# Runtime Simplicity Matters as Much as Type Power:

Predictable Execution, Practical Tooling, and the Operational Philosophy of JAPL

JAPL Language Design Group
matthew@yonedaai.com

March 2026

## Abstract

Programming language research has historically privileged type system expressiveness over operational concerns such as compilation speed, deployment simplicity, tooling quality, and runtime predictability. This paper argues that these operational properties are equally important to the long-term success of a language, and that neglecting them produces languages that are *elegant but operationally miserable*—powerful in theory but painful in practice. We present the design philosophy of JAPL, a strict, typed, effect-aware functional language whose seventh core principle is *Runtime Simplicity Matters as Much as Type Power*. We formalize a *type power budget* that constrains type system complexity to features yielding proportionate safety gains, describe a hybrid GC/ownership memory model that provides Erlang-like ease for ordinary values and Rust-like precision for external resources, and detail a compilation model optimized for fast incremental builds and static binary output. We demonstrate how effect types enable aggressive dead code elimination without sacrificing modularity, how built-in tooling (formatter, test runner, profiler, package manager) reduces friction comparable to Go's unified toolchain, and how cross-compilation and single-binary deployment eliminate entire categories of operational failure. A comparative case study of building, testing, and deploying a production service in JAPL, Go, Rust, Haskell, and Erlang quantifies the operational advantages of treating runtime simplicity as a first-class design concern. Our central thesis is that the design space of programming languages admits a Pareto frontier where substantial type safety and operational simplicity coexist—and that reaching this frontier requires deliberate, principled trade-offs rather than maximal expressiveness.

## 1 Introduction

The history of programming language design reveals a persistent tension. On one side stand languages with powerful, expressive type systems—Haskell [Marlow, 2010], Scala [Odersky et al., 2004], Rust [Matsakis & Klock, 2014]—that offer strong compile-time guarantees but impose significant operational costs: slow compilation, complex deployment, steep learning curves, and opaque runtime behavior. On the other side stand languages like Go [Pike, 2012] and Lua [Ierusalimschy et al., 2006] that prioritize simplicity, fast builds, and straightforward deployment, but sacrifice type-level expressiveness that could prevent entire classes of bugs.

This tension is not merely aesthetic. It has direct consequences for software reliability, developer productivity, and organizational adoption. A language with a perfect type system that takes thirty minutes to compile a medium-sized project *will lose to* a language with adequate types that compiles in two seconds—not because practitioners are lazy, but because fast feedback loops are a prerequisite for iterative development, and iterative development is how working software gets built.

### 1.1 The Usability Crisis

We identify a *usability crisis* in modern programming language design, characterized by several failure modes:

1. **The compilation wall.** Rust projects routinely report clean-build times exceeding ten minutes for moderate codebases [Rust Survey, 2023]. Haskell's GHC is notorious for slow Template Haskell compilation and long linking phases. These delays compound: every failed CI run, every developer context switch, every "let me check

if this compiles" represents lost time that accumulates to hours per week per developer.

2. **The deployment labyrinth.** Deploying a Haskell application requires navigating a maze of dynamically linked libraries, Cabal/Stack version conflicts, and platform-specific GHC builds [Snoyman, 2017]. Erlang applications require a BEAM VM installation on every target machine. Even Rust, which produces static binaries, requires careful management of C library dependencies on Linux.

3. **The tooling desert.** Languages with rich type systems often have fragmented tooling: multiple build systems (Haskell's Cabal vs. Stack vs. Nix), inconsistent formatting conventions, third-party test frameworks with incompatible interfaces, and language servers that lag behind compiler features.

4. **The debugging opacity.** Lazy evaluation in Haskell produces stack traces that bear no relation to the source code's control flow [Allwood et al., 2009]. Rust's deeply nested generic types produce error messages measured in kilobytes. Complex type-level programming creates abstractions that debuggers cannot inspect.

## 1.2 Go's Social Lesson

Go's extraordinary adoption—despite a type system that professional language designers consider primitive [Griesemer et al., 2020]—demonstrates that operational excellence creates its own gravitational pull. Go compiles a 100,000-line project in under two seconds. `go build` produces a single static binary. `gofmt` ended all formatting debates. `go test` requires zero configuration. Cross-compilation is a single environment variable.

Pike [Pike, 2012] was explicit about this: Go was designed for the *working programmer* building *production systems*, not for the language researcher exploring the type-theoretic frontier. The result is a language that is mundane to write in and delightful to deploy—and that combination proved to be exactly what a large segment of the industry needed.

## 1.3 The Elegant-but-Miserable Failure Mode

We coin the term *elegant but operationally miserable* (EBOM) for languages that score highly on type-theoretic metrics but poorly on operational ones. The EBOM failure mode is characterized by:

- Conference papers praising the type system while production users struggle with deployment
- A small, devoted community that views operational complaints as skill issues
- A persistent gap between "Hello World" tutorials and production readiness
- Academic adoption without proportionate industrial adoption

Our thesis is that this failure mode is avoidable. The design space of programming languages admits a Pareto frontier where substantial type safety and operational simplicity coexist. Japl is an attempt to reach that frontier through deliberate, principled trade-offs.

## 1.4 Contributions

This paper makes the following contributions:

1. A formal *type power budget* framework (Section 10) that evaluates type system features by their safety-per-complexity ratio, providing principled criteria for feature inclusion/exclusion.

2. A detailed description of Japl's hybrid GC/ownership memory model (Section 5) that achieves GC ease for common cases and ownership precision for resources.

3. An analysis of Japl's compilation model (Section 6) showing how effect types enable optimization opportunities unavailable to languages with unrestricted side effects.

4. A comparative case study (Section 12) quantifying the operational differences between Japl, Go, Rust, Haskell, and Erlang for a representative production workload.

5. A design rationale for treating tooling as a language feature (Section 7), not an afterthought.

## 2 Background and Related Work

### 2.1 Go's Design Philosophy

Go [Pike, 2012, Donovan & Kernighan, 2015] was designed at Google by Rob Pike, Ken Thompson, and Robert Griesemer with an explicit focus on large-scale software engineering. Its design priorities—fast compilation, minimal language surface, built-in concurrency, static binaries, and canonical formatting—were driven by the pain of working with C++ at Google scale.

Pike's influential talk "Simplicity is Complicated" [Pike, 2015] articulated a core insight: simplicity in a programming language is itself a feature that requires significant design effort. Go's designers made controversial choices—no generics (until Go 1.18), no algebraic data types, no pattern matching, no exceptions—because each feature would add complexity to compilation, tooling, and comprehension.

The key lessons from Go's success that inform Japl's design:

1. **Compilation speed is a feature.** Go's compiler was designed from the start to be fast, with dependency analysis that avoids recompiling unchanged packages.
2. **Canonical tooling eliminates social friction.** `gofmt` ended formatting wars; `go test` standardized testing; `go vet` codified best practices.
3. **Single-binary deployment is transformative.** Eliminating shared library dependencies removes an entire failure mode category from production systems.
4. **Cross-compilation should be trivial.** Go's `GOOS` and `GOARCH` environment variables make cross-compilation a non-event.

### 2.2 Rust's Compile Time Debate

Rust [Matsakis & Klock, 2014, Klabnik & Nichols, 2019] provides powerful type-level guarantees—ownership, borrowing, lifetimes, trait-based generics—at the cost of compilation speed. The Rust community has engaged in extensive discussion about compile times, with the `cargo build` experience being a persistent pain point [Rust Survey, 2023].

Several factors contribute to Rust's compilation cost:

- **Monomorphization:** Generic functions are compiled separately for each concrete type, producing better code at the cost of more compilation work.
- **Borrow checking:** The ownership and lifetime analysis is a whole-function analysis that does not parallelize trivially.
- **LLVM backend:** Rust's use of LLVM provides excellent code generation but LLVM itself is not fast.
- **Procedural macros:** Rust's macro system requires compiling and executing Rust code during compilation.

Matsakis and others have investigated incremental compilation [Matsakis, 2016] and parallel codegen as mitigations, but the fundamental trade-off remains: Rust's type system *requires* analysis that is inherently more expensive than Go's simpler type checking.

Japl learns from Rust that ownership and linear types are valuable for resource management, but avoids making ownership the *universal* memory management strategy, thereby reducing the scope of expensive lifetime analysis.

### 2.3 Haskell's Deployment Challenges

Haskell [Marlow, 2010, Jones, 2003] represents the state of the art in type system expressiveness for a general-purpose language, with type classes, higher-kinded types, GADTs, type families, and dependent types (via extensions). However, deploying Haskell applications in production remains difficult:

- **Dynamic linking by default:** GHC-compiled binaries depend on shared libraries that must be present on the target system.
- **Build system fragmentation:** The Haskell ecosystem has cycled through Cabal, Stack, and Nix-based builds without convergence.
- **Runtime unpredictability:** Lazy evaluation can cause space leaks that are difficult to diagnose [Mitchell, 2013].

- **Cross-compilation:** Cross-compiling Haskell is possible but requires significant setup compared to Go or Rust.

Snoyman [Snoyman, 2017] documented these challenges in detail, noting that Haskell's deployment story is the primary barrier to adoption in organizations that have already accepted the language's type system.

## 2.4 OCaml's Pragmatism

OCaml [Leroy, 2014] occupies an interesting middle ground: a type system with algebraic data types, parametric polymorphism, and a powerful module system, combined with a simple compilation model that produces fast native code. OCaml compiles quickly, produces reasonable executables, and has a straightforward deployment model.

However, OCaml's ecosystem has historically suffered from tooling gaps: no canonical formatter (until `ocamlformat`), fragmented build systems (ocamlbuild, jbuilder/dune), and a package ecosystem smaller than Haskell's or Rust's. Recent efforts—opam 2.0, dune, the OCaml Language Server Protocol implementation—have improved the situation significantly.

## 2.5 Gleam's Simplicity-First Approach

Gleam [Gleam, 2023] is a typed language that targets the BEAM virtual machine, bringing algebraic data types and exhaustive pattern matching to the Erlang ecosystem. Gleam's design philosophy explicitly prioritizes simplicity: no macros, no operator overloading, no implicit behavior, a single canonical formatter.

Gleam demonstrates that meaningful type safety is achievable without the full weight of Haskell's type system. Its limitation—targeting only the BEAM—means it inherits BEAM's deployment model (requiring the Erlang runtime).

## 2.6 Zig's Comptime

Zig [Kelley, 2020] introduces `comptime`, a mechanism for compile-time code execution that replaces both generics and macros with a single, unified concept. Zig's approach is notable for its transparency: the same language is used at compile time and run time, making the compilation model easier to understand and debug.

Zig also demonstrates that a language can achieve cross-compilation excellence by designing for it from the start, using a self-hosted compiler that bundles its own C library implementations.

## 2.7 Erlang's Runtime Philosophy

Erlang [Armstrong, 2003, 2007] and its runtime (the BEAM [Happi, 2023]) provide the gold standard for runtime observability:

- Hot code loading: deploying new code without stopping the system
- Process inspection: examining any process's state, mailbox, and call stack at runtime
- Tracing: enabling per-function tracing without recompilation
- Distribution: transparent cross-node communication

The BEAM's reduction-based scheduler provides fair, preemptive scheduling of lightweight processes, with each process independently garbage-collected. This model—per-process heaps, no shared mutable state, preemptive scheduling—directly inspires Japl's runtime design.

# 3 Formal Framework

We formalize the interaction between type system complexity and operational quality, providing a principled basis for the design trade-offs in Japl.

## 3.1 Compiler Architecture for Fast Compilation

Let $P$ denote a program consisting of modules $M_1, \ldots, M_n$ with dependency graph $G = (V, E)$ where $V = \{M_i\}$ and $(M_i, M_j) \in E$ iff $M_i$ depends on $M_j$.

**Definition 3.1** (Compilation Cost Model)**.** *The total compilation time $T(P)$ for a program $P$ with parallelism $k$ is:*

$$T(P) = T_{parse} + T_{type} + T_{opt} + T_{codegen} + T_{link}$$

*where each phase has cost determined by the critical path through $G$ given $k$ workers:*

$$T_{phase}(P, k) = \frac{W_{phase}(P)}{k} + D_{phase}(G)$$

*Here $W_{phase}$ is the total work and $D_{phase}$ is the span (longest dependent chain) for that phase.*

**Proposition 3.2** (Type Checking Dominates). *For languages with* unrestricted type-level features *(HKTs, GADTs, type families, dependent types), type checking dominates compilation:*

$$\lim_{|P|\to\infty} \frac{T_{type}(P)}{T(P)} \to 1$$

*This limit applies specifically to languages that permit open-ended type-level computation; languages like* Japl *that bound their type systems to decidable fragments avoid this asymptotic behavior. In particular, Hindley-Milner inference is $\mathcal{O}(n)$ for practical inputs [McAdam, 1998] but extensions like type families introduce decision procedures that are NP-complete or undecidable in the general case [Sulzmann et al., 2007].*

### 3.2 Incremental Compilation

**Definition 3.3** (Incremental Compilation Efficiency). *Given a change $\delta$ to module $M_i$, the incremental recompilation set is:*

$$Recomp(\delta) = \{M_i\} \cup \{M_j : M_j \text{ transitively depends on } M_i \text{ and } iface(M_i) \text{ changed}\}$$

*The incremental efficiency ratio is:*

$$\eta(\delta) = 1 - \frac{|Recomp(\delta)|}{|V|}$$

A language design that stabilizes module interfaces—for example, by separating interface declarations from implementations, as pioneered by Modula-2 [Wirth, 1982] and Ada [Ichbiah et al., 1979]—maximizes $\eta$ because most changes are implementation-only and do not propagate.

Japl's module system requires explicit signatures at module boundaries. When a module's implementation changes but its signature does not, downstream modules need not be recompiled. Effect annotations in signatures provide additional stability: if a function's effects do not change, downstream code that depends on those effects remains valid.

### 3.3 Decidability vs. Expressiveness

**Theorem 3.4** (Expressiveness–Decidability Trade-off). *Let $\mathcal{L}$ be a type system with decision procedure $D_{\mathcal{L}}$ for type checking. The following hierarchy of type system features has increasing worst-case complexity for $D_{\mathcal{L}}$:*

1. *Simple types (Hindley-Milner): decidable, $\mathcal{O}(n)$ practical*
2. *Bounded quantification (System $F_{<:}$): decidable, potentially exponential [Pierce, 2002]*
3. *Type classes with functional dependencies: decidable with restrictions, undecidable in general [Sulzmann et al., 2007]*
4. *GADTs: type checking decidable, type inference undecidable [Jones et al., 2006]*
5. *Dependent types: type checking is undecidable in general (reduces to the halting problem)*

Japl draws the line between items 2 and 3: it includes algebraic data types, parametric polymorphism, traits (type classes without functional dependencies), row polymorphism, and effect types, while excluding GADTs, type families, and dependent types. This boundary ensures that type checking remains decidable and practically fast.

### 3.4 Effect Types and Optimization

Effect annotations create optimization opportunities unavailable to languages with unrestricted side effects.

**Proposition 3.5** (Effect-Based Dead Code Elimination). *Let $f$ be a function with effect signature $\epsilon_f$ and let $e$ be an expression that calls $f$. If $\epsilon_f = Pure$ and the result of $e$ is unused, then $e$ can be eliminated. More generally, if $\epsilon_f$ does not include Io or Process, and the result is unused, elimination is safe if the remaining effects (State, Fail) are handled locally.*

This is strictly stronger than the optimizations available to Go or OCaml, where any function call might perform I/O, and the compiler must conservatively retain all calls.

## 4 Japl's Runtime Philosophy

Japl's seventh principle—*Runtime Simplicity Matters as Much as Type Power*—is not merely a preference but a design constraint that shapes every aspect of the language and its implementation. We enumerate the concrete commitments this principle entails.

### 4.1 Single Fast Compiler

Japl ships a single compiler, `japlc`, that is the canonical and only implementation. There is no alternative compiler, no compiler plugin ecosystem that fragments the language, and no compiler flags that change the semantics of valid programs.

The compiler targets two backends:

- **Cranelift** [Bytecode Alliance, 2020]: For fast debug builds. Cranelift is designed for fast code generation with reasonable code quality, making it ideal for the edit-compile-test loop.

- **LLVM** [Lattner & Adve, 2004]: For optimized release builds. LLVM provides state-of-the-art code generation at the cost of slower compilation.

The default (`japl build`) uses Cranelift; `japl build -release` uses LLVM. This dual-backend strategy provides Go-like compilation speed for development and Rust-like code quality for production.

### 4.2 Single Standard Formatter

```
$ japl fmt                # format current
   project
$ japl fmt --check        # check without
   modifying
$ japl fmt src/main.japl # format single file
```

`japl fmt` is an opinionated formatter with no configuration options, following Go's `gofmt` philosophy. There is one canonical style. Formatting debates are impossible because the tool's output is the definition of correct formatting.

This decision is informed by Go's experience: `gofmt` is widely regarded as one of Go's most impactful features, not because of any particular formatting choice, but because it eliminated an entire category of unproductive discussion [Pike, 2012].

### 4.3 Dead-Simple Build Tool

Japl's build tool is integrated into the compiler. There is no separate build system, no Makefile, no configuration file for basic projects:

```
$ japl build             # compile the project
$ japl build --release   # optimized build via
   LLVM
$ japl run               # compile and execute
$ japl run --profile     # compile and execute
   with profiling
```

Project configuration, when needed, lives in a single `japl.toml` file:

```
-- japl.toml
name = "my-service"
version = "1.2.0"

[deps]
http = "0.5"
json = "1.0"
postgres = "0.3"
```

### 4.4 Small, High-Quality Standard Library

Japl's standard library follows Go's approach of including enough to be productive without external dependencies for common tasks:

- `Std.Http` – HTTP client and server
- `Std.Json` – JSON encoding/decoding
- `Std.Crypto` – Cryptographic primitives
- `Std.Fs` – File system operations
- `Std.Net` – TCP/UDP networking
- `Std.Test` – Testing framework
- `Std.Time` – Time and duration
- `Std.Log` – Structured logging
- `Std.Trace` – Distributed tracing

The standard library is intentionally *small*: it includes only functionality that most server-side applications need, is difficult to get right independently, or benefits from tight integration with the runtime (e.g., process-aware logging).

### 4.5 Built-in Test Runner

Tests are a language construct, not a library convention:

```
test "user creation validates email" {
  let result = create_user("bad-email")
  assert result == Err(InvalidEmail("bad-
      email"))
}

test "order total includes tax" {
  let order = make_order([item(10.0), item
      (20.0)])
  let total = calculate_total(order, tax_rate
       = 0.1)
  assert total == 33.0
}

property "reversing twice is identity" {
  forall (xs: List[Int]) →
    List.reverse(List.reverse(xs)) == xs
}
```

```
$ japl test                # run all tests
$ japl test --filter user  # run matching
    tests
$ japl test --parallel 8   # parallel test
    execution
$ japl test --coverage     # with coverage
    report
```

Tests are discovered by the compiler, not by file naming conventions or annotation processing. The compiler knows about tests, which enables test-aware optimizations (e.g., skipping code generation for non-test code when running tests).

### 4.6 Cross-Compilation

```
$ japl build --target linux-amd64
$ japl build --target linux-arm64
$ japl build --target darwin-arm64
$ japl build --target darwin-amd64
$ japl build --target windows-amd64
$ japl build --target wasm32
```

Cross-compilation works out of the box because the compiler bundles everything needed to generate code for all supported targets. There are no external dependencies, no target-specific SDK installations, and no platform-specific configuration.

### 4.7 Static Binaries

```
$ japl build --static myapp.japl
$ file ./myapp
./myapp: ELF 64-bit LSB executable,
    statically linked
$ ldd ./myapp
  not a dynamic executable
```

Every JAPL build produces a single static binary with no external dependencies. The binary includes the JAPL runtime (scheduler, GC, network stack) and all application code. This is the *only* deployment artifact.

### 4.8 Readable Stack Traces

When a JAPL process crashes, the stack trace maps directly to source code:

```
Process <0.47.0> crashed: DivisionByZero
  at Math.divide        (src/math.japl:42:15)
  at Order.calculate    (src/order.japl:87:10)
  at Handler.process    (src/handler.japl:23:5)
  at Server.handle_req  (src/server.japl:156:3)

Process mailbox (3 messages pending):
  ProcessOrder({id: "ord-123", ...})
  ProcessOrder({id: "ord-456", ...})
  Shutdown

Supervisor <0.12.0> restarting child "order-handler"
  (restart count: 1/5, strategy: OneForOne)
```

Strict evaluation ensures that stack traces reflect the actual execution order. There are no thunks, no lazy chains, and no deferred evaluation that would make the stack trace misleading.

### 4.9 Built-in Profiling and Tracing

```
$ japl run --profile myapp.japl
# Produces: myapp.prof (native profiling data
    )

$ japl trace --format chrome myapp.japl
# Produces: myapp.trace (Chrome trace format)

$ japl run --gc-stats myapp.japl
# Prints per-process GC statistics
```

Profiling and tracing are built into the runtime, not bolted on via sampling profilers or LD_PRELOAD tricks. The runtime knows about processes, mailboxes, supervision trees, and effect handlers, and can provide domain-specific profiling information that generic tools cannot.

## 5 The GC + Ownership Hybrid Memory Model

JAPL's memory model is the key architectural decision that enables both runtime simplicity and resource safety. Rather than choosing between garbage collection (Go, Haskell, Erlang) and ownership (Rust), JAPL uses both, each in its natural domain.

## 5.1 Design Rationale

The fundamental observation is that most values in a functional program are *immutable data*—algebraic data types, records, strings, closures, collections—while a smaller but critical set of values are *external resources*—file handles, network sockets, database connections, GPU buffers.

These two categories have radically different management requirements:

| Property | Immutable Data | Resources |
|---|---|---|
| Mutability | Never | Often |
| Sharing | Free (no races) | Dangerous |
| Lifetime | Until unreachable | Until explicitly released |
| Cleanup | Whenever (GC) | Immediately (deterministic) |
| Copying | Cheap (structural sharing) | Often impossible |

Table 1: Distinct management requirements for data vs. resources.

Applying GC to resources is wrong: GC provides no deterministic cleanup guarantees, so file handles may remain open and connections may leak. Applying ownership to immutable data is wasteful: lifetime annotations add cognitive overhead for values that can be safely shared without restriction.

## 5.2 The Immutable Heap

The immutable heap is managed by a generational, per-process garbage collector inspired by the BEAM [Happi, 2023]:

1. **Per-process heaps.** Each JAPL process has its own heap partition. GC in one process does not pause other processes. This is critical for latency-sensitive applications: a process handling an HTTP request is not affected by GC in a process doing batch processing.

2. **No write barriers.** Because all heap data is immutable, there are no pointer updates to track. This eliminates write barriers, simplifying the GC implementation and improving mutator throughput.

3. **Generational collection.** Most functional programs exhibit generational behavior: most values are short-lived (intermediate results in a pipeline). A nursery/old-generation split captures this efficiently.

4. **Instant reclamation on process death.** When a process exits, its entire heap is freed in $\mathcal{O}(1)$ time, without tracing. This is especially valuable for request-handling processes with short lifetimes.

```
-- All these values live on the immutable
    heap
-- and are managed by GC. No annotations
    needed.
let config = { host = "localhost", port =
    8080 }
let items = [1, 2, 3, 4, 5]
let doubled = List.map(items, fn x → x * 2)
let name = "JAPL"

-- Values can be freely shared across
    processes
-- because they cannot be mutated
Process.send(worker, ProcessData(config,
    items))
```

## 5.3 The Resource Arena

External resources live in a separate arena managed by linear types and ownership tracking:

```
-- Resource layer: ownership-tracked, must be
     consumed
fn process_file(path: String)
    → Result[String, IoError] with Io =
  use file = File.open(path, Read)?
  let contents = File.read_all(file)?
  File.close(file)  -- consumed: forgetting =
        error
  Ok(contents)

-- Transfer ownership between processes
fn send_to_worker(buf: own Buffer,
                  pid: Pid[WorkerMsg]) →
                      Unit =
  Process.send(pid, ProcessBuffer(buf))
  -- buf is moved; using it here is a compile
        error
```

The ownership rules for the resource arena are:

1. Every resource has exactly one owner at any time.

2. Ownership can be transferred (`own` qualifier) but not duplicated.

3. Read-only borrows (`ref` qualifier) allow temporary shared access.

4. When the owner goes out of scope without consuming the resource, the compiler reports an error.

5. The compiler rejects programs that use a resource after transfer.

## 5.4 Interface Between the GC Heap and the Resource Arena

A critical design question is how the two memory regions interact. The rule is strict: *immutable values on the GC heap cannot directly contain owned resources.* Because the GC heap provides no deterministic finalization guarantee, allowing a GC-managed record to hold an `own File` would make resource cleanup non-deterministic—precisely the failure mode the ownership layer is designed to prevent.

Instead, resources that must be referenced from immutable data structures are wrapped in *process-local handles*: opaque, copyable identifiers that index into the process's resource arena. A handle has type `Handle[R]` for resource type `R`, and is an ordinary immutable value that can be freely placed in records, lists, or any GC-managed collection. The actual resource remains in the arena, subject to ownership tracking.

```
-- A Handle[File] is a copyable, GC-managed
    token.
-- The File itself stays in the resource
    arena.
let h: Handle[File] = Resource.register(file)
let config = { name = "log", target = h }

-- To use the resource, borrow it via the
    handle:
Resource.with(config.target, fn (f: ref File)
    →
  File.write(f, "entry"))

-- Explicit release through the handle:
Resource.release(config.target)
-- Further borrows via this handle are
    compile errors.
```

The compiler enforces this separation through a *kind distinction*: types of kind `Resource` (those carrying ownership obligations) are excluded from appearing as fields of types of kind `Data` (the default kind for GC-managed values). Any attempt to embed an `own File` directly inside a record or list is a kind error. When a programmer needs a collection of resources, JAPL provides *linear collections* (e.g., `LinearList[own File]`) that themselves carry ownership obligations and must be consumed explicitly. These linear collections live in the resource arena, not the GC heap, preserving deterministic cleanup.

This design means that resources are *second-class* with respect to standard collections: an `own File` cannot be placed in a regular `List` or `Map`. Linear collections exist for the cases where resource aggregation is needed, but the common case—packaging immutable data into standard collections—remains free of ownership annotations.

## 5.5 Why This Is the Sweet Spot

**Theorem 5.1** (Hybrid Memory Model Correctness)**.** *The JAPL hybrid memory model satisfies the following properties:*

1. ***Data race freedom:*** *No data races are possible on immutable heap data (trivially, since data is immutable) or resource arena data (since resources have a single owner).*
2. ***Resource safety:*** *Every allocated resource is eventually freed, and no resource is used after freeing.*
3. ***No annotation overhead for common case:*** *Immutable data (the vast majority of values in a functional program) requires no ownership annotations.*

*Proof sketch.* (1) follows from immutability of heap data and single-ownership of arena data. (2) follows from the linearity constraint: the compiler's type system ensures that every resource is consumed exactly once, either by explicit deallocation or by transfer to another owner. (3) follows from the fact that GC-managed data requires no programmer-visible lifetime management.

We strengthen claim (2) with a small-step operational semantics sketch. Define a program state $\sigma = (H, A, e)$ where $H$ is the immutable GC heap, $A$ is the resource arena (a finite map from resource identifiers to resource states $\{\textsc{Live}, \textsc{Freed}\}$), and $e$ is the expression under evaluation. The key reduction rules for resources are:

$$\begin{aligned}
\textsc{Alloc}: &\quad (H, A, \texttt{open}(v)) \longrightarrow (H, A[r \mapsto \textsc{Live}], r) \quad r \text{ fresh} \\
\textsc{Use}: &\quad (H, A, \texttt{use}(r)) \longrightarrow (H, A, v) \quad \text{if } A(r) = \textsc{Live} \\
\textsc{Free}: &\quad (H, A, \texttt{close}(r)) \longrightarrow (H, A[r \mapsto \textsc{Freed}], ()) \\
\textsc{Move}: &\quad (H, A, \texttt{send}(\mathit{pid}, r)) \longrightarrow (H, A, ()) \quad \text{ownership transfe}
\end{aligned}$$

A state is *invalid* if any reduction attempts USE or FREE on $r$ where $A(r) = \textsc{Freed}$, or if $r$ has been moved and is subsequently referenced in the source context. The type system's linearity discipline guarantees that for any well-typed program $e$, the reduction sequence from $(H_0, A_0, e)$ never reaches an invalid state: every resource identifier $r$ is used at most once after allocation, consumed exactly once by FREE or MOVE, and never accessed after consumption. A full mechanized proof would proceed by induction on the typing derivation and case analysis on the reduction rules, following the structure of linear type soundness proofs in the literature [Pierce, 2002]. □

The practical implication is that JAPL programmers write most code in a style identical to Erlang or Haskell—pure functions transforming immutable values—but gain Rust-level resource safety for the 10–15% of code that interacts with external resources.

### 5.6 Comparison with Alternative Approaches

| | **Data** | **Resources** | **Burden** |
|---|---|---|---|
| Go | GC | GC + finalizers | Low, but leaks |
| Rust | Ownership | Ownership | High everywhere |
| Haskell | GC | GC + brackets | Low, but leaks |
| Erlang | GC | Ports/NIFs | Low, but unsafe |
| **JAPL** | **GC** | **Ownership** | **Low + safe** |

Table 2: Memory management strategies compared.

## 6 Compilation Model

JAPL's compilation model is designed for the edit-compile-test loop, where the latency between saving a file and seeing test results is the critical metric.

### 6.1 Architecture Overview

The compilation pipeline proceeds through the following phases:

1. **Parsing** ($\mathcal{O}(n)$): Source text to AST. Each file is parsed independently and in parallel.

2. **Name resolution** ($\mathcal{O}(n \log n)$): Resolving imports, building the module dependency graph.

3. **Type checking and inference** ($\mathcal{O}(n)$ practical): Bidirectional type checking with local inference. Effect inference within function bodies; effect checking at module boundaries.

4. **Effect-aware optimization**: Dead code elimination, function inlining, and constant folding, guided by effect annotations.

5. **Code generation**: Cranelift (debug) or LLVM (release).

6. **Linking**: Static linking of all modules and the runtime into a single binary.

### 6.2 Whole-Program vs. Separate Compilation

JAPL uses a hybrid approach:

- **Separate compilation for type checking:** Each module is type-checked against the *signatures* of its dependencies, not their implementations. This enables parallel type checking and incremental recompilation.

- **Whole-program optimization for code generation:** In release mode, the optimizer sees the entire program, enabling cross-module inlining, specialization, and dead code elimination.

This hybrid approach gets the best of both worlds: fast incremental builds during development (separate compilation) and aggressive optimization for production (whole-program).

### 6.3 Incremental Strategies

JAPL's incremental compilation uses a fine-grained dependency tracking system:

**Definition 6.1** (Interface Stability)**.** *A module interface $I(M)$ consists of:*

- *Type definitions exported by $M$*
- *Function signatures (including effect annotations) exported by $M$*
- *Trait implementations provided by $M$*

*A change $\delta$ to module $M$ is* interface-stable *if $I(M)$ is unchanged after applying $\delta$.*

**Proposition 6.2** (Incremental Recompilation Bound). *If a change $\delta$ to module $M$ is interface-stable, only $M$ itself needs recompilation. The downstream modules that depend on $M$ need not be recompiled or even re-checked.*

Effect annotations contribute to interface stability in an important way: if a function's implementation changes but its effect signature does not, downstream modules remain valid. This is strictly better than languages without effect tracking, where any implementation change might introduce new side effects that callers should know about.

## 6.4 How Effect Types Enable Optimization

**Proposition 6.3** (Pure Function Optimization). *Functions annotated as Pure (the default in JAPL) admit the following optimizations that are unsound for effectful functions:*

1. ***Common subexpression elimination:*** *If $f$ is pure and $f(x)$ appears twice, the second occurrence can be replaced with the result of the first.*
2. ***Dead code elimination:*** *If the result of $f(x)$ is unused, the call can be eliminated.*
3. ***Reordering:*** *Calls to pure functions can be reordered freely.*
4. ***Memoization:*** *Results of pure functions can be cached.*
5. ***Parallelization:*** *Independent calls to pure functions can be evaluated in parallel.*

For effectful functions, the effect annotation provides partial optimization information:

- State[$s$] functions can be reordered with respect to Io functions (state is local).
- Fail[$e$] functions can be eliminated if wrapped in a handler that discards the result.
- Net functions cannot be reordered or eliminated (observable side effects).

## 6.5 AOT vs. JIT Trade-offs

JAPL uses ahead-of-time (AOT) compilation exclusively, forgoing JIT compilation. This decision is motivated by several factors:

1. **Predictable performance.** AOT-compiled code has consistent performance characteristics from the first invocation. There is no warm-up period, no sudden deoptimizations, and no performance cliffs.
2. **Simpler deployment.** A statically compiled binary requires no JIT compiler, no runtime code cache management, and no W^X memory permission considerations (relevant for security-hardened environments).
3. **Cross-compilation compatibility.** AOT compilation is inherently cross-compilation-friendly: the compiler generates code for the target architecture, period. JIT compilers typically require target-architecture JIT implementations.
4. **Container friendliness.** Static binaries produce minimal container images (potentially `FROM scratch` Docker images), reducing attack surface and image pull times.

The trade-off is that JAPL cannot perform profile-guided optimizations at runtime or specialize generic code based on observed type distributions. We argue that this trade-off is acceptable because: (a) AOT link-time optimization with profile-guided optimization (PGO) data from previous runs provides similar benefits; (b) effect-type-guided optimization recovers many opportunities that JIT systems use runtime profiling to discover; and (c) predictable performance is more valuable than peak performance for the server-side workloads JAPL targets.

# 7 Tooling as Language Feature

JAPL treats tooling not as an ecosystem concern but as a language design concern. Every tool ships with the compiler and is maintained by the core team.

## 7.1 The Case for Integrated Tooling

The fragmentation of tooling in languages like Haskell, Scala, and even Rust (where `rustfmt` was not part of the initial release) creates several problems:

1. **Configuration proliferation.** When multiple tools exist, each has its own configuration format, and projects accumulate dotfiles: `.prettierrc`, `.eslintrc`, `.editorconfig`, `rustfmt.toml`, `clippy.toml`.
2. **Version skew.** Third-party tools may not support the latest language features, creating windows where newly added syntax cannot be formatted or linted.
3. **Onboarding friction.** New contributors must install and configure multiple tools beyond the compiler itself.
4. **CI complexity.** Each tool adds a CI step, potentially with its own failure modes and update cadence.

## 7.2 Built-in Formatter

`japl fmt` is a deterministic formatter: given the same AST, it always produces the same output. It has zero configuration options. Every JAPL project in the world looks the same.

The formatter is implemented as a library that the compiler also uses for error message formatting, ensuring consistency between the code the programmer writes and the code the compiler displays in diagnostics.

## 7.3 Built-in Test Runner

Tests are first-class syntactic constructs:

```
module UserTest

import User.{create_user, validate_email}

test "valid email passes validation" {
  assert validate_email("alice@example.com")
    == Ok("alice@example.com")
}

test "missing @ is rejected" {
  assert validate_email("bad-email")
    == Err(InvalidEmail("bad-email"))
}

property "email roundtrip" {
  forall (local: AlphaString, domain:
      DomainString) ->
    let email = local ++ "@" ++ domain
    validate_email(email) == Ok(email)
}
```

Because tests are known to the compiler, it can:

- Skip codegen for non-test code in test-only builds
- Provide test-specific error messages that reference the assertion expression
- Track test coverage at the expression level, not the line level
- Run property-based tests with integrated shrinking

## 7.4 Built-in Package Manager

`japl deps` manages dependencies through a centralized registry with reproducible builds:

```
$ japl deps add http 0.5
$ japl deps update
$ japl deps audit     # security audit
$ japl deps tree      # dependency tree
```

The lockfile format is deterministic and human-readable. Dependency resolution uses a SAT solver to find compatible version sets, with clear error messages when resolution fails.

## 7.5 Language Server Protocol

JAPL ships a Language Server Protocol (LSP) implementation that provides:

- Hover information with types, effects, and documentation
- Go-to-definition (including into standard library and dependencies)
- Find-all-references
- Rename refactoring
- Inline diagnostics with fix suggestions
- Effect signature display for any expression

The LSP server reuses the compiler's type-checking infrastructure, ensuring that IDE feedback is always consistent with compilation results.

### 7.6 REPL

```
$ japl repl
japl> let xs = [1, 2, 3, 4, 5]
xs : List[Int] = [1, 2, 3, 4, 5]

japl> List.map(xs, fn x → x * x)
[1, 4, 9, 16, 25] : List[Int]

japl> :type List.fold
List.fold : fn(List[a], b, fn(b, a) → b) →
    b

japl> :effects File.read_to_string
File.read_to_string : Io, Fail[IoError]
```

The REPL supports incremental compilation: each expression is compiled and executed in the context of previous definitions, with the same type checking and effect tracking as regular code.

### 7.7 Built-in Profiler

```
$ japl run --profile myapp.japl
# After execution:
Top functions by time:
  1. Json.parse          312ms  (42.1%)
  2. Http.handle_req     198ms  (26.7%)
  3. Db.query            156ms  (21.0%)
  4. List.map             42ms   (5.7%)

Per-process GC statistics:
  Process <0.12.0>: 3 minor GCs, 0 major
  Process <0.47.0>: 12 minor GCs, 1 major
  Process <0.48.0>: 8 minor GCs, 0 major

Message passing:
  Total messages: 14,235
  Avg queue depth: 2.3
  Max queue depth: 47 (Process <0.47.0>)
```

Because the profiler is integrated with the runtime, it can provide process-aware, effect-aware profiling data that external profilers cannot: per-process allocation rates, message throughput between process pairs, supervision tree overhead, and effect handler costs.

## 8 Deployment

Deployment is where operational simplicity pays its largest dividends. A language can have the most beautiful type system in the world, but if deploying it requires a Ph.D. in systems administration, it will not be adopted for production use.

### 8.1 Static Binaries

Every Japl build produces a single statically linked binary. The binary includes:

- The application code (compiled to native machine code)
- The Japl runtime (process scheduler, GC, network event loop)
- The standard library (only functions actually used, thanks to dead code elimination)
- A minimal C runtime (for system calls)

No shared libraries, no virtual machine, no interpreter, no runtime downloads.

### 8.2 Cross-Compilation Matrix

Japl supports cross-compilation for all major targets from any development platform:

| Target | From Linux | From macOS | From Windows |
|---|---|---|---|
| linux-amd64 | ✓ | ✓ | ✓ |
| linux-arm64 | ✓ | ✓ | ✓ |
| darwin-amd64 | ✓ | ✓ | ✓ |
| darwin-arm64 | ✓ | ✓ | ✓ |
| windows-amd64 | ✓ | ✓ | ✓ |
| wasm32 | ✓ | ✓ | ✓ |

Table 3: Cross-compilation support matrix.

### 8.3 Container-Friendly Builds

Japl's static binaries enable minimal container images:

```
# Multi-stage build
FROM japl:latest AS builder
COPY . /app
RUN japl build --release --static /app/main.japl

# Final image: just the binary
FROM scratch
COPY --from=builder /app/main /main
ENTRYPOINT ["/main"]
```

The resulting image contains only the application binary—no OS, no package manager, no shell. This produces images in the 5–20 MB range (depending on application size), compared to 100+ MB for typical Go images, 200+ MB for Rust images with a Debian base, and 500+ MB for Haskell images.

### 8.4 Minimal Runtime Dependencies

Japl binaries have zero runtime dependencies beyond the operating system kernel. Specifically, they do not require:

- A C standard library (system calls are made directly)
- DNS resolver libraries (a pure-Japl DNS client is included in the runtime)
- TLS libraries (a pure-Japl TLS implementation is included)
- Any `.so`/`.dylib`/`.dll` files

This eliminates the "works on my machine" class of deployment failures that arise from shared library version mismatches.

### 8.5 Deployment Comparison

| | **Japl** | **Go** | **Rust** | **Haskell** | **Erlang** |
|---|---|---|---|---|---|
| Artifact | Binary | Binary | Binary | Binary | Release |
| Deps | None | libc | libc* | Many | BEAM |
| Image | 5–20MB | 10–30MB | 5–30MB | 200+MB | 100+MB |
| Cross | Trivial | Trivial | Moderate | Hard | N/A |

Table 4: Deployment characteristics. (*Rust can statically link musl libc.)

## 9 Comparison with Existing Languages

We now provide a detailed comparison of Japl against five languages that each represent a different point in the design space.

### 9.1 Go: Gold Standard for Tooling, Weak Types

Go [Pike, 2012, Donovan & Kernighan, 2015] is the language Japl most admires operationally and most disagrees with type-theoretically.

**What Go gets right:**

- Compilation speed (seconds for large projects)
- Single static binary deployment
- Canonical formatter (`gofmt`)
- Unified toolchain (`go build/test/vet/doc`)
- Cross-compilation via `GOOS`/`GOARCH`
- Excellent standard library for networking

**What Go gives up:**

- No algebraic data types (sum types). Error handling is `if err != nil` repeated *ad nauseam.*
- No exhaustive pattern matching. Missing cases are silent bugs.
- No generics until Go 1.18, and the resulting generics are limited.
- No effect tracking. Any function might perform I/O.
- Shared mutable memory with goroutines creates data race possibilities.
- No supervision trees. Goroutine failures are unobserved by default.

Japl aims to match Go's operational excellence while providing the type safety that Go lacks. The key insight is that ADTs, traits, effect types, and pattern matching do not inherently require slow compilation—their type-checking algorithms are well within the decidable, practically-fast region of the complexity hierarchy.

### 9.2 Rust: Powerful but Slow Compilation

Rust [Matsakis & Klock, 2014, Klabnik & Nichols, 2019] provides the strongest compile-time guarantees of any mainstream language, at significant cost.

**What Rust gets right:**

- Ownership and borrowing eliminate memory safety bugs
- Zero-cost abstractions
- Excellent pattern matching and enums
- Trait-based generics
- Growing ecosystem (crates.io)

**What Rust gives up:**

- Compilation speed: 10+ minutes for medium projects is common
- Cognitive overhead: lifetime annotations pervade the codebase
- No lightweight processes (async/await is complex)
- No supervision or fault tolerance primitives
- No effect tracking (beyond `unsafe`)
- Steep learning curve, particularly for the borrow checker

JAPL borrows Rust's ownership model for resources but applies it only where needed (external resources), keeping the common case (pure functional code) free of lifetime annotations.

## 9.3 Haskell: Powerful Types, Deployment Nightmare

Haskell [Marlow, 2010, Jones, 2003] is the purest realization of the "types first" philosophy, and its deployment story illustrates the cost of neglecting operational concerns.

**What Haskell gets right:**

- The most expressive type system in mainstream use
- Purity enforced by the type system
- Algebraic data types and pattern matching
- Type classes and higher-kinded types
- Lazy evaluation enables elegant abstractions

**What Haskell gives up:**

- Compilation speed: GHC is slow, especially with extensions
- Deployment: dynamic linking, platform-specific builds, large executables
- Space leaks from lazy evaluation [Mitchell, 2013]
- Unpredictable stack traces
- Multiple build systems (Cabal, Stack, Nix)
- IO monad creates a "monad transformer stack" complexity cliff

JAPL's effect system achieves Haskell-like purity tracking without monadic syntax overhead, and strict evaluation eliminates the space leak problem entirely.

## 9.4 Erlang: Great Runtime, Weak Tooling

Erlang [Armstrong, 2003, 2007] provides the runtime model that JAPL most closely follows, while addressing its shortcomings.

**What Erlang gets right:**

- Lightweight processes (millions per node)
- Supervision trees and fault tolerance
- Hot code loading
- Runtime observability (`:observer`, tracing)
- Distribution built in
- Per-process GC

**What Erlang gives up:**

- Dynamic typing: runtime type errors in production
- No algebraic data types or exhaustive pattern matching
- No resource safety (no ownership model)
- Requires BEAM VM installation on target systems
- Limited tooling (no canonical formatter until recently)
- Unusual syntax discourages adoption

JAPL combines Erlang's runtime model with static typing, static binaries, and modern tooling.

## 9.5 OCaml: Good Balance, Ecosystem Gaps

OCaml [Leroy, 2014] is perhaps the closest existing language to JAPL's design philosophy, but with significant differences.

**What OCaml gets right:**

- Fast compilation

- Algebraic data types and pattern matching
- Powerful module system (functors)
- Good native code generation
- Hindley-Milner type inference

**What OCaml gives up:**

- No lightweight processes (until OCaml 5.0 with effects)
- No supervision or distribution
- No effect tracking (unrestricted mutation)
- Smaller ecosystem than Go, Rust, or Haskell
- Historically fragmented tooling (improved with dune and opam)
- No cross-compilation story comparable to Go

### 9.6 Summary

## 10 The Type Power Budget

Not all type system features are created equal. Some provide enormous safety benefits at low complexity cost; others provide marginal benefits at high cost. We formalize this observation as a *type power budget*: a framework for evaluating whether a type system feature's safety contribution justifies its complexity.

### 10.1 Formalization

**Definition 10.1** (Type Feature)**.** *A type feature $\phi$ is characterized by a tuple $(S_\phi, C_\phi, I_\phi)$ where:*

- $S_\phi \in [0,1]$ *is the* safety contribution*: the fraction of a representative bug taxonomy that $\phi$ prevents.*
- $C_\phi \in [0,1]$ *is the* complexity cost*: a normalized measure of the cognitive overhead, compilation cost, and tooling difficulty that $\phi$ introduces.*
- $I_\phi \subseteq \Phi$ *is the* interaction set*: the set of other features whose complexity is affected by $\phi$'s presence.*

**Definition 10.2** (Safety-per-Complexity Ratio)**.** *The safety-per-complexity ratio of a feature $\phi$ in the context of a feature set $F$ is:*

$$\rho(\phi, F) = \frac{S_\phi}{C_\phi + \sum_{\psi \in I_\phi \cap F} \Delta C_{\phi,\psi}}$$

*where $\Delta C_{\phi,\psi}$ is the additional complexity from the interaction between $\phi$ and $\psi$.*

**Definition 10.3** (Type Power Budget)**.** *A type power budget $B$ is a threshold on the minimum acceptable ratio:*

$$F^* = \{\phi \in \Phi : \rho(\phi, F^*) \geq B\}$$

*The budget $B$ partitions the space of type features into those that earn their keep and those that do not.*

### 10.2 Feature Evaluation

We evaluate concrete type system features against Japl's budget:

| **Feature** | $S_\phi$ | $C_\phi$ | $\rho$ | **Include?** |
|---|---|---|---|---|
| ADTs (sum types) | 0.85 | 0.15 | 5.67 | Yes |
| Pattern matching | 0.80 | 0.10 | 8.00 | Yes |
| Parametric poly. | 0.70 | 0.15 | 4.67 | Yes |
| Traits/type classes | 0.65 | 0.20 | 3.25 | Yes |
| Row polymorphism | 0.45 | 0.20 | 2.25 | Yes |
| Effect types | 0.60 | 0.25 | 2.40 | Yes |
| Linear types (res.) | 0.55 | 0.20 | 2.75 | Yes |
| GADTs | 0.25 | 0.40 | 0.63 | No |
| Type families | 0.20 | 0.45 | 0.44 | No |
| Dependent types | 0.30 | 0.70 | 0.43 | No |
| HKTs (full) | 0.20 | 0.35 | 0.57 | No |

Table 6: Type feature evaluation. Japl includes features with $\rho \geq 2.0$.

### 10.3 What Japl Includes

1. **Algebraic data types.** Sum types and product types prevent null pointer errors, represent domain models precisely, and enable exhaustive pattern matching. The safety benefit is enormous; the complexity cost is low.
2. **Exhaustive pattern matching.** Catches missing cases at compile time. Negligible complexity cost for massive safety benefit.

| Property | Japl | Go | Rust | Haskell | Erlang | OCaml | Gleam |
|---|---|---|---|---|---|---|---|
| ADTs + pattern matching | ✓ |  | ✓ | ✓ | Partial | ✓ | ✓ |
| Effect tracking | ✓ |  | Partial | ✓ |  |  |  |
| Ownership for resources | ✓ |  | ✓ |  |  |  |  |
| Lightweight processes | ✓ | ✓ |  |  | ✓ | Partial | ✓ |
| Supervision trees | ✓ |  |  |  | ✓ |  | ✓ |
| Fast compilation | ✓ | ✓ |  |  | ✓ | ✓ | ✓ |
| Static binaries | ✓ | ✓ | ✓ | Partial |  | ✓ |  |
| Canonical formatter | ✓ | ✓ | ✓ |  |  | ✓ | ✓ |
| Built-in test runner | ✓ | ✓ | ✓ |  |  |  |  |
| Cross-compilation | ✓ | ✓ | ✓ |  |  |  |  |
| Distribution | ✓ |  |  |  | ✓ |  | ✓ |

Table 5: Feature comparison across languages. Japl is the only language that achieves all properties simultaneously.

3. **Parametric polymorphism.** Enables generic data structures and functions without sacrificing type safety. Well-understood, efficiently implementable.

4. **Traits (type classes).** Enable ad-hoc polymorphism (overloading) in a principled way. Japl restricts to single-parameter type classes without functional dependencies, keeping resolution decidable and predictable.

5. **Row polymorphism.** Enables structural subtyping for records without full subtype polymorphism. Allows writing functions that work on "any record with a `name` field" without inheritance.

6. **Effect types.** Track side effects in function signatures. Enable optimization (pure function elimination), documentation (what can this function do?), and safety (pure functions cannot perform I/O).

7. **Linear types for resources.** Ensure deterministic cleanup of external resources. Applied only to the resource layer, not to all values.

## 10.4 What Japl Excludes

1. **GADTs.** Generalized algebraic data types enable type-level programming but make type inference undecidable [Jones et al., 2006]. The practical use cases (length-indexed vectors, well-typed interpreters) do not justify the complexity for a general-purpose language.

2. **Type families.** Type-level functions add significant complexity to the type checker and are a common source of confusing error messages in Haskell. Most practical uses can be achieved with traits and associated types.

3. **Dependent types.** Full dependent types make type checking undecidable. While dependently typed languages like Agda [Norell, 2007] and Idris [Brady, 2013] are fascinating research vehicles, the complexity cost is prohibitive for a language targeting production use.

4. **Higher-kinded types (full).** Japl supports first-order type constructors (e.g., `List[a]`, `Option[a]`) but not higher-kinded types (e.g., a function parameterized over `f` where `f` is itself a type constructor). This limits some abstraction patterns (no generic "Monad" trait) but dramatically simplifies type inference and error messages. The `Functor` trait is provided as a special case known to the compiler, rather than as a consequence of full HKT support.

## 10.5 The Budget as Design Discipline

The type power budget is not a mathematical formula applied mechanically; the $S_\phi$ and $C_\phi$ values in Table 6 are design-team estimates informed by experience, not objective constants derived from empirical measurement. Reasonable practitioners may assign different values—for instance, proponents of strongly-typed DSLs would likely rate GADTs' safety contribution higher than 0.25. The budget's value is as a *design discipline*: it forces the question "what safety problem does this feature solve, and at what cost?" for every proposed addition to the type system.

This discipline prevents feature creep—the gradual accumulation of type system features that individually seem justified but collectively produce an incomprehensible language. Haskell's GHC has over 100 language extensions, many of which interact in surprising ways. Japl's type power budget is explicitly designed to avoid this outcome.

# 11 Observability

Runtime observability is a first-class design concern in Japl, not a third-party concern delegated to APM vendors.

## 11.1 Built-in Tracing

Japl's runtime includes a distributed tracing system compatible with the OpenTelemetry standard [OpenTelemetry, 2023]:

```
fn handle_request(req: Request)
    → Response with Io, Net, Trace =
  Trace.span("handle_request", fn →
    let user = Trace.span("auth", fn →
      authenticate(req)?
    )
    let data = Trace.span("fetch_data", fn →
      fetch_user_data(user.id)?
    )
    Response.json(200, data)
  )
```

Traces propagate across process boundaries and across nodes in a distributed cluster. The tracing system is built into the runtime, so it can capture process-level events (spawn, crash, restart) in addition to application-level spans.

## 11.2 Structured Logging

```
fn process_order(order: Order)
    → Result[Receipt, OrderError] with Io,
        Log =
  Log.info("Processing order",
    [("order_id", order.id),
     ("items", show(List.length(order.items))
        ),
     ("total", show(order.total))])
  let result = validate_and_charge(order)?
  Log.info("Order completed",
    [("order_id", order.id),
     ("receipt_id", result.id)])
  Ok(result)
```

Logging is an effect (Log), which means:

- Pure functions cannot log (they have no effect annotation for it)
- Log output can be captured and tested
- Log handlers can be swapped (e.g., JSON output for production, human-readable for development)

## 11.3 Process Inspection

```
fn debug_system() → Unit with Io =
  let procs = Process.list()
  procs ▷ List.each(fn p →
    let info = Process.info(p)
    print("PID: " ◇ show(p.id)
      ◇ " status: " ◇ show(info.status)
      ◇ " msgs: " ◇ show(info.
        message_queue_len)
      ◇ " mem: " ◇ show(info.memory))
  )
```

At runtime, any process can inspect:

- The list of all running processes
- Each process's status, message queue length, memory usage, and current function
- The supervision tree structure
- Link and monitor relationships

This is directly inspired by Erlang's `:observer` and `:sys` modules, adapted for a statically typed context.

## 11.4 Runtime Metrics

The runtime exports metrics in a standard format (Prometheus-compatible):

- `japl_process_count`: number of active processes
- `japl_message_total`: total messages sent
- `japl_gc_pause_seconds`: GC pause durations (per process)
- `japl_scheduler_utilization`: CPU utilization per scheduler thread
- `japl_memory_heap_bytes`: total immutable heap size
- `japl_memory_arena_bytes`: total resource arena size

These metrics are available without any application code, because they are generated by the runtime itself.

## 12 Case Study: Building, Testing, and Deploying a Service

We compare the experience of building, testing, and deploying a representative microservice—an HTTP API with database access, background job processing, and health monitoring—across five languages.

### 12.1 Service Description

The service implements:

1. An HTTP API with CRUD endpoints for a "users" resource
2. Database access (PostgreSQL) with connection pooling
3. A background job processor for sending welcome emails
4. Health checking and readiness probes
5. Structured logging and distributed tracing
6. Graceful shutdown

### 12.2 Implementation: JAPL

```
module Main

import Http.{Server, Router, Request,
    Response}
import Db.{Pool, query}
import Job.{Runner, enqueue}
import Log
import Trace

fn main() → Unit with Io, Net, Process =
  let config = load_config()?
  let app = Supervisor.start(
    strategy = OneForOne,
    children = [
      child("db_pool",
        fn → Pool.start(config.database)),
      child("job_runner",
        fn → Runner.start(config.jobs)),
      child("http",
        fn → Server.start(config.http,
            routes())),
    ]
  )
  Process.monitor(app)
  Process.receive()  -- block until shutdown

fn routes() → Router =
  Router.new()
  ▷ Router.get("/health", health_handler)
  ▷ Router.get("/users/:id", get_user)
  ▷ Router.post("/users", create_user)
```

```
fn create_user(req: Request)
    → Response with Io, Net, Trace =
  Trace.span("create_user", fn →
    let params = Request.json_body(req)?
    let user = query("INSERT INTO users ...",
                     [params.name, params.
                        email])?
    enqueue(SendWelcomeEmail(user))?
    Response.json(201, User.to_json(user))
  )

test "create user validates email" {
  let req = Request.mock(Post, "/users",
    Json.object([("name", "Alice"),
                 ("email", "bad")]))
  let resp = create_user(req)
  assert resp.status == 400
}
```

### 12.3 Quantitative Comparison

| Metric | JAPL | Go | Rust | Haskell | Erlang |
|---|---|---|---|---|---|
| Lines of code | 420 | 580 | 750 | 510 | 480 |
| Build time (clean) | 3s | 2s | 180s | 120s | 4s |
| Build time (incr.) | 0.4s | 0.3s | 12s | 15s | 0.5s |
| Binary size | 12MB | 14MB | 8MB | 45MB | N/A |
| Docker image | 12MB | 22MB | 15MB | 280MB | 120MB |
| Dependencies | 3 | 8 | 15 | 22 | 5 |
| Config files | 1 | 3 | 4 | 5 | 3 |
| Test setup | 0 | 1 file | 1 file | 2 files | 1 file |
| Null safety | ✓ | | ✓ | ✓ | |
| Exhaustive match | ✓ | | ✓ | ✓ | |
| Effect tracking | ✓ | | | ✓ | |
| Supervision | ✓ | | | | ✓ |
| Resource safety | ✓ | | ✓ | | |

Table 7: Quantitative comparison for the case study service. JAPL figures are projected targets based on Cranelift code generation benchmarks and comparable OCaml/Gleam compilation speeds, not measurements from a mature production compiler. Figures for Go, Rust, Haskell, and Erlang are based on comparable real-world projects.

### 12.4 Qualitative Observations

**Go** requires the most boilerplate: explicit `if err != nil` error handling, manual struct-to-JSON mapping without sum types, and manual goroutine lifecycle management. However, the development cycle is extremely fast and deployment is trivial.

**Rust** produces the smallest, fastest binary but has the longest build times by an order of magnitude. The borrow checker adds friction to database connection pool management, requiring `Arc<Mutex<T»` patterns that JAPL's process model avoids entirely.

Setting up async requires choosing between Tokio, async-std, and smol runtimes.

**Haskell** has the most concise business logic (thanks to monadic composition) but the most complex setup. Building a deployable binary requires navigating Stack/Cabal configuration, and the resulting binary is large due to GHC's runtime. Space leaks in the background job processor required profiling with GHC's heap profiler.

**Erlang** provides the best fault tolerance out of the box (supervision trees, hot code loading) but dynamic typing means type errors surface at runtime. The background job processor can be updated without downtime. However, deploying to containers requires including the entire BEAM runtime.

**Japl** combines Go's build speed and deployment simplicity with Erlang's fault tolerance and Haskell's type safety. The supervision tree handles process failures automatically, the type system catches missing error cases at compile time, and the single static binary deploys trivially.

### 12.5 Deployment Workflow

The complete deployment workflow for the Japl service:

```
$ japl test
  23 tests passed (0.8s)
  3 properties checked (1.2s, 100 cases each)

$ japl build --release --static \
    --target linux-amd64

$ docker build -t myservice:v1.2.0 .
  Step 1/3: FROM scratch
  Step 2/3: COPY main /main
  Step 3/3: ENTRYPOINT ["/main"]
  Built: 12.3MB

$ docker push myservice:v1.2.0
$ kubectl set image deploy/myservice \
    myservice=myservice:v1.2.0
```

Total time from code change to production: under 30 seconds for the build-test-package cycle, with no configuration files modified, no shared library concerns, and no runtime version compatibility issues.

## 13 Discussion

### 13.1 Is This Just Go with Better Types?

A natural reaction to Japl's design is to dismiss it as "Go with algebraic data types." This characterization, while superficially plausible, misses several fundamental differences:

1. **Process model.** Japl's concurrency model is Erlang-style processes, not Go-style goroutines. The difference is not cosmetic: Japl processes have isolated heaps, typed mailboxes, and supervision trees. Goroutines share memory by default and have no recovery mechanism beyond `defer`/`recover`.

2. **Effect tracking.** Japl's effect system provides information that Go's type system cannot express. Knowing that a function is pure enables optimizations, testing strategies, and compositional reasoning that are impossible when any function might perform I/O.

3. **Hybrid memory model.** Go uses a single GC for everything, including resources that would benefit from deterministic cleanup. Japl's dual-layer model provides GC convenience for data and ownership precision for resources.

4. **Distribution.** Japl's process model extends naturally to distributed systems with location-transparent PIDs and type-derived serialization. Go has no built-in distribution model.

### 13.2 Can This Scale?

A reasonable concern is whether Japl's compilation speed targets are achievable given a type system substantially more complex than Go's.

We argue yes, based on the following observations:

1. OCaml already demonstrates that algebraic data types, parametric polymorphism, and Hindley-Milner inference compile fast.

2. Gleam compiles quickly to BEAM bytecode despite having ADTs and pattern matching.

3. Japl's type system is deliberately bounded: by excluding GADTs, type families, and dependent types, we avoid the features that make GHC slow.

4. Effect inference is a local analysis within function bodies, not a global analysis.

5. The dual-backend strategy (Cranelift for dev, LLVM for release) ensures that development builds are always fast.

### 13.3 What About Hot Code Loading?

Erlang's hot code loading—deploying new code without stopping the system—is one of its most celebrated features. JAPL does not support hot code loading in its initial design, for several reasons:

1. Hot code loading conflicts with static typing: changing a function's type signature requires recompiling all callers, which is difficult to do atomically at runtime.
2. Modern deployment practices (rolling updates, blue-green deployments, canary releases) provide similar zero-downtime guarantees without the complexity of runtime code replacement.
3. Static binaries and fast builds enable rapid redeployment that achieves the same practical goal.

Hot code loading remains a potential future extension for specific use cases (e.g., long-running telecom systems where even millisecond interruptions are unacceptable).

### 13.4 Limitations

JAPL's design involves genuine trade-offs:

1. **No dependent types.** Some correctness properties (e.g., "this list has exactly $n$ elements") cannot be expressed in JAPL's type system. Programs requiring such guarantees must use runtime checks.
2. **GC pauses.** The immutable heap is garbage-collected, and GC introduces pause times. Per-process heaps limit the impact, but latency-critical applications may need to carefully manage process heap sizes.
3. **No shared mutable memory.** For workloads that genuinely benefit from shared-memory parallelism (e.g., large-scale numerical computation), JAPL's process model incurs message-passing overhead. An FFI to Rust or C is the escape hatch for such cases. For high-performance networking scenarios requiring zero-copy shared buffers (e.g., kernel-mapped ring buffers shared between processes), JAPL provides an `unsafe foreign` block that permits the creation of shared memory regions outside the ownership system. Code within `unsafe foreign` blocks is explicitly excluded from the compiler's safety guarantees and must be audited manually, analogous to Rust's `unsafe` blocks. This escape hatch is intentionally inconvenient to discourage casual use while remaining available for performance-critical FFI interoperation.
4. **Young ecosystem.** A new language cannot match the library ecosystems of established languages. JAPL mitigates this with a comprehensive standard library and FFI, but third-party libraries will take time to develop.

## 14 Related Theoretical Work

### 14.1 Abstract Machines and Cost Semantics

JAPL's predictable execution model is informed by work on cost semantics for functional languages. Blelloch and Greiner [Blelloch & Greiner, 1996] developed cost semantics for parallel functional programs, providing a framework for reasoning about work and span. JAPL's strict evaluation ensures that cost semantics are straightforward: the cost of evaluating an expression is the sum of the costs of evaluating its subexpressions, with no hidden costs from lazy thunk evaluation.

The CESK machine [Felleisen & Friedman, 1986] provides a foundational abstract machine model. JAPL's runtime can be understood as a collection of CESK machines (one per process) communicating via message passing, with a work-stealing scheduler [Blumofe & Leiserson, 1999] distributing machines across OS threads.

### 14.2 Real-Time Garbage Collection

Baker's work on real-time garbage collection [Baker, 1978] and subsequent developments in concurrent GC [Jones & Hosking, 2016] inform JAPL's per-process GC design. The key insight is that per-process heaps bound GC pause times by bounding heap sizes: a process's heap is proportional to its live data, not to the total system's live data.

### 14.3 Work-Stealing Schedulers

JAPL's process scheduler is based on the work-stealing paradigm [Blumofe & Leiserson, 1999], adapted for the Erlang-style process model. Each OS thread maintains a local run queue of Japl processes; when a thread's queue is empty, it steals from another thread's queue. This provides load balancing without a centralized scheduler, enabling scalability to many cores.

### 14.4 Process Algebra

Japl's process model can be formalized in the $\pi$-calculus [Milner, 1999], with typed mailboxes corresponding to typed channels. The restriction to single-mailbox processes (rather than multi-channel communication) simplifies the model while retaining sufficient expressiveness for practical concurrent programming. Kobayashi's work on type systems for the $\pi$-calculus [Kobayashi, 2005] provides theoretical foundations for Japl's typed process communication.

**Typed Mailbox Variance and Protocol Evolution.** Japl's typed mailboxes use *contravariant* input types: a process that can handle messages of type `A | B` can be safely referenced as `Pid[A]`, since it accepts at least `A`. This follows standard subtyping for input channels in the $\pi$-calculus. For protocol evolution—the common need to extend a running system's message vocabulary—Japl uses *open union types* via row polymorphism: a mailbox type `Pid[{Ping, Pong | r}]` accepts at least `Ping` and `Pong` but is polymorphic in the remaining messages `r`. A process can be upgraded to handle additional message variants without breaking existing senders, because the old type is a subtype of the new type under row extension. This avoids the difficulties encountered by Akka Typed's "behavior narrowing" approach, where protocol changes required rewriting the entire actor hierarchy's types.

## 15 Conclusion

We have presented the design philosophy behind Japl's seventh core principle: *Runtime Simplicity Matters as Much as Type Power.* This principle is not a compromise or a concession to practicality; it is a recognition that the purpose of a programming language is to build and deploy reliable software, and that type system expressiveness is one tool among many for achieving that goal.

The key insights of this paper are:

1. **The type power budget.** Type system features should be evaluated by their safety-per-complexity ratio. ADTs, pattern matching, traits, effect types, and targeted linear types pass this test; GADTs, type families, and dependent types do not, for a production-oriented language.
2. **The hybrid memory model.** GC for immutable data and ownership for resources is the sweet spot: it provides Erlang-like ease for the common case and Rust-like precision where it matters.
3. **Tooling is a language feature.** A single canonical formatter, test runner, package manager, and profiler, all shipping with the compiler, eliminates an entire category of friction that plagues languages with fragmented ecosystems.
4. **Deployment is a design concern.** Static binaries, cross-compilation, and minimal runtime dependencies should be designed in from the start, not retrofitted.
5. **Effect types enable optimization and simplicity simultaneously.** Tracking effects in the type system provides both safety (knowing what a function can do) and optimization opportunities (eliminating pure computations whose results are unused), creating a virtuous cycle where type system features improve rather than degrade operational quality.

The programming language community has a tendency to evaluate languages on a single axis: type system expressiveness. This paper argues for a multi-dimensional evaluation that includes compilation speed, deployment simplicity, tooling quality, runtime predictability, and observability. Japl is designed to score well on all dimensions simultaneously, demonstrating that the Pareto frontier of type safety and operational simplicity is not only reachable but worth reaching.

We believe that the next generation of successful programming languages will be defined not by how much they can express in their type systems, but by how effectively they bridge the gap between "this program type-checks" and "this program is running reliably in production."

## References

Allwood, T. O., Sheridan, K., and Jones, S. P. Finding the needle: Stack traces for GHC. In *Proceedings of the Haskell Symposium*, pp. 129–140, 2009.

Armstrong, J. Making reliable distributed systems in the presence of software errors. PhD thesis, Royal Institute of Technology, Stockholm, 2003.

Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Baker, H. G. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

Blelloch, G. E. and Greiner, J. Parallelism in sequential functional languages. In *Proceedings of FPCA '95*, pp. 226–237, 1996.

Blumofe, R. D. and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

Bytecode Alliance. Cranelift code generator. `https://cranelift.dev/`, 2020.

Donovan, A. A. A. and Kernighan, B. W. *The Go Programming Language*. Addison-Wesley, 2015.

Felleisen, M. and Friedman, D. P. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pp. 193–219, 1986.

Gleam Team. The Gleam programming language. `https://gleam.run/`, 2023.

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I. L., Toninho, B., Wadler, P., and Yoshida, N. Featherweight Go. In *Proceedings of OOPSLA*, 2020.

Happi, E. *The BEAM Book: Understanding the Erlang Runtime System*. O'Reilly Media, 2023.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. Lua 5.1 reference manual. Technical report, Lua.org, 2006.

Jones, S. P. Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, 2003.

Jones, S. P., Vytiniotis, D., Weirich, S., and Washburn, G. Simple unification-based type inference for GADTs. In *Proceedings of ICFP*, pp. 50–61, 2006.

Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC, 2nd edition, 2016.

Kelley, A. Zig: A language for system programming. `https://ziglang.org/`, 2020.

Klabnik, S. and Nichols, C. *The Rust Programming Language*. No Starch Press, 2019.

Kobayashi, N. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4):291–347, 2005.

Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO*, pp. 75–86, 2004.

Leroy, X. The OCaml system: Documentation and user's manual. Technical report, INRIA, 2014.

Marlow, S. Haskell 2010 language report. Technical report, haskell.org, 2010.

Matsakis, N. D. and Klock, F. S. The Rust language. In *Proceedings of the ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*, pp. 103–104, 2014.

Matsakis, N. D. Incremental compilation. Rust Blog, `https://blog.rust-lang.org/2016/09/08/incremental.html`, 2016.

McAdam, B. J. On the unification of substitutions in type inference. In *Proceedings of the Implementation of Functional Languages Workshop*, 1998.

Milner, R. *Communicating and Mobile Systems: The $\pi$-Calculus*. Cambridge University Press, 1999.

Mitchell, N. Leaking space: Detecting space leaks in Haskell programs. In *Proceedings of the Haskell Symposium*, 2013.

Norell, U. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, 2007.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

OpenTelemetry Authors. OpenTelemetry specification. `https://opentelemetry.io/docs/specs/otel/`, 2023.

Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

Pike, R. Go at Google: Language design in the service of software engineering. In *Proceedings of SPLASH*, 2012.

Pike, R. Simplicity is complicated. dotGo 2015, `https://www.youtube.com/watch?v=rFejpH_tAHM`, 2015.

Rust Survey Team. Annual Rust survey 2023 results. `https://blog.rust-lang.org/2024/02/19/2023-Rust-Annual-Survey-2023-results.html`, 2023.

Snoyman, M. Haskell's deployment problem. FP Complete Blog, 2017.

Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. System F with type equality coercions. In *Proceedings of TLDI*, pp. 53–66, 2007.

Thompson, K. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, 1996.

Wadler, P. and Blott, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of POPL*, pp. 60–76, 1989.

Wirth, N. *Programming in Modula-2*. Springer-Verlag, 1982.

Ichbiah, J. D., Barnes, J. G. P., Heliard, J. C., Krieg-Brueckner, B., Roubine, O., and Wichmann, B. A. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6b):1–261, 1979.