ENGINEERING DISTRIBUTED SYSTEMS

ON THE

BEAM

BUILDING RELIABLE, SCALABLE APPLICATIONS WITH ERLANG/OTP AND THE ECOSYSTEM

BEAM FUNDAMENTALS	CONCURRENCY & SUPERVISION	MESSAGING & PROTOCOLS
PERSISTENCE & CONSISTENCY	FAULT TOLERANCE & RECOVERY	DEPLOYMENT & OPERATIONS

DESIGN • BUILD • DEPLOY • OPERATE • OBSERVE

FROM FIRST PRINCIPLES TO PRODUCTION SYSTEMS

MARCUS HALSTEAD

Engineering Distributed Systems on the BEAM

Building Reliable, Scalable Applications with Erlang/OTP and the Ecosystem

Steve Publications

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

This version was published on 2026-08-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

Building Reliable, Scalable Applications with Erlang/OTP and the Ecosystem

This book takes you from the fundamentals of the BEAM virtual machine through to designing, implementing, deploying, and operating production-grade distributed systems. Using a realistic reference application built incrementally throughout each chapter, you will learn how to translate distributed-system requirements into process boundaries, supervision strategies, message protocols, persistence models, and recovery mechanisms. The coverage includes BEAM distribution, clustering, consistency trade-offs, fault tolerance, observability, deployment on Kubernetes, and honest comparisons with other ecosystems. By the end, you will understand not only how to build distributed systems on the BEAM but also how to reason about their correctness, reliability, scalability, performance, security, and operational behavior in production.

Introduction

In 1986, engineers at Ericsson faced a problem that would shape an entire ecosystem: they needed a system for managing telephone switches that could run continuously for years without restarting, handle thousands of concurrent connections, survive hardware failures gracefully, and be updated without interrupting service. The result was Erlang, a language whose design decisions were dictated not by academic elegance but by the brutal requirements of carrier-grade telecommunications.

Decades later, those same design decisions make the BEAM virtual machine uniquely suited for a very different problem: building reliable distributed systems in cloud environments where networks partition, dependencies fail, and scale is measured in millions of concurrent connections rather than thousands.

This book is about translating that heritage into modern production practice. It assumes you are a competent programmer who understands concurrency concepts at some level but may be new to Erlang, Elixir, the BEAM runtime, and the engineering discipline required for distributed systems. You do not need prior functional programming experience, though familiarity will help. What you do need is willingness to think differently about failure, state, and process boundaries.

The Promise of This Book

This book makes three commitments:

First, it treats the BEAM as a production platform, not a curiosity. Every concept is grounded in practical implications: what happens when your cluster spans three availability zones, how you detect and recover from network partitions, why your mailbox is growing unbounded at 2 AM, and how to design for rolling upgrades that do not drop active connections.

Second, it builds complexity incrementally around a single reference application called Nexus, a multi-region order processing and inventory management system for an e-commerce platform. You will see process boundaries drawn, supervision trees designed, message protocols defined, state ownership assigned, persistence strategies chosen, and failure modes handled as the system evolves from a single-node prototype to a distributed cluster running across regions in Kubernetes.

Third, it is honest about trade-offs. The BEAM is not universally superior. It excels at certain classes of problems: high-concurrency workloads with many independent interactions, systems where fault tolerance matters more than raw throughput, applications that benefit from long-lived stateful processes. It is less natural for CPU-intensive computation, tight integration with existing Java or Go microservice ecosystems, or teams that prioritize developer familiarity over architectural fit. This book explains when BEAM is the right choice and when it is not.

What Makes the BEAM Different

Three properties distinguish the BEAM from conventional runtimes like the JVM, V8, or even Go:

Process isolation. Every BEAM process has its own private heap and stack. Processes communicate exclusively through message passing. There are no shared mutable variables, no locks protecting critical sections, no risk of one process corrupting another's memory. When a process crashes, it dies alone. This isolation is not an optimization; it is the foundation of fault tolerance.

Lightweight processes. A newly spawned BEAM process uses approximately 2 kilobytes of memory and takes microseconds to create. A single node can run millions of processes without exhausting system resources. This means you can model concurrency at a high level: one process per connection, one process per user session, one process per business entity. The cost of fine-grained concurrency is negligible.

Per-process garbage collection. Because each process owns its memory, garbage collection happens independently and locally. When one process pauses to collect garbage, all other processes continue running. There are no stop-the-world pauses that freeze your entire application. This property enables soft real-time guarantees: even under heavy load, responsive processes remain responsive.

These properties combine with OTP (the Open Telecom Platform), a framework of battle-tested design patterns and behaviors that turn raw concurrency into engineering discipline. OTP gives you supervisors that restart failed processes automatically, GenServers that encapsulate stateful logic in predictable interfaces, applications that compose independently deployable units, and release management that supports hot code upgrades without downtime.

Together, BEAM and OTP provide something rare: a coherent system where the runtime primitives, the framework patterns, and the operational requirements all reinforce each other instead of fighting against one another.

The Reference Application: Nexus

Throughout this book, we will design and implement Nexus, an order processing system for a fictional e-commerce platform called Northwind Retail. Northwind operates in multiple regions, handles thousands of orders per minute at peak, and cannot tolerate data loss or extended downtime during deployments.

Nexus must manage several core responsibilities:

- Accepting and validating orders through an API
- Checking and reserving inventory across warehouses
- Processing payments with external providers
- Notifying customers via email and push notifications
- Tracking order lifecycle events for analytics and auditing
- Surviving node failures, network partitions, and dependency outages without losing data or corrupting state

We will start with a single-node implementation using basic OTP behaviors. Then we will distribute it across multiple nodes, add clustering and service discovery, introduce replication and partitioning strategies, handle consistency trade-offs, implement observability, deploy to Kubernetes, and evolve the architecture toward multi-region operation.

Every design decision will be motivated by requirements and justified with reasoning about trade-offs. You will see supervision trees drawn as diagrams, message flows traced through processes, failure scenarios walked through step by step, and production configurations shown in full.

How to Read This Book

The book is organized into thirteen chapters that build on each other:

Chapters 1 through 2 establish the foundation: the BEAM execution model, lightweight processes, schedulers, garbage collection, and OTP design principles including GenServer, Supervisor, and Application behaviors. If you are new to BEAM, these chapters are essential; if you already know Erlang or Elixir well, skim them but pay attention to details about internal behavior that affect production systems.

Chapters 3 through 5 introduce the reference application and move into distribution: process boundaries for Nexus, BEAM node communication, clustering strategies, and service discovery in orchestrated environments like Kubernetes.

Chapters 6 through 9 address distributed state and messaging: where to store data, how to replicate it, consistency models, distributed transactions, event-driven architectures, backpressure, and queue processing using tools like Broadway and Oban.

Chapters 10 through 12 cover production operations: testing distributed behavior including failure scenarios, observability with OpenTelemetry and telemetry, profiling and performance tuning, building releases, deployment strategies, and disaster recovery.

Chapter 13 steps back to compare BEAM approaches with microservices, actor systems like Akka, and other ecosystems, helping you make architecture decisions grounded in honest trade-offs rather than hype.

You can read the book sequentially from start to finish, which is the recommended approach since each chapter builds on previous material. Once you have finished, individual chapters serve as reference material for specific topics: clustering setup, consistency model selection, observability configuration, or deployment procedures.

Code examples use Elixir for readability and ecosystem breadth, with Erlang shown where concepts differ significantly or where Erlang is the more natural choice. All code is designed to be internally consistent across chapters; Nexus evolves incrementally rather than being reimagined from scratch in each section. Where Gleam is relevant (particularly for type safety on BEAM), it is mentioned briefly.

Prerequisites and Tooling

To follow along with the examples, you will need:

- Erlang/OTP 26.x or later (the book targets OTP 26+ behavior; version-sensitive details are flagged where they differ)
- Elixir 1.17 or later
- Basic familiarity with functional programming concepts: immutability, pattern matching, recursion
- Comfort with command-line tools and containerized deployment (Docker, Kubernetes basics)

You do not need prior experience with distributed systems theory, though the book will introduce concepts like CAP trade-offs, consistency models, consensus algorithms, and partition handling as they become relevant to practical design decisions.

A Note on Guarantees

Distributed systems are hard because you cannot trust anything: networks drop packets, clocks drift, processes crash at unpredictable times, and dependencies fail without warning. The BEAM does not eliminate these problems; it gives you tools for reasoning about them systematically.

Throughout this book, I will distinguish carefully between guarantees (properties the system provably maintains), assumptions (conditions required for those guarantees to hold), and heuristics (practices that work well in practice but lack formal backing). Confusing assumptions for guarantees is one of the most common sources of production incidents in distributed systems.

With that understanding, let us begin by examining what makes the BEAM tick at the lowest level. Understanding the execution model is not academic; it directly affects how you design processes, structure supervision trees, tune performance, and diagnose failures in production.

Chapter 1: The BEAM Execution Model

The BEAM virtual machine is unusual among production runtimes because its design was driven by a single overriding requirement: keep a telecommunications switch running for years without restart while handling thousands of simultaneous calls, surviving hardware failures, and accepting code updates without dropping connections. Every architectural decision in BEAM traces back to that goal.

This chapter examines how BEAM works internally: the scheduler that distributes work across cores, the lightweight processes it manages, the message-passing mechanism connecting them, the garbage collector that keeps memory under control, and the isolation guarantees that make fault tolerance possible. You will learn not only what these components do but why they are designed this way, what trade-offs they encode, and how their behavior affects production systems.

Virtual Machine Architecture and Schedulers

BEAM is technically a register-based virtual machine embedded within ERTS (the Erlang Runtime System). The name BEAM evolved from earlier implementations: the original JAM (Joe's Abstract Machine) was an attempt to compile Erlang directly to C, which proved only marginally faster than interpreted execution for small programs. The successor, named BEAM after its designers Bogumil and Bjorn, became the standard implementation and has remained so for decades.

The key architectural choice is that BEAM does not map processes to operating system threads in a one-to-one relationship. Instead, it uses schedulers: C-level threads that each maintain a run queue of Erlang processes and execute them cooperatively with preemptive time-slicing based on reductions.

A reduction is an abstraction for “a unit of work” performed by the VM: a function call, a pattern match, a message send, or any other computational step. Each process has a reduction counter that increments as it performs work. When the counter reaches a limit (default 2000 in recent OTP versions), the scheduler forcibly preempts that process and selects another from the run queue. This mechanism guarantees fairness: no single process can monopolize a scheduler indefinitely, even if it enters an infinite loop or performs unbounded computation.

The number of schedulers defaults to the number of available CPU cores but is configurable via command-line flags such as `+S` for scheduling parameters and `-smp` options. In production deployments, you typically leave this at the default unless profiling reveals specific contention patterns.

BEAM's scheduler design has evolved significantly over time. Early SMP support (introduced in OTP R11B around 2006) used shared run queues that created lock contention under heavy load. Modern implementations use per-scheduler run queues with work-stealing: when a scheduler exhausts its local queue, it can steal processes from other schedulers' queues to maintain utilization. This design scales efficiently across many cores while minimizing synchronization overhead.

The scheduler also manages special threads beyond the core schedulers:

- **ASync threads** handle asynchronous I/O operations (file reads/writes, network operations) without blocking schedulers
- **Dirty schedulers** run CPU-intensive tasks that are not pure Erlang code (NIFs, port drivers) to prevent them from starving normal processes

Understanding this architecture matters for production because it explains BEAM's performance characteristics: excellent at high-concurrency I/O-bound workloads where many lightweight processes share cores cooperatively, less ideal for CPU-bound computation where dirty schedulers and NIFs can create bottlenecks.

Lightweight Processes: Creation, Cost, and Lifecycle

BEAM processes are fundamentally different from operating system threads or processes in several ways that have direct operational consequences.

A newly spawned BEAM process allocates approximately 327 words of memory (roughly 2 kilobytes on 64-bit systems), covering its initial heap, stack, and control structures. Process creation takes microseconds rather than milliseconds, making it cheap enough to spawn processes liberally without worrying about resource exhaustion from the act of spawning itself.

Each process has:

- A unique PID (process identifier) for addressing
- Its own private heap and stack that grow and shrink dynamically
- A mailbox for receiving messages
- A process dictionary for local storage
- Links and monitors connecting it to other processes
- A reduction counter managed by the scheduler

Because processes are isolated, you can create millions of them on a single node. The practical limit is memory: each process has overhead, so running 10 million idle processes would consume tens of gigabytes. In realistic production systems, however, process counts in the hundreds of thousands or low millions are routine and not inherently problematic.

Processes are created using `spawn/3` (or variants like `spawn_link`, `spawn_monitor`) which takes a module, function, and argument list. The spawned process begins executing immediately on an available scheduler:

```
pid = spawn(fn ->
  # Process logic here
  :done
end)
```

The calling function receives the PID and can send messages to it, but cannot directly access its state or memory. This isolation is deliberate: shared mutable state is the primary source of concurrency bugs in most systems, so BEAM eliminates it entirely at the language level.

Processes terminate when their executing function returns normally, when they crash with an unhandled exception, or when explicitly killed via `exit(pid, :kill)`. On termination, all memory owned by the process (heap, stack, mailbox contents) is immediately reclaimed. There is no finalization step, no destructor callback that can block, no risk of resource leaks through forgotten cleanup code.

This lifecycle has important implications for system design:

- Short-lived processes are cheap and safe to create freely
- Long-lived processes must be designed carefully to avoid unbounded memory growth
- Process death is expected and handled by supervisors rather than prevented at all costs

Mailboxes and Message Passing Semantics

Communication between BEAM processes happens exclusively through asynchronous message passing. There are no shared variables, no channels with blocking semantics (in the Go sense), no locks protecting critical sections. One process sends a message to another using its PID, and the receiving process retrieves messages from its mailbox using pattern matching.

The send operator (`!` in Elixir, `->` in Erlang) is non-blocking: it copies the message into the recipient's mailbox and returns immediately. The sender does not wait for the message to be received or processed:

```
pid <- {:request, data}
# Execution continues immediately; no waiting
```

The receiving process uses `receive` blocks to match and handle messages:

```
receive do
  {:request, data} ->
    # Handle request
    {:ok, result}
  {:cancel, id} when is_integer(id) ->
    # Handle cancellation
    :cancelled
after
  5000 ->
    # Timeout if no matching message within 5 seconds
    :timeout
end
```

Several properties of this model are important for production systems:

Ordering. Messages from a single sender arrive at the recipient in the order they were sent. Messages from multiple senders may interleave in any order. This guarantee is per-sender, not global.

Delivery. In a single node, message delivery is guaranteed: if a process sends a message to another process that exists, the message will eventually be delivered unless the recipient terminates first (in which case the message is discarded). Across distributed nodes, delivery depends on network connectivity and connection state.

Memory cost. Messages are copied into the recipient's mailbox. Large messages consume memory proportional to their size until processed and garbage collected. This means sending megabyte-sized binaries between processes can cause significant memory pressure if not handled carefully.

Mailbox growth. If a process receives messages faster than it can process them, its mailbox grows. An unbounded mailbox is a common symptom of design problems: the process is overwhelmed, a downstream dependency has failed, or there is a mismatch between producer and consumer rates. In production, monitoring mailbox sizes per process is essential for detecting these conditions early.

BEAM also supports priority messages (introduced in OTP 28) which allow certain messages to be delivered before others in the same mailbox. This feature exists but should be used sparingly: it complicates reasoning about message ordering and can introduce starvation if overused. The official guidance is explicit: very few systems genuinely need priority messaging, and those that do require careful analysis of the implications.

Immutability and Concurrency Safety

Immutability in BEAM languages is not a stylistic preference; it is enforced by the runtime. Every term created during execution cannot be modified after creation. When you "update" a variable, you are actually creating a new term and rebinding the variable name to reference it.

This property has profound consequences for concurrency:

- There is no race condition because there is nothing to race over
- There is no need for locks, mutexes, or semaphores to protect shared state
- Processes can safely pass data to each other without worrying about concurrent modification

- The only observable side effect of a process is messages it sends and processes it spawns

Immutability also enables structural sharing. When you create a new term based on an existing one (for example, prepending an element to a list), the runtime shares unchanged portions between the old and new terms rather than copying everything. This optimization makes operations like list construction extremely cheap:

```
original = [1, 2, 3, 4, 5]
new_list = [0 | original]
# original still exists as [1, 2, 3, 4, 5]
# new_list is [0, 1, 2, 3, 4, 5], sharing the tail with original
```

The trade-off is that modifying large data structures can become expensive if it requires deep copying. For example, updating a single element in a deeply nested map may require copying multiple levels of structure. In practice, this rarely becomes a bottleneck because BEAM's structural sharing and generational garbage collection handle most cases efficiently. However, understanding when immutability creates copy overhead is important for performance-sensitive code.

Garbage Collection: Generational, Concurrent, and Stop-the-World

BEAM uses a per-process generational semi-space copying garbage collector based on Cheney's algorithm with several optimizations specific to BEAM's architecture. The key insight driving this design is that process isolation enables garbage collection without stopping the entire system.

Each process has its own heap divided into two generations:

- **Young generation:** Recently allocated data, collected frequently
- **Old generation:** Data that has survived multiple collections, collected less often

When a process's heap fills up (specifically, when the stack and heap grow toward each other and collide), garbage collection triggers for that process alone. The collector copies live terms from the "from space" to the "to space", updates pointers using forwarding markers (leaves), and deallocates the old space. This happens while all other processes continue running normally on their schedulers.

The generational optimization exploits the weak generational hypothesis: most objects die young. By keeping recently allocated data separate and collecting it frequently, the collector avoids scanning long-lived data unnecessarily. Data that survives a collection is promoted to the old generation, reducing future scan overhead.

Several details matter for production systems:

Heap growth. A newly spawned process starts with a small heap (233 words). As it allocates memory, the heap grows in Fibonacci steps initially, then in approximately 20% increments after reaching about one million words. The old generation stays one stage ahead of the young generation to ensure promotion has space.

Full sweep collections. When normal generational collection cannot reclaim enough memory (for example, if a process accumulates many long-lived objects), a full sweep collection scans both generations. This is more expensive but necessary for correctness. Full sweeps also occur when explicitly triggered via `erlang:garbage_collect/1` or when spawning processes with specific options like `{fullsweep_after, N}` to force collection after N reductions.

Binary handling. Large binaries (over 64 bytes) are stored off-heap in a reference-counted binary heap. When all references to a binary are dropped, it is freed immediately rather than waiting for garbage collection. This prevents large binaries from dominating process heaps and causing expensive collections. However, excessive creation of large binaries can pressure the binary heap independently.

No global stop-the-world. Because each process has its own heap and GC runs per-process, there are no system-wide pauses that freeze all execution. This is BEAM's most distinctive advantage over JVM-style garbage collectors for latency-sensitive workloads. A single process may pause for milliseconds during collection, but the overall system remains responsive.

Tuning. GC behavior can be tuned via process flags: `min_heap_size` sets a minimum heap before GC triggers (useful for processes that allocate many small terms), `fullsweep_after` controls when full sweeps occur, and `heap_size` pre-allocates initial heap space. These are advanced tuning options that should only be adjusted based on profiling data showing specific problems.

Process Isolation as the Foundation of Fault Tolerance

Process isolation in BEAM is not merely a concurrency feature; it is the foundation of fault tolerance. Because each process has its own memory, when one process crashes due to an error (unhandled exception, bad match, arithmetic overflow), only that process dies. Its supervisor detects the crash and decides whether to restart it, but the failure does not propagate to other processes unless explicitly linked.

This property enables the "let it crash" philosophy: instead of writing extensive error-handling code to prevent every possible failure, you design systems where failures are contained within individual processes and handled at a higher level by supervisors. This approach reduces complexity dramatically because:

- Error handling logic is centralized in supervisors rather than scattered throughout workers
- Supervisors can implement sophisticated restart policies without workers knowing about them
- Crashes become observable events rather than hidden state corruption

For example, consider a web server handling thousands of concurrent requests. In a thread-per-request model with shared memory, one buggy request could corrupt shared data and affect all other requests. In BEAM, each request runs in its own process. If one request causes an error, only that process crashes; the supervisor restarts it for the next request, and no other requests are affected.

This isolation extends to garbage collection: a process performing expensive GC does not block others. It extends to scheduling: a process stuck in an infinite loop is preempted after its reduction limit, allowing other processes to continue. It extends to distribution: if one node crashes, processes on other nodes continue running (though they may need to handle the loss of their distributed peers).

The trade-off is that isolation requires explicit coordination. Processes cannot share mutable state, so coordinating complex operations requires message passing, which adds latency and complexity compared to shared-memory approaches. For certain workloads (CPU-intensive batch processing, tight numerical computation), this overhead can be significant. But for the class of problems BEAM was designed for (high-concurrency, fault-tolerant, long-running distributed systems), isolation is not a trade-off; it is the point.

Summary

The BEAM execution model provides three core guarantees that shape how you design distributed systems:

1. **Isolation:** Each process owns its memory and crashes independently
2. **Concurrency:** Millions of lightweight processes share cores cooperatively via schedulers using reduction-based preemption
3. **Responsiveness:** Per-process garbage collection prevents system-wide pauses

These guarantees are not free. They come with costs: message passing overhead instead of shared-memory efficiency, copy semantics for data transfer, explicit coordination requirements for complex operations. But for building distributed systems where reliability and availability matter more than raw throughput, the trade-offs are overwhelmingly favorable.

The next chapter builds on this foundation by introducing OTP, the framework that turns these runtime guarantees into engineering practices through supervisors, GenServers, applications, and release management.

Chapter 2: OTP Design Principles and Core Behaviours

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

The OTP Philosophy: Let It Crash and Why It Works

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

GenServer: Stateful Processes with Structured Interfaces

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

Supervisors: Trees, Strategies, and Recovery Semantics

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

Applications as Composable Units of Deployment

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

Task Behaviours: OneShot, Sync, and Async Patterns

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

Designing Process Hierarchies for a Real Application

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

Summary

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

Chapter 3: Building the Reference Application — Architecture Foundations

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

Requirements: Throughput, Latency, Availability, Consistency Goals

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

Domain Decomposition: Services, Bounded Contexts, Process Ownership

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

Supervision Topology Design for the Order System

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

Message Protocol Design: Requests, Responses, Events, Timeouts

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

State Ownership and Data Flow Architecture

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

Initial Implementation: Local Single-Node Version

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

Summary

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

Chapter 4: BEAM Distribution Fundamentals

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

Nodes and Naming: Short Names, Long Names, and DNS

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

The Distribution Protocol: epmd, Handshake, and Connection Management

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

Cookies, Authentication, and Node Security

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

Remote Process Calls and Distributed Registries

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

Links and Monitors Across Nodes

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

When BEAM Distribution Is (and Isn't) the Right Tool

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

Summary

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

Chapter 5: Clustering and Service Discovery

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

Static vs Dynamic Clusters: Trade-offs and Use Cases

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

libcluster Strategies: DNS, Kubernetes, Consul, Etcd Integration

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

Horde: Distributed ETS, Registry, and Application Controller

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

Running BEAM on Kubernetes: Pods, Services, and Node Discovery

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

Cluster Membership Changes: Joins, Leaves, Partitions

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

Security in Dynamic Clusters: TLS, mTLS, and Cookie Management

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

Summary

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

Chapter 6: State Management, Replication, and Partitioning

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

State Placement: In-Memory vs Persistent vs External Stores

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

ETS and DETS: High-Performance Local Storage

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

Mnesia: Distributed Database on BEAM — Capabilities and Limits

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

Replication Strategies: Primary-Backup, Multi-Master, Quorum-Based

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

Partitioning and Sharding: Keys, Ranges, and Hot Spots

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

Choosing Between BEAM-Native and External Databases

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

Summary

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

Chapter 7: Consistency Models and Distributed Transactions

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

The CAP Theorem in Practice: What It Actually Means for Your System

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

Consistency Models: Strong, Sequential, Causal, Eventual

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

Distributed Transactions on BEAM: Mnesia DTC and Alternatives

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

Saga Patterns and Compensating Actions for Long-Running Workflows

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

Idempotency Design: Keys, Tokens, and Deduplication

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

Retries, Timeouts, and Circuit Breakers in Concurrent Systems

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

Summary

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

Chapter 8: Fault Detection, Recovery, and Network Partitions

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

Failure Detection: Monitors, Heartbeats, and Timeouts

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

Handling Node Crashes and Process Restarts in a Cluster

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

Network Partitions: Detection, Diagnosis, and Recovery Strategies

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

Split-Brain Prevention: Quorums, Leaders, and Tiebreakers

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

Graceful Degradation: Read-Only Modes, Queuing, and Fallbacks

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

Designing for Partial Outages: Dependency Failure Handling

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

Summary

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

Chapter 9: Event-Driven Architectures and Message Flows

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

Event Sourcing vs Event-Carried State Transfer: Choosing Wisely

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

Pub/Sub on BEAM: :rpc, GenEvent, Phoenix PubSub, and Beyond

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

Queues and Workers: Oban, Broadway, and Custom Implementations

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

Backpressure and Flow Control: Preventing Overwhelm

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

RabbitMQ and External Message Brokers in BEAM Systems

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

Event Ordering Guarantees and Their Costs

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

Summary

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

Chapter 10: Testing Distributed Systems

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

Unit Testing OTP Behaviours: Isolating Processes and Time

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

Integration Testing Across Nodes: Spin-Up, Teardown, Cleanup

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

Simulating Failures: Crashes, Delays, Partitions, and Slow Links

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

Property-Based Testing for Concurrent and Distributed Code

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

Chaos Engineering on BEAM: Tools and Practices

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

Testing Idempotency, Retries, and Recovery Paths

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

Summary

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

Chapter 11: Observability, Profiling, and Performance Tuning

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

Structured Logging in Distributed BEAM Applications

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

Metrics: Counters, Gauges, Histograms, and Cardinality

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

Distributed Tracing with OpenTelemetry on BEAM

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

Profiling: Scheduler Utilization, Reductions, and Hot Paths

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

Memory Analysis: Mailbox Growth, Process Spawns, and Leaks

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

GC Tuning and Performance Optimization Techniques

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

Summary

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

Chapter 12: Deployment, Releases, and Operations

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

Building Production Releases: Mix Release, Rebar3, and Docker

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

Configuration Management: Environments, Overrides, and Hot Reloading

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

Secrets Management: Vault, KMS, and Runtime Injection

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

TLS Termination and Node-to-Node Encryption

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

Rolling Upgrades and Zero-Downtime Deployments

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

Backups, Disaster Recovery, and Multi-Region Strategies

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

Summary

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

Chapter 13: BEAM in the Modern Ecosystem — Comparisons and Trade-offs

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

BEAM vs Microservices: Monolithic Clusters vs Distributed Services

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

BEAM vs Actor Systems: Akka, Orleans, and Other Frameworks

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

BEAM in Kubernetes: Fit, Friction, and Best Practices

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

When to Choose BEAM: Strengths, Weaknesses, and Honest Trade-offs

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

Hybrid Architectures: BEAM as One Component of a Larger System

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

Architecture Decision Checklist for Distributed Systems

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

Summary

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

Conclusion

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

References

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