic operations… the type system ensures at compile time that your program can never have data races” and notes that “Pony has no locks at all… they definitely don’t deadlock” [7]. Rust could consider built-in actor-like primitives or stronger type-level capabilities (the *Send*/*Sync* traits are a start) to mirror this guarantee. Similarly, Go’s goroutines and channels provide easy concurrency but rely on garbage collection and have a built-in race detector; Rust has none of those by default, so adding static data-race analysis or easier task scheduling (like a first-class async runtime) could be beneficial.

- **Metaprogramming and Reflection**: Rust has powerful macros but no runtime reflection. This means you cannot iterate over struct fields or easily do generic serialization without custom code. Languages like Java, C#, or even Swift allow runtime introspection of types and fields. While Rust's focus is on zero-cost abstractions (and indeed it has minimal runtime overhead by design), one could envision *limited reflection*: for example, compiler plugins that auto-generate code or metadata, or a future `#[derive]`-style mechanism that exposes type information. At minimum, many request a better "debuggable" environment (improved REPL or RTTI) for dynamic inspection, which Rust currently lacks.

Compiler, Runtime, and Tooling Enhancements

Beyond language syntax, Rust's compiler and tooling could add new **safety and security analyses**, and performance aids. Static analysis tools (Clippy, Rustc) already catch many errors, but there is room to grow:

- **Formal Verification Integration**: There's an active push to incorporate formal methods into Rust. For example, AWS and academic groups are using tools like **Prusti, Kani, Verus, Miri**, etc., to prove Rust code correct [8]. The Rust core team is even adding *contracts* (pre/postconditions) to Rust's future releases [3]. A new "Rust verification challenge" crowdsources proofs that std library functions have no undefined behavior [9]. Realistically, future Rust versions might include **optional specification annotations** or tighter compiler checks for unsafe blocks, making it easier to formally verify entire programs.

- **Enhanced Lints & Scanners**: Security-savvy languages often provide built-in or extensible scanners for common flaws. Rust could bolster its linting: for instance, Clippy could gain rules for constant-time crypto or ban insecure functions (like MD5). The SEI report notes that Rust's language cannot prevent use of broken algorithms (RustCrypto includes MD5) [10]. Having default lint rules to discourage such patterns, or even a "safe by default" cryptographic library, would raise the security bar. Likewise, tools like CodeQL or Semgrep for Rust (now emerging [11]) could be officially supported.

- **Sanitizers and Runtime Checks**: Rust benefits from LLVM's Sanitizers (ASan/UBSan), but integration could be smoother. A future Rust could automatically instrument code for *control-flow integrity (CFI)* or *memory tagging* (ARM MTE, Intel CET) to catch attacks. On the performance side, support for *profile-guided optimization (PGO)* and tools like BOLT (for binary optimization) are available, but could be more accessible through `cargo` commands [12]. Enabling easier use of address-sanitizer, thread-sanitizer, or even a built-in *fuzzer* driver in `cargo test` would aid security audits. Also, better leak detection (Rust doesn't treat leaks as "unsafe" but leaks can be serious) might be added: for example, a lint or tool to flag reference cycles or non-`Drop` memory, which Rust currently leaves unchecked [13].

- **Embedded and Low-Level Support**: For systems/embedded use, Rust could improve direct hardware interfacing. Currently, reading arbitrary addresses or using CPU instructions requires `unsafe`. A more fine-grained model (like *memory-mapped I/O abstractions*) or language support for *hardware capabilities* (e.g. sandboxing between user code and I/O) could be envisioned. Similarly, while inline assembly (`asm!`) is stabilized, cross-architecture SIMD support is still incomplete: stable Rust exposes some vendor intrinsics but lacks *portable SIMD* on stable [14].

The Rust SIMD RFC notes "a notable gap" is explicit SIMD access [14]; fully stabilizing wide vector types or making auto-vectorization more predictable would boost performance for numeric code.

- **Developer Tooling**: Rust's fast progress brings complexity. Tools could be improved: for example, automatic **code coverage** integration (the forum suggests better coverage reports like Coveralls [1]). Official support for IDE debugging (beyond externals) or hot-reloading (like Go's rapid rebuilds) would improve developer confidence and thus indirectly security/quality. A built-in REPL (`evcxr` exists unofficially) or simplified compilation (like single-file scripts) could lower the barrier for safe experimentation.

Ecosystem, Supply-Chain, and Security Hardening

Rust's package ecosystem is maturing, but more **supply-chain security** features could be added. Key ideas include:

- **Crate Signing and Verification**: Work is underway to integrate Sigstore-like signing for `crates.io`. A Rust internals proposal outlines using Sigstore keys so that crate downloads can verify author signatures [15] [16]. In practice, this would let users require that a crate is signed by known maintainers, preventing man-in-the-middle or typosquatting attacks. (Similar efforts in npm/PyPI have shown the importance of cryptographic provenance.) Fully automatic `cargo` support for fetching signatures and transparency logs would make supply-chain attacks much harder.

- **Vulnerability Auditing and "Cargo Vet"**: Rust already has `cargo-audit` for known CVEs and `cargo-vet` for auditing licenses. But these remain optional. A more automated system (e.g. default CI checks) could be provided by the toolchain. For example, on every `cargo publish`, crates.io could run vulnerability scans or enforce that certain lints pass. The ECOSYSTEM could also introduce **trusted registries** or snapshots (like Stackage in Haskell) for high-assurance projects. This is analogous to **apt/yum** signing in Linux or NPM audit modes: Rust could add an official audit mode in `cargo` that checks crates' signatures, provenance, and known issues at install time.

- **Standard Library Hardening**: The Rust standard library and core crates (alloc, std, etc.) rely heavily on `unsafe` code [17]. Ongoing efforts to formally verify these routines (AWS's "Rust stdlib verification" challenges) may lead to safer implementations. In the future, the Rust toolchain might include optional safety modes that, for example, abort on out-of-bounds or integer overflow even in release builds (beyond the current debug panics). While this incurs cost, it would catch a wide class of bugs. Some have suggested a "SAFE RUST" profile (akin to MISRA or CERT C profiles) that prohibits certain unsafe patterns or enforces stricter checks. In fact, an upcoming MISRA C addendum is evaluating how to apply such rules to Rust [18]. Official guidance (and perhaps a linter) for "safety-critical Rust" would help industries like aerospace/auto adopt Rust with confidence.

- **Dependency Hygiene and Policies**: Rust could adopt policies like "no-unsafe by default": crates might be required to clearly document or warn about unsafe code. Tools could flag crates that use `unsafe` excessively. Similarly, Rust could allow "deny-unsafe" as a crate attribute to refuse compilation of unsafe blocks, easing code reviews. Even sandboxing crate execution via WebAssembly or containers (a planned Vale feature) would contain potentially malicious or buggy code. In short, Rust's ecosystem can grow features to ensure supply-chain trust, far beyond its current state.

Comparisons with Other Languages

- **C and C++:** Rust was designed to match C/C++ performance while adding safety. It omits **C++ templates** in favor of generics, which limits certain patterns (no specialization or const-eval type computations as easily). Rust has no preprocessor or function overloading, nor default args – these make some C++ libraries more concise. On the other hand, C++ has *exceptions* and *runtime polymorphism*, which Rust deliberately avoids. If anything, Rust could consider "zero-cost abstractions" in more domains: e.g. adding something like optional panic-handling strategies (current panics unwind by default, or abort).

- **Go:** Go's simplicity comes from garbage collection and built-in tools. It has goroutines/channels and a race detector in the runtime. Rust's `async` model is more manual (no built-in executor) and has no automatic GC or race checking. Go's compile times are very fast and cross-compilation trivial – Rust is improving (with `cargo bloat`, incremental compilation), but not quite as instantaneous. Some have suggested learning from Go: e.g. **official async runtime** (instead of relying on Tokio or others), a built-in race detection mode, and even optional GC for certain programs (though Rust's community is cautious about this for security/performance reasons).

- **Zig:** Zig is a new systems language that, like Rust, avoids hidden runtime. It offers easy cross-compiling and compile-time reflection (`comptime`). Rust has *const fn* and *macros* for computation, but Zig's `comptime` is arguably more powerful and ergonomic. Rust could integrate more compile-time introspection (e.g. allowing types to be manipulated in `const fn` or macro more flexibly). Zig also emphasizes safety via simplicity; Rust already has that, but adopting Zig's single-stage compiler model could simplify build pipelines.

- **Swift/Java/C#:** These high-level languages all provide runtime features Rust lacks. For instance, Java/Swift have **garbage collection/ARC**, exceptions, and extensive **reflection**. They trade performance and determinism for ease of use. Rust gains performance by eschewing GC, but at the cost of boilerplate (managing lifetimes). One possible compromise is optional reference counting (Rust has `Rc` and `Arc`, but they are manual). Auto-ARC (like Swift) isn't in Rust, but could be a future library feature. C#/.NET offer JIT compilation and a huge standard library; Rust could bridge more with high-level runtimes (e.g. compiling to .NET or JVM for sandboxing, though that's speculative).

- **Python/JavaScript:** Scripting languages excel at rapid prototyping, dynamic typing, and huge ecosystems. They have **interactive consoles** and **dynamic code loading**. While Rust doesn't need these for performance, adding an official REPL or simpler scripting mode (single-file execution) could attract more developers. Python's `async` (via `asyncio`) and JS's event loop show different concurrency models; Rust's `async` is on par in principle, but lacks ergonomic language syntax (async closures, async trait methods are still in development). Drawing inspiration from these, Rust might later support *async functions in traits* or *generator state machines* more seamlessly.

- **Pony and Erlang:** These emphasize fault-tolerant concurrency (actors) and safe message-passing. Rust's standard library has no actor model, but crates like Actix exist. Studying Pony/

Erlang could inspire built-in features (e.g. mailbox types, guaranteed delivery primitives) that maintain Rust's safety.

- **Ada/SPARK and MISRA Languages:** For embedded/safety-critical domains, Ada and SPARK provide **formal contracts** and forbid unsafe operations. Rust is on the path (no nulls, no data races by default), but it could adopt Ada-like strong typing (subranges) or certify toolchains. For instance, SPARK's proof tools show how to eliminate entire bug classes; similar tools for Rust (like the aforementioned Prusti, Verus) could be first-class. The emerging MISRA C guidelines for Rust [18] suggest Rust will gain a "defensive" style checklist, potentially adding built-in annotations to satisfy ISO 26262 or similar standards.

- **Experimental Languages (Vale, Dhall, etc.):** Vale (not to be confused with Project Everest's VALE) is pushing *completely memory-safe systems programming* via generational references and "region isolation" [4] [5]. Such ideas could feed back into Rust: e.g. a future Rust borrow checker variant with "regions" or a hybrid with run-time checks (Vale's "check override" operator). Similarly, safe DSLs like Dhall (pure config) influence the idea of guaranteed pure computations, which Rust could adopt in certain modules to eliminate I/O side effects.

Vision: Future Security and Performance Features

Looking ahead, the Rust community is already planning many enhancements. For security and safety, we can expect **stronger verification and contract support** – the Rust team is "working with the language team to introduce function and loop contracts" into main Rust [3], enabling static checking of invariants. We'll likely see *official support for formal tools*: Prusti or Verus may eventually integrate into `cargo` or the compiler (so you can write specs alongside code). Package security will improve via **Sigstore-based signing of crates** (an active proposal) and automated auditing (e.g. requiring crates to pass `cargo-audit` or policy checks before publishing). On performance, Rust is trending toward better SIMD and parallelism support [14] (portable SIMD types are in development), more aggressive LLVM optimizations, and easier Profile-Guided Optimization. Build times are a perennial concern, so incremental and parallel compilation improvements will continue (perhaps even JIT-driven components).

In the realm of **hardware and sandboxing**, future Rust might leverage hardware protections: for instance, automatic pointer tagging (to catch out-of-bounds) or alignment checks. WebAssembly is already used as a sandboxed target, but one can imagine better WASM integration for embedded Rust or safe plugin execution. In embedded systems, Rust could add *heap exhaustion checks* (in place of panics) or cooperative scheduling primitives for real-time safety.

Finally, Rust's philosophy of "safety, performance, productivity" means any new feature must balance those. We may see **selective enabling of extra safety** (e.g. an Opt-in "safe" mode that runs extra checks) and **opt-in performance modes** (like Vale's idea of a release-mode "override" to skip checks). The goal will be to let developers choose the right level of guarantees: from blazing-fast "no-UB" Rust to "formally verified" Rust. In sum, Rust's future looks to expand on its strengths – adding static analyses, cryptographic assurances, and high-level abstractions – while learning from languages across the spectrum (C, Go, Ada, Pony, etc.) to close its remaining gaps in safety, security, and ease of use [3] [7].

Sources: Technical analyses of Rust's safety model and ecosystem [19] [20] [10]; Rust RFCs and project goals (SIMD support, contract specs) [14] [3]; language comparisons and proposals (Pony, Vale, forum discussions) [7] [4] [1]; and AWS and Rust Foundation blogs on verification efforts [17] [3].

---

[1] [2] What makes Rust a high security language? What could be done better? - The Rust Programming Language Forum
https://users.rust-lang.org/t/what-makes-rust-a-high-security-language-what-could-be-done-better/9879

[3] [8] [9] [17] Verify the Safety of the Rust Standard Library | AWS Open Source Blog
https://aws.amazon.com/blogs/opensource/verify-the-safety-of-the-rust-standard-library/

[4] [5] [6] The Most Memory Safe Native Programming Language
https://vale.dev/memory-safe

[7] Pony Tutorial
https://tutorial.ponylang.io/index.html

[10] [13] [19] [20] Rust Software Security: A Current State Assessment
https://www.sei.cmu.edu/blog/rust-software-security-a-current-state-assessment/

[11] Rust security scanning options : r/rust
https://www.reddit.com/r/rust/comments/14ekt42/rust_security_scanning_options/

[12] Optimizing Rust programs with PGO and BOLT using cargo-pgo
https://kobzol.github.io/rust/cargo/2023/07/28/rust-cargo-pgo.html

[14] 2325-stable-simd - The Rust RFC Book
https://rust-lang.github.io/rfcs/2325-stable-simd.html

[15] [pre-RFC] Using Sigstore for signing and verifying crates - tools and infrastructure - Rust Internals
https://internals.rust-lang.org/t/pre-rfc-using-sigstore-for-signing-and-verifying-crates/18115

[16] Rust - Sigstore
https://docs.sigstore.dev/language_clients/rust/

[18] MISRA vs CERT · Issue #75 · rustfoundation/safety-critical ... - GitHub
https://github.com/rustfoundation/safety-critical-rust-coding-guidelines/issues/75