ChatGPT

Aero: A High-Performance, Ergonomic Programming Language

Aero (from *Aero+*, meaning swift and lightweight) is a new compiled systems programming language designed to combine the raw performance of C++ with the programmer productivity and clarity of Python. It emphasizes *performance and zero-cost abstractions* (following C++'s "you don't pay for what you don't use" principle [1]) while upholding code *simplicity and readability* à la the Zen of Python ("Explicit is better than implicit; Readability counts" [2]). Aero's philosophy also mirrors languages like Go and Rust: Go was created to "combine the simplicity... of Python with the efficiency and performance of... C++" [3], and Rust's core principles are *safety, performance, and concurrency* [4]. Aero's design manifesto, therefore, is to provide **zero-cost performance and strong safety without sacrificing developer ergonomics**.

- **Explicit Simplicity**: Language syntax is clear and concise. *There should be one obvious way to do it* (Zen of Python [2]), with no hidden control flows or magic. Default semantics favor safety and clarity.
- **Zero-cost Abstractions**: All high-level features compile down to efficient machine code. Aero follows C++'s *"zero-overhead"* principle [1], ensuring abstractions (generics, iterators, async) impose no hidden runtime cost.
- **Safe Systems Control**: Memory and concurrency safety are first-class. Aero uses an ownership/borrowing model (inspired by Rust [4] [5]) to prevent data races and dangling pointers at compile time, eliminating the need for a garbage collector and avoiding its performance overhead [6] [5].
- **Modern Concurrency**: Concurrency primitives (lightweight tasks, message-passing channels, async/await) are built-in and ergonomic. Unlike raw threading, Aero encourages models like CSP or actors where possible [7] [8], avoiding complex lock management (see diagram below).
- **Compositional Design**: Aero moves away from heavy class-based OOP. It favors *composition over inheritance*, with structs (plain data records), algebraic sum types, and *trait-like interfaces* for polymorphism. This yields a flatter, more predictable design.
- **Strong Static Typing with Type Inference**: Aero has a robust static type system with generics, ADTs, and inference. The compiler catches mismatches early, yet local variables and function return types can often omit explicit types for brevity.
- **Ergonomic Tooling**: The language comes with built-in support for IDE integration (LSP), a package manager, and formatting tools, ensuring a modern developer experience out of the box.

Together, these principles form Aero's **design manifesto**: to enable writing high-level, readable code that runs as efficiently as hand-optimized C++, while reducing programmer error through strong types and safety checks [4] [2]. Like Rust, Aero aims for a *"fast, reliable, concurrent"* systems language [4].

Core Language Constructs

Syntax Overview (EBNF)

Aero's syntax is C-like with optional semicolons and braces, balancing familiarity and readability. It uses `let` / `var` for variable declarations (immutable vs mutable) and `func` for functions. Indentation is allowed but not required (unlike Python). A sample grammar snippet (in simplified EBNF) illustrates key constructs:

```
<program>        ::= { <import_stmt> } { <decl> }
<import_stmt>    ::= "import" <identifier> ("." <identifier>)* [" as "
<identifier>] ";"
<decl>           ::= <func_decl> | <struct_decl> | <enum_decl> | <const_decl>
<func_decl>      ::= "func" <identifier> "(" [ <param_list> ] ")" [ "->"
<type> ] <block>
<param_list>     ::= <param> { "," <param> }
<param>          ::= <identifier> ":" <type>
<struct_decl>    ::= "struct" <identifier> [ "<" <type_params> ">" ] "{" {
<field_decl> } "}"
<field_decl>     ::= ( "let" | "var" ) <identifier> ":" <type> ";"
<enum_decl>      ::= "enum" <identifier> "{" <enum_variant> { ","
<enum_variant> } "}"
<enum_variant>   ::= <identifier> [ "(" <type_list> ")" ]
<const_decl>     ::= "const" <identifier> ":" <type> "=" <expr> ";"
<type_params>    ::= <identifier> { "," <identifier> }
<type>           ::= <identifier> [ "<" <type_list> ">" ]
<type_list>      ::= <type> { "," <type> }
<block>          ::= "{" { <statement> } "}"
<statement>      ::= <decl> | <expr_stmt> | <if_stmt> | <match_stmt> |
<return_stmt> | <while_stmt>
<expr_stmt>      ::= <expr> ";"
<if_stmt>        ::= "if" <expr> <block> [ "else" <block> ]
<match_stmt>     ::= "match" <expr> "{" { <pattern> "=>" <expr> "," } "}"
<return_stmt>    ::= "return" [ <expr> ] ";"
<while_stmt>     ::= "while" <expr> <block>
<expr>           ::= <literal> | <identifier> | <expr> <binop> <expr> |
<unop> <expr>
                   | <identifier> "(" [ <arg_list> ] ")" | "(" <expr> ")"
<arg_list>       ::= <expr> { "," <expr> }
```

This grammar shows functions with optional return types, generic type parameters, and an explicit **match** expression for pattern matching on enums, akin to Rust or Scala. Statements and declarations are clearly separated (no automatic semicolon insertion). Aero's syntax strives for clarity: keywords are mostly full words (`func`, `struct`, `enum`), and blocks use braces. *Indentation is supported for readability but not semantically significant.*

Static Type System

Aero features a **strong static type system** with type inference. All variables and expressions have compile-time types; common cases (e.g. `let x = 10`) infer types from context. Built-in basic types include `bool`, `int` (32- or 64-bit), `float` (64-bit), `char`, and `string`. Aero also provides tuple types (e.g. `(int, string)`), fixed-size arrays (e.g. `[int; 10]`), and dynamic sequences (`List<T>`, `Map<K,V>`).

Polymorphism is supported via **generic types** and **traits** (interfaces). For example, one can write a generic function:

```
func max<T: Comparable>(a: T, b: T) -> T { ... }
```

Here `T: Comparable` is a trait bound requiring type `T` implement the `Comparable` interface (similar to Rust's traits or Go's interfaces). Structs can also be generic and implement traits. This trait-based polymorphism, combined with pattern matching on enums, covers most use cases of classes and inheritance without traditional OOP overhead. In particular, Aero **eschews inheritance hierarchies**; instead, *composition* is used for code reuse. Data structures are typically defined as simple `struct`s with fields and methods, or as `enum` sum-types with variants.

Control Flow and Memory Safety

Control flow constructs include `if/else`, `while`, `for`-loops, and a powerful `match` expression for exhaustive pattern matching on enums (with support for guards). Aero enforces *no hidden side effects*: all references and pointer-like types are clearly annotated. By default, variables are immutable (`let`); mutable variables (`var`) must be explicitly marked. This design ensures that aliasing and mutation are always explicit, aiding reasoning and optimization.

Crucially, Aero's type system incorporates an **ownership and borrowing** discipline (inspired by Rust) to manage memory without a garbage collector. Each heap-allocated value has a single *owner*; when the owner goes out of scope, the memory is freed immediately. References can be taken either as immutable (`&T`) or mutable (`&mut T`), but rules prevent data races and dangling pointers. For example, at compile time Aero ensures you cannot have two simultaneous mutable references to the same data, and you cannot use data after its owner is dropped. These rules are enforced by the type checker (zero runtime cost) [5]. The result is deterministic, RAII-style destruction of objects, just like in C++, but with automated checks. In practice, this means Aero programs have C++-level performance and no runtime GC pauses, while also avoiding common bugs like use-after-free or data races [5] [6].

Standard Library Philosophy

Aero's **standard library** focuses on essential functionality while remaining lean. It includes basic data structures (vectors, hash maps, strings), I/O, threading primitives, and math routines. Concurrency support (threads, channels, futures) is built in. Unlike Python's "batteries included" approach [9], Aero's core library is minimalist by design: it provides only what systems programmers truly need, avoiding large monoliths. More specialized capabilities (networking, GUI, numeric computing, etc.) are delivered through separately versioned packages (via the package manager). This balances rich functionality with quick compilation and

small runtimes. All standard library code is written in Aero for consistency and safety; when necessary, low-level OS or C interop is allowed in controlled “unsafe” blocks.

Memory and Concurrency Model

Aero’s memory and concurrency models are designed in tandem to enable **fearless concurrent programming**.

- **Memory Model (No Garbage Collector)**: Aero uses explicit ownership and scope-based allocation. There is *no garbage collector*; instead, Aero tracks resource lifetimes at compile time. As with Rust, every value has a unique owner and is dropped deterministically [5]. Functions either take ownership of their arguments or borrow them temporarily. For shared resources, Aero provides optional reference-counted pointers (`Rc<T>` or `Arc<T>` for thread-safe ref-counting) and scoped locks, but these are used explicitly. This eliminates GC overhead and unpredictability: “the dual benefit [is] (1) improving runtime performance by avoiding garbage collection, and (2) improving predictability by preventing accidental ‘leaks’ of data” [5]. The language also includes compile-time facilities for defining custom memory allocators or using arena allocators for special domains (embedded or real-time). Overall, Aero achieves memory safety *without GC*, relying on zero-cost abstractions and ownership checks [10] [6].

- **Concurrency Model**: Aero provides **built-in concurrency primitives** and encourages models that avoid shared mutable state. By default, Aero threads do *not* share memory arbitrarily; shared data must be wrapped in explicit synchronization or messaging constructs. In practice, Aero follows a message-passing or CSP style (like Go’s goroutines and channels) alongside an optional actor library for isolated state. For example, one can spawn lightweight tasks (`spawn { ... }`) that communicate via channels or futures. The language also supports async/await syntax for asynchronous I/O and tasks, with a runtime scheduler. Internal threads are multiplexed onto OS threads by the runtime, allowing millions of concurrent “green” threads.

Figure: The parallel worker model with shared state. Managing locks and avoiding race conditions is complex, as every thread may contend for shared data [11]. Aero’s default concurrency model avoids this by using message passing or scoped tasks instead.

In contrast to naive shared-memory threading, Aero’s model treats communication as isolated “actors” or channels. In the **actor model**, each actor has private state and reacts to messages [7]. Alternatively, Aero’s channels (inspired by Go) convey ownership of messages, preventing implicit sharing: “channels in any programming language are similar to single ownership… Shared-memory concurrency is like multiple ownership… Rust’s ownership rules greatly assist in getting this management correct” [8]. This means data races are caught at compile time. Aero also provides atomic types and mutexes for the rare cases where manual locking is needed, but these are discouraged in favor of structured concurrency. Overall, Aero ensures *concurrent safety*: its ownership system and type checker prevent data races by construction [12]. Like Rust’s “fearless concurrency” philosophy, developers can write high-performance parallel code that the compiler guarantees to be race-free [12] [4].

Metaprogramming Features

Aero offers powerful metaprogramming while keeping code analysis tractable. Core metaprogramming capabilities include:

- **Generics and Type Constraints**: As noted, Aero has fully generic types and functions. Generics are reified (monomorphized) at compile time, so no runtime type information or boxing is needed. This yields performance akin to C++ templates but with safer syntax. Aero's traits can have default method implementations, and the compiler can derive implementations (e.g. equality, ordering) for simple structs automatically.
- **Compile-Time Functions and Macros**: Aero includes a hygienic macro system similar to Rust's `macro_rules!`. Macros can generate code at compile time, enabling domain-specific syntax or boilerplate elimination. Additionally, Aero supports a limited form of compile-time function evaluation (CTFE) for constant expressions and generics, so some computations can be done at compile time (e.g. computing array sizes, string concatenation, static assertions).
- **Reflection and Annotations**: While Aero does not have full runtime reflection (to maintain static guarantees), it allows compile-time reflection of type metadata for serialization and code generation. For instance, one can annotate a struct with attributes (e.g. `@serialize`) and the compiler will auto-generate code to serialize its fields. These annotations are expanded at compile time by the compiler or via plugin macros.
- **Tooling for Code Generation**: The compiler toolchain includes an integrated DSL for code generators (e.g. protocol buffers, GUI builders). These run as part of the build process and emit Aero source before compilation, ensuring seamless integration.

Metaprogramming is a core enabler of Aero's ergonomics: it allows libraries to provide high-level abstractions (ORMs, UI frameworks, serialization, etc.) without sacrificing performance, since generated code is compiled away. All macros and compile-time features are optional and explicit, ensuring they do not interfere with simple code if not used.

Compiler Architecture and Implementation Roadmap

Aero is designed for high efficiency in both compiled code and the compilation process itself. The compiler architecture comprises the following stages:

1. **Front End**: Parses Aero source files (using the above EBNF) and performs semantic analysis. A modern parser-generator (or handwritten parser) produces an Abstract Syntax Tree (AST). The compiler then resolves names, checks types and borrow rules, and expands generics and macros. Any semantic errors (type mismatch, borrow violation, etc.) are reported with helpful messages.
2. **Intermediate Representation (IR)**: The compiler lowers the AST into a typed IR, which is designed to facilitate optimizations. Aero's IR supports SSA form, control-flow graphs, and representation of ownership/borrowing lifetimes.
3. **Optimizations**: Various optimizations run on the IR: inlining, dead code elimination, monomorphization of generics, loop unrolling, escape analysis (to allocate some data on stack instead of heap), etc. Aero's design emphasizes that all optimizations should be "zero-cost" – if you didn't write it explicitly, it shouldn't slow you down.
4. **Back End**: The IR is translated into efficient machine code. Aero will initially target LLVM as a backend for wide platform support (x86-64, ARM, etc.), benefiting from LLVM's mature optimizer and

codegen. Eventually, a custom back end could be developed for specialized targets (embedded, game consoles). The runtime is minimal (just threading and I/O libs, also compiled ahead-of-time).

Bootstrapping: The first implementation of Aero will be written in an existing language (likely C++ or Rust) to get a working compiler quickly. Over time, the compiler will be self-hosted: future versions will be written in Aero itself. Bootstrapping ensures that developers can contribute using familiar languages before the language matures. The implementation plan follows these phases: *Design → Minimal Prototype → Self-Host → Stabilize → Optimize*. This is similar to how other systems languages like Zig or Swift evolved.

Tooling Ecosystem

To ensure high developer productivity, Aero comes with a robust suite of tools, many of which are designed from day one (in contrast to retrofitting). Key tooling includes:

- **Language Server (LSP)**: Aero provides a Language Server implementing the Language Server Protocol (LSP) [13] . This gives editors (VS Code, Neovim, IntelliJ, etc.) support for code completion, go-to-definition, hover documentation, and refactoring. By standardizing on LSP [13] , Aero ensures that most code editors can support it with minimal effort.
- **Package Manager**: Aero will have an integrated package manager (tentatively called `aero-cargo`) for dependency management, building, and publishing libraries. It will use a central repository with semantic versioning, inspired by Rust's Cargo or Go modules. The package manager will enforce reproducible builds and sandboxed compilation to avoid "dependency hell."
- **Formatter and Linter**: A built-in code formatter (like `go fmt`) enforces a single canonical style, making code reviews simpler and codebases uniform. A linter tool provides warnings for common pitfalls (e.g. unused variables, unchecked errors). These tools are distributed as part of the compiler toolchain.
- **Documentation Generator**: Aero includes a documentation generator that produces HTML docs from source annotations (e.g. comments). Similar to Rust's `rustdoc` or JavaDoc, it builds an API reference site automatically.
- **Build System Integration**: While the package manager handles most builds, Aero will also provide Makefile or CMake integration for projects mixing Aero with C/C++ code.

In short, Aero's tooling strategy is to "batteries included" in terms of developer workflow (editor support, builds, docs) but keep the standard library itself minimal. All these tools will be open-source and community-maintained, ensuring Aero projects have a polished development environment from the start.

Governance and Community Model

Aero will be developed as an open-source project with a **transparent, community-driven governance**. The governance model is inspired by successful languages like Rust:

- **Open Governance & RFC Process**: Major language changes or features will follow a Request For Comments (RFC) process. Any contributor can draft an RFC to propose a new feature or change. These are discussed publicly, iterated with feedback, and only merged when there is broad consensus. Rust's governance page aptly notes: *"Each major decision in Rust starts as a Request for Comments… community deliberation is Rust's secret sauce for quality."* [14] Aero will adopt a similar RFC-driven approach to ensure design quality and community buy-in.

- **Leadership Council and Working Teams**: We foresee a multi-tier project structure. A small leadership council (eventually formalized as an Aero Foundation) will steward releases and policies. Beneath it, teams will form around compiler development, libraries, tooling, documentation, and community moderation (each team led by volunteers). This mirrors Rust's teams (compiler, tools, libraries, etc.) [15].
- **Community Involvement**: All development will occur on public Git repositories. Issues, PRs, and discussion forums (like Discourse or Zulip) will be open to everyone. We will adopt a Contributor Covenant Code of Conduct to ensure an inclusive, respectful environment. Regular community meetings (online) and a roadmap roadmap (published) will maintain alignment.
- **License and Foundation**: Aero's source will be under a permissive open-source license (e.g. MIT/Apache), allowing wide adoption. As the project grows, a non-profit Aero Foundation will be formed (similar to the Rust Foundation [16]) to handle trademarks, release management, and funding. Major industry stakeholders can contribute resources via the Foundation.

By building an open, RFC-based governance structure, Aero aims to grow a vibrant community of users and contributors. This inclusive model (as in other successful OSS languages) helps Aero evolve in a transparent way, with real-world feedback driving improvements.

Target Niche and "Killer App"

Aero's niche is modern systems programming where **performance and programmer productivity must go hand-in-hand**. Potential killer domains include:

- **High-Performance Computing & Data Pipelines**: Aero is ideal for data-intensive servers and analytics pipelines requiring both speed and quick iteration. In domains like ML/AI infrastructure or real-time data processing, Aero can replace Python for prototyping (with a static compile mode for performance) or C++ for final implementation, offering "10× improvement" in safety and maintainability while matching raw speed. The research suggests that safe high-performance data analysis could be a killer domain for such languages [17].
- **Game and Graphics Engines**: Game development demands low-level control with high speed. Aero's ergonomics (expressive generics, RAII, easy concurrency) would streamline engine and game logic coding. Aero could become the "Unity script" or engine language of the future, combining C++ speed with a friendly syntax.
- **Embedded and IoT Systems**: In embedded devices, resource constraints make GC impractical. Aero's no-GC model and manual resource control are attractive here. The safety guarantees also help prevent costly device bugs. Aero could target microcontroller development (like a safer alternative to C/C++ on Arduino or RTOS).
- **Safe Systems Software**: Following Rust's lead, Aero could enable safe development of OS kernels, drivers, and networking stacks. Its memory safety without GC is crucial for OS code, and its zero-cost abstractions allow writing low-level code succinctly. A future Aero-based embedded OS or network service stack could showcase its strengths.

A killer application need not be a single program, but rather a domain. For example, a high-performance web framework (like Rails was for Ruby) could spur Aero adoption. Aero might feature a standard async web stack or scientific computing library as an "anchor" project to draw users. In all cases, Aero's *unique selling point* is safe concurrency and memory management without compromising speed [17] [6], which addresses critical needs in these domains.

Conclusion

Aero unites the best of both worlds: the **raw performance** and **fine control** of C++-style systems languages (honoring the zero-overhead principle [1]) with the **clarity**, **modern safety**, and **conciseness** demanded by today's developers (echoing the Zen of Python [2] and Go's simplicity [3]). Through strong static typing, ownership-based memory management, and first-class concurrency, it eliminates entire classes of bugs at compile time [4] [12]. Aero's rich generics and metaprogramming empower developers to write high-level abstractions without performance cost. With careful design, comprehensive tooling (LSP, packages, formatters) and open governance [14] [16], Aero aims to forge a thriving ecosystem. We anticipate Aero becoming the language of choice for next-generation high-performance applications: from real-time data platforms and game engines to safe embedded systems. The combination of *performance* and *ergonomics* offers a compelling 10× improvement in developer productivity and system reliability [17] [6]—the kind of "killer advantage" needed to drive adoption.

Sources: Principles and comparisons are drawn from existing language documentation and research: Python's Zen and design philosophy [2] [9]; C++'s zero-overhead principle [1]; Go's design goals [3] [18]; Rust's safety and concurrency model [4] [12] [5]; and general programming language research [6] [7]. (Diagrams and quotes are cited accordingly.)

---

[1] Zero-overhead principle - cppreference.com
https://en.cppreference.com/w/cpp/language/Zero-overhead_principle.html

[2] PEP 20 – The Zen of Python | peps.python.org
https://peps.python.org/pep-0020/

[3] [18] The Go Programming Language: Simplicity and Power for the Modern Developer - DEV Community
https://dev.to/empiree/the-go-programming-language-simplicity-and-power-for-the-modern-developer-2ng0

[4] [16] Rust: The Modern Programming Language for Safety and Performance | by Make Computer Science Great Again | Medium
https://medium.com/@MakeComputerScienceGreatAgain/rust-the-modern-programming-language-for-safety-and-performance-b003774d7166

[5] Ownership Recap - The Rust Programming Language
https://rust-book.cs.brown.edu/ch04-05-ownership-recap.html

[6] Garbage collection (computer science) - Wikipedia
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[7] Actor model - Wikipedia
https://en.wikipedia.org/wiki/Actor_model

[8] Shared-State Concurrency - The Rust Programming Language
https://doc.rust-lang.org/book/ch16-03-shared-state.html

[9] Python (programming language) - Wikipedia
https://en.wikipedia.org/wiki/Python_(programming_language)

[10] Rust – memory safety without garbage collector | theburningmonk.com
https://theburningmonk.com/2015/05/rust-memory-safety-without-gc/

11 Concurrency Models
https://jenkov.com/tutorials/java-concurrency/concurrency-models.html

12 Rust Common Interview Questions - Part 3 - Tao's Blog
https://www.ubitools.com/en/rust-interview-03/

13 Official page for Language Server Protocol
https://microsoft.github.io/language-server-protocol/

14 15 Governance - Rust Programming Language
https://www.rust-lang.org/governance

17 Novel Programming Language Research Path.txt
file://file-XwZb9uxZxvhfJtvCqMhogM