THE ART OF ERLANG

BUILDING CONCURRENT, FAULT-TOLERANT SYSTEMS

WITH THE LANGUAGE OF THE TELECOMMUNICATIONS ERA

LIGHTWEIGHT CONCURRENCY

FAULT TOLERANCE

DISTRIBUTED SYSTEMS

OTP BEHAVIOURS

ETS & MNESIA DATA STORES

PERFORMANCE & SCALABILITY

ERIK THOMPSON

The Art of Erlang

Building Concurrent, Fault-Tolerant Systems with the Language of the Telecommunications Era

Steve T. Publications

This book is available at https://leanpub.com/theartoferlang

This version was published on 2026-07-15

Contents

Building Concurrent, Fault-Tolerant Systems with the Language of the Telecommunications Era

About this book: Erlang was born in 1986 inside Ericsson to solve a problem that no other language could handle: building telephone switching systems that must run continuously for years, handle hundreds of thousands of concurrent conversations, and survive hardware failures without dropping a single call. Today, more than forty years later, the principles that made Erlang indispensable for telecommunications are the same principles that power WhatsApp's messaging infrastructure, RabbitMQ's message broker, and countless distributed systems around the world. This book takes you from the very foundations of the language through the full breadth of OTP (the Open Telecom Platform), covering every concept a serious practitioner needs: pattern matching, recursion, lightweight processes, message passing, GenServer and Supervisor behaviours, fault tolerance, distributed programming, ETS and Mnesia data stores, networking, performance optimization, and production deployment. Each chapter builds progressively from beginner to expert level, with extensive code examples, real-world case studies, and deep dives into the internal mechanisms that make Erlang unique. This is not a language reference; it is a practitioner's guide to building systems that run forever.

Introduction

In 2014, WhatsApp was processing roughly forty billion outbound messages per day. It had approximately four hundred sixty-five million monthly active users. The team behind it numbered about fifty engineers. They were running on roughly five hundred servers and eleven thousand CPU cores [1]. A single server handled one million open TCP connections. The programming language at the heart of all of this was Erlang, a language invented thirty years earlier at a Swedish telecommunications company to solve problems that most software engineers never encounter.

Most programming languages are designed for programs that start, do some work, and stop. Web frameworks process requests and return responses. Batch jobs run overnight. Mobile apps launch when you tap an icon and terminate when you switch away. Erlang was designed for the opposite kind of program: systems that must simply never stop. Telephone exchanges cannot go down for maintenance. Messaging platforms cannot lose messages. Financial trading systems cannot freeze during market hours. These are not edge cases; they are the entire point.

This book is about how Erlang makes that possible, and how you can use it to build the same kind of resilient, concurrent, distributed software in your own projects.

What Makes Erlang Different

Erlang differs from most programming languages in ways that go far beyond syntax. The differences are philosophical and structural, baked into the virtual machine itself.

Lightweight processes. In Java or Python, creating a thread is expensive. It requires allocating a stack of one megabyte or more, registering with the operating system scheduler, and managing locks to protect shared data. Erlang processes cost roughly three hundred words of memory to create and can be spawned at rates exceeding a million per second [2]. A typical production system runs tens or hundreds of thousands of concurrent processes. Each

process has its own heap, its own stack, and its own garbage collector. There is no shared memory between processes.

Message passing. Because processes share nothing, they communicate exclusively through asynchronous message passing. You send a message to another process by specifying its identifier and the message content. The receiving process pulls messages from its mailbox and processes them one at a time. This eliminates entire categories of bugs: no race conditions, no deadlocks from lock ordering, no data corruption from concurrent writes.

Let it crash. In most programming languages, error handling is explicit and local. You wrap risky operations in try-catch blocks and handle each possible exception. Erlang takes a different approach. Instead of trying to prevent every possible failure, the language assumes that failures will happen and makes it easy to recover from them. When a process crashes, it does not affect any other process. A supervisor process detects the crash, decides what to do about it, and typically restarts the failed process with a clean slate. This "let it crash" philosophy is not laziness; it is a carefully engineered strategy for building systems that are more reliable than the sum of their parts.

The BEAM virtual machine. Erlang code runs on the BEAM (Bogdan/Bjorn's Erlang Abstract Machine), a virtual machine designed specifically for massive concurrency and fault tolerance [3]. BEAM uses preemptive scheduling based on "reductions" (a measure of computational work) rather than wall-clock time, ensuring that no single process can starve others. It supports symmetric multiprocessing natively, with one scheduler thread per CPU core. Hot code swapping is built into the runtime, allowing you to update running systems without downtime.

Distributed by design. Erlang nodes communicate transparently over the network using the same message-passing primitives used locally. A process on one machine sends a message to a process on another machine using identical syntax. Links and monitors work across network boundaries. This is not an add-on feature; it is fundamental to the language model.

Who This Book Is For

This book assumes you are already comfortable with at least one programming language. You should understand basic concepts like functions, data structures, conditionals, and loops. You do not need any prior experience with

functional programming, concurrency, or distributed systems. The book builds these concepts from the ground up.

If you come from an object-oriented background (Java, C#, Python), you will find Erlang's approach to state and behavior radically different. If you come from a systems programming background (C, Rust), you will appreciate the absence of manual memory management while learning to think about computation in terms of processes rather than threads. If you are already familiar with functional programming (Haskell, Scala, Clojure), you will find the language syntax accessible and the concurrency model distinctive.

How to Read This Book

The book is organized into seventeen chapters plus a conclusion, arranged to take you from foundational concepts through production-grade techniques.

Chapters 1 through 3 establish context: the history of Erlang, its philosophical foundations, and how to set up your development environment. If you are already familiar with functional programming and just want to learn Erlang specifically, you can skim these chapters.

Chapters 4 through 7 cover the language itself: data types, pattern matching, control flow, functions, recursion, and the concurrency primitives (processes, message passing, linking, monitoring). These chapters form the core of what it means to write Erlang code.

Chapters 8 through 11 move into OTP: the framework that transforms raw Erlang into a production-ready system. You will learn about behaviours (GenServer, Supervisor, Application), supervision trees, fault tolerance strategies, application structure, and release management.

Chapters 12 through 14 cover data storage (ETS, DETS, Mnesia), distributed programming, and networking (TCP/UDP sockets, HTTP, NIFs, ports). These chapters address the practical needs of real applications: persisting data, communicating across machines, and integrating with external systems.

Chapters 15 through 16 cover quality assurance and optimization: error handling, testing frameworks, debugging tools, profiling techniques, and performance tuning.

Chapter 17 presents case studies from production systems that use Erlang at scale, drawing out the lessons that matter most.

The conclusion ties everything together and discusses the future of the language and its ecosystem.

A Note on Code Examples

Throughout this book, code examples are written for modern Erlang/OTP (version 26 and later). Where features or behaviors differ in older versions, I note them explicitly. All examples are self-contained and can be tried directly in the Erlang shell. Some examples reference modules from the standard library; these are included with every Erlang installation.

I use a consistent style: two-space indentation, meaningful variable names (Erlang convention uses uppercase first letters for variables), and comments that explain the *why* rather than the *what*. When an example introduces a new concept, I break it down line by line. As the book progresses and concepts become familiar, examples become more compact.

The Promise of This Book

By the end of this book, you will be able to:

- Write idiomatic Erlang code using pattern matching, recursion, and higher-order functions
- Design concurrent systems using lightweight processes and message passing
- Build fault-tolerant applications using OTP behaviours and supervision trees
- Store and retrieve data using ETS, DETS, and Mnesia
- Deploy distributed Erlang systems across multiple nodes
- Profile, debug, and optimize production Erlang applications
- Understand the architectural patterns used by systems like WhatsApp and RabbitMQ

More importantly, you will learn to think about software differently. Erlang teaches you to design systems as collections of independent, communicating processes rather than monolithic programs with shared state. This way of thinking is valuable even if you never write another line of Erlang code. It has

influenced Go's goroutines, Elixir's OTP adoption, Akka's actor model in the JVM ecosystem, and the concurrency models of many modern languages.

The language that was invented to keep telephone exchanges running is now keeping the world's messaging platforms, financial systems, and distributed databases online. Let us learn how.

Chapter 1: The Origins of Erlang

The Telecom Crisis of the 1980s

In the mid-1980s, Ericsson, the Swedish telecommunications giant, faced a problem that no amount of engineering talent could solve with existing tools. The company was building telephone switching systems that had to meet requirements no other software had to satisfy. These systems needed to handle hundreds of thousands of simultaneous telephone conversations. They needed to run continuously for years without interruption. They needed to survive hardware failures without dropping a single call. And they needed to be updatable in the field, because you cannot take a telephone exchange offline for maintenance.

The programming languages available at the time were not designed for any of these requirements. Assembly language was too low-level and error-prone for the complexity involved. C provided no memory safety guarantees and no built-in concurrency model. Ada was being developed for real-time systems but was still maturing. Prolog offered declarative programming but had poor performance characteristics. Pascal was structured but lacked the flexibility needed for complex distributed control logic.

Ericsson's internal situation was even more dire. The company was using over four hundred fifty different programming languages across its various divisions and projects. There was no standardization, no shared tooling, and no way to transfer expertise between teams. A programmer who worked on one switching system could not easily contribute to another because the languages, tools, and paradigms were completely different.

The specific catalyst was the development of the AXE digital telephone exchange. The AXE was Ericsson's flagship product, and its software complexity was growing beyond what any single existing language could manage. The control logic for call routing, billing, network management, and fault detection required a new approach entirely.

Joe Armstrong, Robert Virding, and Mike Williams

In 1986, three researchers at the Ericsson Computer Science Laboratory in Stockholm began work on what would become Erlang. Joe Armstrong, a British computer scientist with a background in artificial intelligence and logic programming, led the effort. Robert Virding and Mike Williams joined him, bringing complementary expertise in systems programming and language design [5].

Armstrong's initial approach was influenced by Prolog, the logic programming language that was gaining attention in academia at the time. The first version of Erlang was actually implemented as a compiler that translated Erlang source code into Prolog. This prototype proved that the basic concepts worked but also revealed serious performance limitations. Prolog's execution model was simply too slow for the real-time requirements of telephone switching.

The name "Erlang" came from Agner Krarup Erlang (1878-1929), the Danish mathematician whose work on queuing theory formed the mathematical foundation of modern telecommunications [5]. Erlang's B and C formulas are still used today to calculate the capacity requirements of telephone networks. Naming the language after him was both a tribute and a statement of intent: this language would be designed for the same domain that Erlang's mathematics described.

Erlang was not the first programming language Ericsson had developed for telecommunications. Two earlier languages preceded it: PLEX (Programming Language for EXchanges, developed in the 1970s) and KIV (a formal specification language). Both of these languages shared a common architectural insight that would carry forward into Erlang: programs should be organized as collections of modules containing processes that communicate through message passing. This architecture was itself influenced by Modula, designed by Niklaus Wirth, the inventor of Pascal.

From a Research Project to Ericsson's Secret Weapon

The transition from Prolog-based prototype to production-ready language required building an entirely new runtime system. By 1988, the research lab had moved to Ellemtel, a joint venture between LM Ericsson and Televerket (the Swedish state telecommunications authority). The team began work on

a dedicated virtual machine, which would eventually become the BEAM (originally standing for Bogdan's Erlang Abstract Machine, named after engineer Bogdan-Bjorn Palmquist).

The first production system to use Erlang was the AXD301 ATM switch, launched in the 1990s [5]. This was a major milestone: it proved that Erlang could handle the most demanding real-time requirements in the telecommunications industry. The AXD301 achieved an availability of nine "nines" (99.9999999 percent), meaning less than thirty-two milliseconds of downtime per year. This level of reliability was not just a marketing claim; it was measured and verified in production deployments.

Throughout the 1990s, Erlang spread through Ericsson's product lines. It was used in mobile switching centers, base station controllers, and network management systems. The Open Telecom Platform (OTP) was developed as a collection of libraries, design principles, and tools that made it practical to build large-scale Erlang applications. OTP included generic server implementations, supervisor frameworks, application management, and release handling. It was the framework that transformed Erlang from an interesting research language into a production-grade platform.

For more than a decade, Erlang remained proprietary software within Ericsson. The company treated it as a competitive advantage: their switches were more reliable and easier to maintain than competitors' because of the language they used to build them. Engineers outside Ericsson had almost no exposure to the language, and the technical community knew very little about its capabilities.

The Design Philosophy: Let It Crash

The design decisions that went into Erlang were driven entirely by the requirements of telecommunications software. Each feature can be traced back to a specific problem that needed solving.

Isolation. In a telephone exchange, a bug in the billing module must not bring down the call-routing module. A crash in one part of the system must not cascade to other parts. Erlang solves this through process isolation: each process has its own memory space, and processes communicate only through message passing. If a process crashes, it takes nothing with it except itself.

Concurrency. A telephone exchange handles thousands or millions of simultaneous conversations, each requiring independent state management and real-time processing. Erlang's lightweight processes make it practical to dedicate one process per conversation, eliminating the need for complex thread management and lock-based synchronization.

Soft real-time. Not all operations in a telephone exchange have the same urgency. Routing an emergency call is more important than updating usage statistics. Erlang's scheduler provides "soft real-time" guarantees: high-priority processes get scheduled promptly, but the system does not promise hard deadlines. This is sufficient for most telecommunications workloads while avoiding the complexity of hard real-time systems.

Continuous operation. Telephone exchanges cannot be shut down for software updates. Erlang supports hot code swapping at the language level, allowing new versions of modules to be loaded into a running system without stopping any processes. Combined with the OTP release handling framework, this enables zero-downtime deployments.

Fault tolerance through supervision. Instead of trying to prevent every possible failure, Erlang embraces failures as inevitable and focuses on recovery. The "let it crash" philosophy means that processes are designed to fail quickly and cleanly when something goes wrong. Supervisor processes monitor their children, detect crashes, and restart failed processes automatically. This approach is more reliable than manual error handling because it does not depend on programmers anticipating every possible failure mode.

These design principles were so effective that they have influenced software engineering far beyond telecommunications. The actor model, popularized by languages like Akka and Scala, shares Erlang's emphasis on process isolation and message passing. Microservices architecture echoes the supervision tree pattern, with each service independently deployable and recoverable. Event-driven programming, reactive systems, and functional reactive programming all draw on ideas that Erlang pioneered decades ago.

Open Source and the Birth of the Modern Ecosystem

In December 1998, Ericsson made a decision that would change the trajectory of Erlang forever: they released it as free and open-source software [5]. The

language was placed under what would eventually become the Apache License 2.0, and the source code was made available to anyone who wanted to use it.

The reasons for this decision were practical rather than ideological. By the late 1990s, the Internet was growing rapidly, and Ericsson saw opportunities in network management, web infrastructure, and Internet telephony that extended beyond their traditional telecommunications business. Making Erlang open source would build a community of developers who could create tools, libraries, and applications that Ericsson itself could use or sell.

The open-source release was transformative. Developers who had never worked in telecommunications discovered that Erlang's concurrency model was perfectly suited for Internet-scale problems: chat servers, real-time collaboration tools, distributed databases, and high-throughput web services. The language attracted a passionate community that built an ecosystem of third-party libraries and tools.

Several projects emerged that demonstrated Erlang's potential beyond its original domain. Ejabberd, an open-source XMPP (Jabber) server written in Erlang, became one of the most widely deployed instant messaging platforms. CouchDB, a document-oriented database, used Erlang for its concurrent request handling and distributed replication. RabbitMQ, an AMQP message broker, leveraged Erlang's process model to handle millions of concurrent connections.

Perhaps the most famous adoption came from WhatsApp. When Jan Koum and Brian Acton founded WhatsApp in 2009, they chose Erlang as the backend language because of its proven ability to handle massive concurrency with minimal infrastructure. By 2014, WhatsApp was processing billions of messages daily on a fleet of roughly five hundred servers, managed by about fifty engineers. The choice of Erlang was a major factor in achieving this scale with such a small team.

The open-source era also brought new challenges. Ericsson had maintained Erlang as an internal tool for decades, and the transition to community-driven development required cultural and organizational changes. The language evolved more slowly than many developers wished, and the learning curve remained steep. But the core design proved remarkably durable: after forty years, the fundamental architecture of Erlang looks much the same as it did in 1986.

In 2011, Jose Valim created Elixir, a new programming language that runs

on the BEAM virtual machine and provides full interoperability with Erlang code. Elixir brought Ruby-like syntax to the BEAM ecosystem and attracted developers who found Erlang's Prolog-inspired syntax intimidating. Elixir did not replace Erlang; it expanded the ecosystem, bringing new users to the BEAM platform while Erlang continued to evolve as the foundational language.

Today, Erlang/OTP is maintained by Ericsson's OTP product unit, with contributions from the open-source community. The latest stable release is version 29.0.3 (as of July 2026) [4], continuing a tradition of regular, backwards-compatible updates that span more than four decades. The language that began as a solution to a specific telecommunications problem has become a general-purpose tool for building any system that demands concurrency, reliability, and continuous operation.

The story of Erlang is not just the story of a programming language. It is the story of how deep domain expertise, combined with principled software design, can produce tools that transcend their original purpose. The telephone exchanges that inspired Erlang's design are largely obsolete today, but the principles that made those exchanges reliable and maintainable are more relevant than ever in an age of distributed cloud infrastructure, real-time data processing, and globally scaled services.

Chapter 2: Philosophy and Architecture

Immutability and Pure Functions

Erlang is a functionally oriented language, and understanding what that means is essential to writing effective code. The core principle is immutability: once a value is bound to a variable, it cannot change. This is not a restriction imposed for its own sake; it is the foundation upon which Erlang's concurrency model is built.

Consider a simple example in the Erlang shell:

```
1> X = 5.
5
2> X = 10.
** exception error: no match of right hand side value 10
```

The second assignment fails because X is already bound to the value 5. In Erlang, the = operator is not an assignment operator in the traditional sense; it is a pattern matching operator. When you write `X = 5`, you are saying "the value on the right must match the pattern on the left, and any unbound variables in the pattern become bound to the corresponding values." Since X was already bound, the pattern X matches only the value 5, and the attempt to match it against 10 raises an exception.

This immutability has profound consequences for concurrent programming. In a language with shared mutable state, two threads can read and modify the same variable simultaneously, leading to race conditions, data corruption, and subtle bugs that are nearly impossible to reproduce. In Erlang, there is no shared mutable state between processes. When a process sends a message to another process, it sends a copy of the data. The receiving process works with its own private copy. There is nothing to lock, nothing to synchronize, and nothing to corrupt.

Pure functions (functions whose output depends only on their input and that have no side effects) are the natural counterpart to immutability. A pure function can be called from any process at any time, and it will always produce the same result for the same arguments. This makes pure functions trivially thread-safe and enables powerful optimizations like memoization, lazy evaluation, and parallel execution.

Erlang does not enforce purity at the language level. Functions can have side effects (writing to files, sending messages, performing I/O), and the language provides no mechanism to distinguish pure functions from impure ones. However, the convention in the Erlang community is to write as much code as possible in a pure style and to isolate side effects to well-defined boundaries. This convention, combined with immutability, makes Erlang programs easier to reason about, test, and debug than programs in languages with unrestricted mutable state.

The Actor Model and Message Passing

Erlang's concurrency model is an implementation of the actor model, a mathematical model of concurrent computation first described by Carl Hewitt at MIT in the 1970s. In the actor model, the basic unit of computation is an "actor" (called a "process" in Erlang). Each actor has:

- A private state (its memory)
- A mailbox (a queue of incoming messages)
- The ability to send messages to other actors
- The ability to create new actors
- The ability to designate how the next message should be handled

Actors do not share memory. They communicate exclusively through asynchronous message passing. When an actor sends a message, it places the message in the recipient's mailbox and continues execution immediately. The recipient processes messages from its mailbox one at a time, in the order they arrive (though it can selectively skip messages that do not match the pattern it is currently waiting for).

This model eliminates several fundamental problems of shared-memory concurrency:

No locks. Since actors do not share memory, there is no need for mutexes, semaphores, or any other locking mechanism. Deadlocks cannot occur because no actor ever waits for a lock held by another actor.

No race conditions. Since each actor processes messages sequentially, there is no possibility of two actors modifying shared state simultaneously. The only "shared" data is the message itself, and messages are copied when sent, so the sender and receiver each work with independent copies.

Natural fault isolation. If an actor crashes, it does not leave shared data in an inconsistent state because there is no shared data. The crash affects only that actor and any actors that are explicitly linked to it.

Erlang implements the actor model with remarkable efficiency. An Erlang process is not an operating system thread; it is a lightweight execution context managed entirely by the BEAM virtual machine. Creating a new process takes microseconds and consumes approximately three hundred words of memory (roughly two thousand four hundred bytes on a sixty-four-bit system). A modern server can easily run hundreds of thousands of concurrent Erlang processes.

The message-passing primitives are simple and uniform. To send a message, you use the `!` operator:

```
ReceiverPid ! {hello, world}.
```

To receive messages, you use the `receive` expression:

```
receive
    {hello, World} ->
        io:format("Hello, ~s!~n", [World]);
    Other ->
        io:format("Got something else: ~p~n", [Other])
end.
```

The `receive` expression blocks the current process until a message arrives that matches one of the patterns. When a match is found, the corresponding body is executed, and any remaining messages stay in the mailbox for future receives. This selective receive capability is powerful: it allows a process to handle different types of messages differently while ignoring messages that are not relevant to its current state.

The BEAM Virtual Machine

The BEAM (Bogdan/Bjorn's Erlang Abstract Machine) is the virtual machine that executes Erlang programs. It is a register-based virtual machine that uses direct threaded code for instruction dispatch [3]. Understanding how BEAM works is important not just for curiosity but for writing efficient code, because the language's performance characteristics are directly determined by the VM's design.

Figure 2.1: BEAM Virtual Machine Architecture

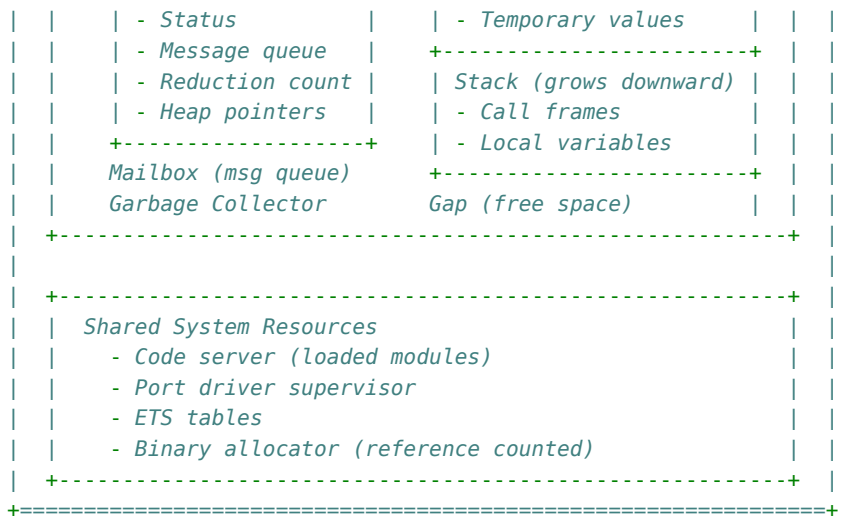

Figure 2.1: The BEAM architecture shows the compilation pipeline from Erlang source through Core Erlang intermediate representation to BEAM bytecode. At runtime, multiple scheduler threads (one per CPU core) execute Erlang processes from their run queues using reduction-based preemptive scheduling. Each process has its own Process Control Block (PCB), heap, stack, mailbox, and garbage collector. Shared system resources include the code server, port drivers, ETS tables, and binary allocator.

Compilation pipeline. When you compile an Erlang source file (`.erl`), the compiler transforms it through several stages:

1. The surface Erlang syntax is parsed and converted to **Core Erlang**, an intermediate representation that is a pure functional language with let-bindings and case expressions.
2. Core Erlang is optimized (constant folding, dead code elimination, function inlining) and then compiled to **BEAM bytecode** (`.beam` files).
3. The BEAM emulator loads the bytecode at runtime and executes it using direct threaded code, where each bytecode instruction contains a pointer to the next instruction, eliminating the need for an indirect jump through a central dispatch table.

This compilation pipeline means that most optimizations happen at compile time, not runtime. There is no just-in-time (JIT) compiler in the standard BEAM implementation, although the HiPE (High Performance Erlang) native compiler was available as an optional component in some OTP releases. The absence

of a JIT is a deliberate design choice: it keeps the runtime simple, predictable, and free from the memory overhead and unpredictability that JIT compilation introduces.

Scheduling. BEAM uses preemptive scheduling based on "reductions," which are a measure of computational work [3]. Each function call, arithmetic operation, and other basic computation counts as one or more reductions. When a process has consumed its allocation of reductions (by default, two thousand reductions per time slice), the scheduler suspends it and schedules another process to run. This ensures that no single process can monopolize the CPU, even if it is in an infinite loop.

The reduction-based scheduler is more predictable than a time-based scheduler because it measures actual computational work rather than wall-clock time. A process performing heavy computation gets fewer reductions per millisecond than a process doing mostly I/O, but the scheduler treats them equally in terms of fairness. This is appropriate for Erlang's use case: the goal is to ensure responsive message handling across all processes, not to maximize raw throughput for any single process.

BEAM supports symmetric multiprocessing (SMP) natively. When the VM starts, it creates one scheduler thread per available CPU core. Each scheduler maintains its own run queue of ready processes and executes them independently. Processes are migrated between schedulers only when necessary (for example, when a process performs I/O and the I/O driver is associated with a different scheduler). This design minimizes lock contention and enables near-linear scaling on multi-core systems.

Memory management. Each Erlang process has its own heap and stack, managed independently by the VM [3]. The heap stores the process's variables, data structures, and heap-allocated terms. The stack stores the call chain and local computation state. The default initial heap size is 233 words (approximately 1.8 kilobytes on a sixty-four-bit system) [6]. A newly spawned process uses roughly 309 words of total memory in the non-SMP emulator, with SMP support adding overhead [6]. When a process needs more heap space, the VM allocates it from a per-scheduler memory pool using a bump-pointer allocator: it simply advances a pointer, which is extremely fast.

Figure 2.2: Erlang Process Memory Layout

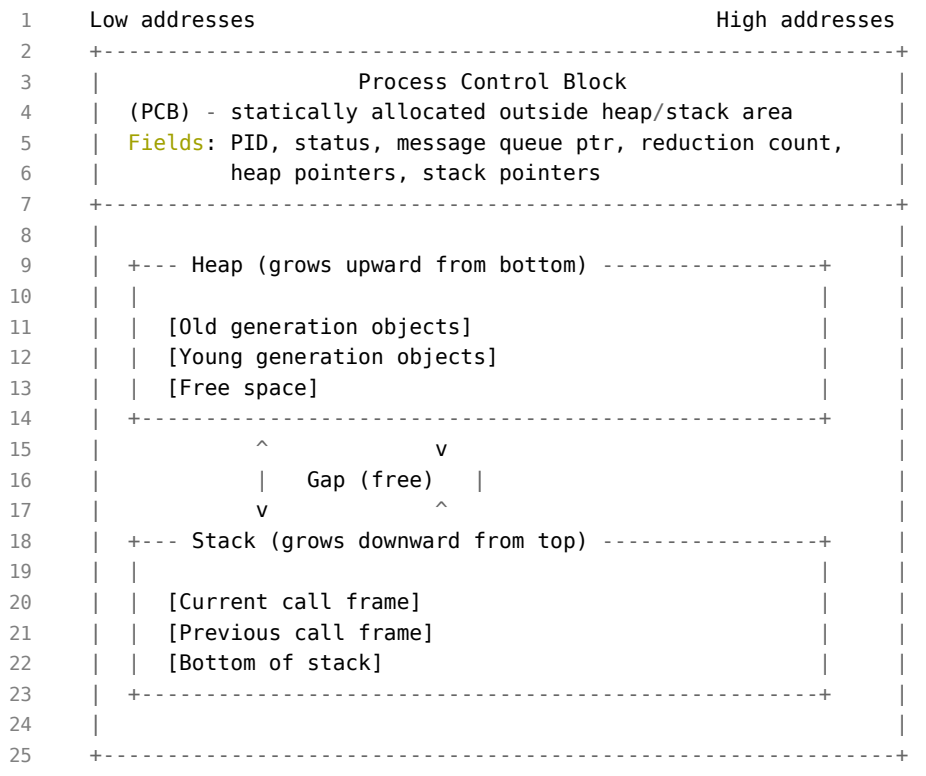

Figure 2.2: Each Erlang process has a Process Control Block (PCB) allocated separately from the heap/stack area. The heap grows upward and the stack grows downward, meeting in a gap of free space between them. This layout allows efficient growth of both regions until they meet, at which point garbage collection is triggered. The mailbox holds incoming messages as a separate linked list.

Garbage collection is generational and copying. Each process has its own garbage collector, which runs independently of other processes. The collector divides the heap into a "young" generation (recently allocated objects) and an "old" generation (objects that have survived multiple collections). When the young generation fills up, the collector copies live objects from the young generation to either a new young generation or the old generation (promoted objects). This approach is efficient because most objects die young: temporary values created during message processing are collected quickly, and only long-lived data (like cached state) survives into the old generation.

Large binaries (typically those larger than sixty-four kilobytes) are handled

differently. They are allocated outside the process heap and use reference counting for lifetime management. When a large binary is sent to another process, the reference count is incremented rather than copying the data. This makes it efficient to pass large amounts of data between processes without the overhead of deep copying.

Garbage Collection in a Concurrent World

The per-process garbage collector in BEAM has several implications for how you write Erlang code:

No stop-the-world pauses. In many virtual machines (the JVM, for example), garbage collection stops all threads while it scans and compacts the heap. In BEAM, each process's garbage collector runs independently. When one process is being garbage collected, other processes continue to run normally. This is essential for soft real-time behavior: a long GC pause in one process cannot delay message handling in another process.

GC frequency depends on allocation rate. Processes that allocate a lot of temporary data (creating many intermediate lists, tuples, and strings) will trigger garbage collection more frequently. Processes that maintain stable state with few allocations will rarely need to collect. This means that the performance impact of GC is proportional to the allocation pattern of each individual process, not a global system-wide concern.

Hibernation. When a process has been idle for a while and its message queue is empty, it can "hibernate" instead of waiting for new messages in the normal loop. Hibernation triggers a full garbage collection and shrinks the process heap to its minimum size, freeing memory for other processes. When a new message arrives, the process wakes up, allocates a fresh small heap, and begins processing. This is particularly useful for systems with many short-lived interactions (like handling individual HTTP requests) where you want idle processes to consume minimal memory.

Binary reference counting. Large binaries use reference counting rather than garbage collection. When a binary is no longer referenced by any process, its reference count drops to zero and the memory is freed immediately. This avoids the delay that would occur if a large binary had to wait for the next GC cycle in the owning process. However, it also means that cycles of references involving binaries must be broken explicitly, or memory will leak. In practice,

this is rarely a problem because Erlang's immutable data structures do not naturally create reference cycles.

Comparing Erlang to Other Languages

Understanding where Erlang fits in the broader landscape of programming languages helps clarify its strengths and trade-offs. Here are comparisons with several languages that share some characteristics with Erlang.

Erlang vs. Go. Both languages are designed for concurrent programming and are used in similar domains (networking, distributed systems, high-throughput services). The differences are fundamental:

- Go uses goroutines, which are lightweight threads managed by the Go runtime. Goroutines share memory and communicate through channels or shared variables with synchronization primitives. Erlang processes do not share memory at all.
- Go's type system is static and compiled to native code. Erlang is dynamically typed and runs on a virtual machine.
- Go's error handling uses explicit return values (multiple return values where the last is an error). Erlang uses exceptions, process crashes, and supervision trees.
- Go compiles to a single binary with no runtime dependency. Erlang requires the BEAM VM to be installed on the target system.

Go is generally easier to learn for developers coming from C, Java, or Python backgrounds. Erlang's concurrency model is more powerful for fault-tolerant systems but has a steeper learning curve.

Erlang vs. Elixir. Elixir runs on the BEAM virtual machine and is fully interoperable with Erlang. You can call Erlang functions from Elixir and vice versa. The differences are primarily syntactic and cultural:

- Elixir's syntax is influenced by Ruby, with significant whitespace, `def/do/end` blocks, and macro-based metaprogramming. Erlang's syntax is influenced by Prolog, with explicit semicolons, periods, and prefix operators.

- Elixir has a more active ecosystem for web development (Phoenix framework) and developer tooling. Erlang has a more mature ecosystem for telecommunications and embedded systems.
- Elixir's macros provide powerful compile-time metaprogramming that Erlang lacks. Erlang's macro system is limited to simple term expansion.

The choice between Erlang and Elixir is often a matter of team preference and existing expertise rather than technical capability, since both languages share the same runtime, concurrency model, and OTP framework.

Erlang vs. Haskell. Both are functional languages with strong emphasis on immutability and pure functions. The differences are philosophical:

- Haskell is lazily evaluated by default; Erlang is eagerly evaluated. This means Haskell can express infinite data structures and defer computation until values are needed, while Erlang computes everything immediately.
- Haskell has a powerful static type system with parametric polymorphism, type classes, and higher-kinded types. Erlang is dynamically typed with no compile-time type checking.
- Haskell's concurrency model (software transactional memory, STM) is based on shared memory with atomic transactions. Erlang's model is based on isolated processes with message passing.

Haskell excels at domains where type-level reasoning and mathematical correctness are paramount (compilers, financial modeling, formal verification). Erlang excels at domains where runtime resilience, concurrency, and continuous operation are paramount (telecommunications, messaging, real-time systems).

Erlang vs. Rust. Both languages emphasize safety and performance, but through very different mechanisms:

- Rust uses compile-time borrow checking to prevent data races and memory errors. Erlang uses runtime isolation (process boundaries) and garbage collection.
- Rust compiles to native code with zero-cost abstractions. Erlang runs on a virtual machine with interpretation overhead.
- Rust's concurrency model is based on owned data with explicit sharing rules. Erlang's model is based on message passing with no shared data.

Rust is generally faster for compute-intensive workloads and provides stronger compile-time guarantees. Erlang is generally more productive for building concurrent, fault-tolerant systems and provides better runtime resilience through supervision trees.

The comparison across languages reveals a pattern: Erlang trades raw computational speed and compile-time safety for runtime resilience, concurrency simplicity, and operational flexibility. For systems where uptime, availability, and graceful degradation are more important than peak throughput, this trade-off is overwhelmingly favorable.

Chapter 3: Getting Started with Erlang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Installing Erlang/OTP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Erlang Shell (REPL)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Your First Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Build Tools: Make, Rebar3, and Mix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Editor Setup and IDEs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 4: Language Fundamentals – Syntax and Data Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Variables, Atoms, and the Assignment Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Numbers: Integers, Floats, and Bignums

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Tuples, Lists, and Maps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Binaries and Bit Syntax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Operators and Expressions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 5: Pattern Matching and Control Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Power of Pattern Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Function Clauses and Clause Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Case Expressions and Multi-Clause Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Guard Expressions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

If Expressions and the Receive Construct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 6: Functions, Modules, and Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Named Functions and Module Organization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Anonymous Functions and Closures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Higher-Order Functions and the Lists Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Recursion Patterns: Tail Recursion and Accumulators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Recursive Data Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 7: Concurrency – Lightweight Processes and Message Passing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

What Is an Erlang Process?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Spawning and Communicating with Processes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Mailbox and Receive Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Linking and Trapping Exits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Monitoring and the Monitored Process Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 8: The Open Telecom Platform (OTP) – Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

What Is OTP and Why Does It Exist?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

OTP Application Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Behaviour Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

OTP Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Building Your First OTP Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 9: GenServer – The Workhorse of OTP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

What Is a GenServer?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Implementing a GenServer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Handling Calls, Casts, and Info Messages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

State Management and the Code Change Callback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Real-World GenServer Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 10: Supervisors and Fault Tolerance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Supervision Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Supervisor Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Building Robust Supervision Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Dynamic Child Processes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Fault Tolerance in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

gen_statem – Finite State Machines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

gen_event – Event Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 11: Applications, Releases, and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Application Behaviour

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Application Resource Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Release Management with Rebar3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Deployment Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Simple Module Replacement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Full Release Upgrade Walkthrough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 12: Data Storage – ETS, DETS, and Mnesia

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

ETS – In-Memory Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

DETS – Disk-Based Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Mnesia – Distributed Database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Choosing the Right Storage Mechanism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Performance and Scalability Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 13: Distributed Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Erlang Nodes and the Distributed Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Remote Procedure Calls and Node Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Distributed Processes and Global Names

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Building Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Network Partitions and Split-Brain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 14: Networking and Interoperability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

TCP and UDP Sockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Building Network Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

HTTP and Web Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

NIFs – Native Implemented Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Ports and External Program Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 15: Error Handling, Testing, and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Error Handling Philosophy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Try-Catch and Exception Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Testing with Common Test and EUnit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Property-Based Testing with PropEr

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Debugging, Tracing, and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 16: Performance Optimization and Advanced Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Understanding Erlang Performance Characteristics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Memory Management and Garbage Collection Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Compiler Flags and Code Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Design Patterns in OTP Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Modern Erlang Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Chapter 17: Real-World Applications and Case Studies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

WhatsApp and the Scale of Erlang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Riak and Distributed Databases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

RabbitMQ and Message Brokers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Heroku's Platform Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Lessons from Production Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Conclusion: The Future of Erlang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Where Erlang Stands Today

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Elixir and the BEAM Family

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

The Continuing Relevance of OTP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

Your Path Forward

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartoferlang.