FEBRUARY 22, 2026 · VARIANT SYSTEMS

Process-Based Concurrency: Why BEAM and OTP Keep Being Right

A first-principles guide to process-based concurrency - what makes BEAM different, how OTP encodes resilience, and why everyone keeps reinventing it.

elixir beam otp concurrency actor-model erlang

Every few months, someone in the AI or distributed systems space announces a new framework for running concurrent, stateful agents. It has isolated state. Message passing. A supervisor that restarts things when they fail. The BEAM languages communities watch, nod, and go back to work.

This keeps happening because process-based concurrency solves a genuinely hard problem, and the BEAM virtual machine has been solving it since 1986. Not as a library. Not as a pattern you adopt. As the

Dillon Mulroy put it plainly:

Dillon Mulroy
@dillon_mulroy

pretty sure we're all just recreating OTP and the BEAM

it's actors all the way down

Thirty thousand people saw that and a lot of them felt it. The Python AI ecosystem is building agent frameworks that independently converge on the same architecture - isolated processes, message passing, supervision hierarchies, fault recovery. The patterns aren't similar to OTP by coincidence. They're similar because the problem demands this shape.

This post isn't the hot take about why Erlang was right. It's the guide underneath that take. We'll start from first principles - what concurrency actually means, why shared state breaks everything, and how processes change the game. By the end, you'll understand why OTP's patterns keep getting reinvented and why the BEAM runtime makes them work in ways other platforms can't fully replicate.

We write Elixir professionally. Our largest production system - a **healthcare SaaS platform** - runs on 80,000+ lines of Elixir handling real-time scheduling, AI-powered clinical documentation, and background job orchestration. This isn't theoretical for us. But we'll explain it like we're explaining it to ourselves when we first encountered it.

The concurrency problem, stated plainly

processing audio transcriptions while serving a real-time dashboard.

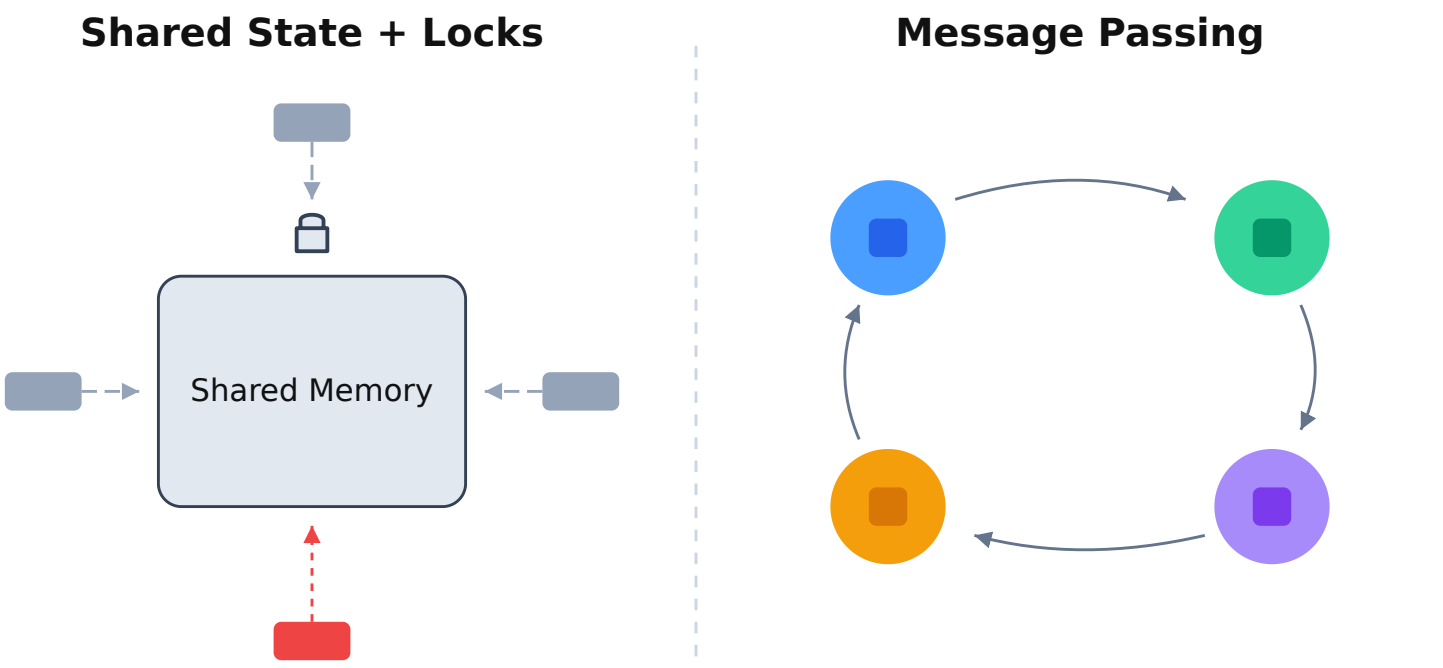
The hardware can do this. Modern CPUs have multiple cores. The question is how your programming model lets you use them.

There are two fundamental approaches.

Shared state with locks. Multiple threads access the same memory. You prevent corruption with mutexes, semaphores, and locks. This is what most languages do - Java, C++, Go (with goroutines, but shared memory is still the default model), Python (with the GIL making it worse), Rust (with the borrow checker making it safer).

The problem with shared state isn't that it doesn't work. It's that it works until it doesn't. Race conditions are the hardest bugs to reproduce, the hardest to test for, and the hardest to reason about. The more concurrent your system gets, the more lock contention slows everything down. And a single corrupted piece of shared memory can cascade through the entire system.

Isolated state with message passing. Each concurrent unit has its own memory. The only way to communicate is by sending messages. No shared memory, no locks, no races.



This is the **actor model**. Carl Hewitt proposed it in 1973. Erlang implemented it as a runtime in 1986. Every few years, the rest of the industry rediscovers it.

When BEAM programmers say “process,” they don’t mean an operating system process. OS processes are heavy - megabytes of memory, expensive to create, expensive to context-switch. They don’t mean threads either, which share memory and need synchronization. And they don’t mean green threads or coroutines, which are lighter but still typically share a heap and lack true isolation.

A BEAM process is something different:

- **~2KB of memory** at creation. You can spawn millions of them on a single machine.
- **Own heap, own stack, own garbage collector.** When a process is collected, nothing else pauses. No stop-the-world GC events affecting the entire system.
- **Preemptively scheduled.** The BEAM scheduler gives each process a budget of approximately 4,000 “reductions” (roughly, function calls) before switching to the next one. No process can hog the CPU. This happens at the VM level - you can’t opt out of it.
- **Completely isolated.** A process cannot access another process’s memory. Period. The only way to interact is by sending a message.

This last point is the one that changes how you think about software. In most languages, when something goes wrong in one part of your program, the blast radius is unpredictable. A null pointer in a thread can corrupt shared state that other threads depend on. An unhandled exception in a Node.js async handler can crash the entire process - every connection, every user, everything.

On BEAM, the blast radius of a failure is exactly one process. Always.

```
  # This process does some work...
  raise "something went wrong"
  # This process dies. Nothing else is affected.
end)

# This code continues running, unaware and unharmed
IO.puts("Still here.")
```

This isn't a try/catch hiding the error. The process that crashed is gone - its memory is reclaimed, its state is released. Everything else keeps running. The question is: who notices, and what happens next?

Message passing and mailboxes

If processes can't share memory, how do they communicate?

Every BEAM process has a mailbox - a queue of incoming messages. You send a message to a process using its process identifier (PID). The message is copied into the recipient's mailbox. The sender doesn't wait (it's asynchronous by default). The recipient processes messages from its mailbox when it's ready.

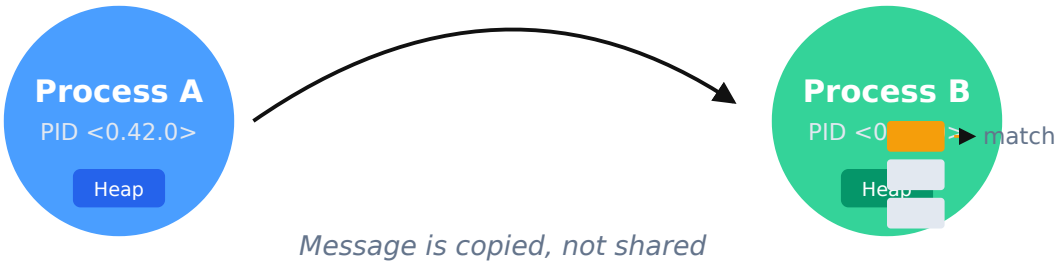
```
# Process A sends a message to Process B
send(process_b_pid, {:temperature_reading, 23.5, ~U[2026-02-22 10:00:00Z]})

# Process B receives it when ready
receive do
  {:temperature_reading, temp, timestamp} ->
    IO.puts("Got #{temp}°C at #{timestamp}")
end
```

A few things to notice:

Messages are copied, not shared. When you send a message, the data is copied into the recipient's heap. This sounds expensive, and for very large messages, it can be. But it means there's zero possibility of two processes modifying the same data. The tradeoff is worth it - you buy correctness by default.

Pattern matching on receive. The `receive` block uses Elixir's pattern matching to selectively pull messages from the mailbox. Messages that don't match stay in the mailbox for later. This means a process can handle different message types in different contexts without any routing logic.

make architectural decisions about it. Contrast this with thread-based systems where overload manifests as increasing latency, deadlocks, or OOM crashes - symptoms that are harder to diagnose and attribute.



The message-passing model creates a natural architecture. Each process is a self-contained unit with its own state, handling one thing well. Processes compose into systems through messages - like microservices, but within a single runtime, with nanosecond message delivery instead of network hops.

”Let it crash” - resilience as architecture

This is the most misunderstood concept in the BEAM ecosystem.

“Let it crash” does not mean “ignore errors.” It does not mean “don’t handle edge cases.” It means: **separate the code that does work from the code that handles failure.**

In most languages, business logic and error recovery are interleaved:

```
def process_payment(order):
    try:
        customer = fetch_customer(order.customer_id)
    except DatabaseError:
        logger.error("DB failed fetching customer")
        return retry_later(order)
```

```
    try:
        charge = payment_gateway.charge(customer, order.total)
    except PaymentDeclined:
        notify_customer(customer, "Payment declined")
        return mark_order_failed(order)
    except GatewayTimeout:
        logger.error("Payment gateway timeout")
        return retry_later(order)
    except RateLimitError:
        sleep(1)
        return process_payment(order)  # retry

    try:
        send_confirmation(customer, charge)
    except EmailError:
        logger.warning("Confirmation email failed")
        # Continue anyway? Or fail? Hard to decide here.

    return mark_order_complete(order)
```

Every function call is wrapped in error handling. The happy path - the actual business logic - is buried under defensive code. And every new failure mode adds another branch. The code becomes harder to read, harder to test, and harder to change.

On BEAM, you write the happy path:

```
defmodule PaymentProcessor do
  use GenServer

  def handle_call({:process, order}, _from, state) do
    customer = Customers.fetch!(order.customer_id)
    charge = PaymentGateway.charge!(customer, order.total)
    Notifications.send_confirmation!(customer, charge)
    {:reply, :ok, state}
  end
end
```

or escalate to a higher-level supervisor.

The business logic is clean because error recovery is a separate concern, handled by a separate process. This isn't about being reckless. It's about putting recovery logic where it belongs - in the supervision tree, not tangled into every function.

Here's the key insight: **the process that crashes loses its state, but that's fine because you designed for it.** You put critical state in a database or an ETS table. The process itself is cheap, stateless enough to restart cleanly, and focused entirely on doing its job.

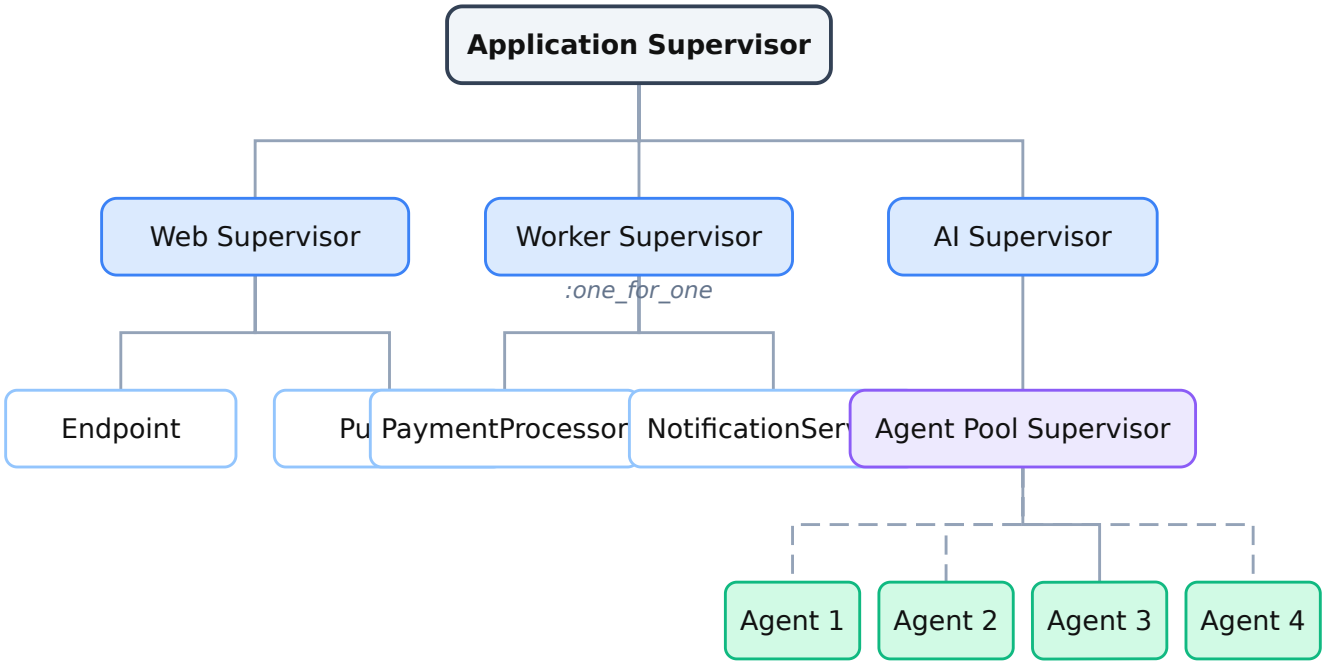
Supervision trees

A supervisor is a process whose only job is watching other processes and reacting when they die. Supervisors are organized into trees - a supervisor can supervise other supervisors, creating a hierarchy of recovery strategies.

```
defmodule MyApp.Supervisor do
  use Supervisor

  def start_link(opts) do
    Supervisor.start_link(__MODULE__, opts, name: __MODULE__)
  end

  def init(_opts) do
    children = [
      {PaymentProcessor, []},
      {NotificationService, []},
      {MetricsCollector, []}
    ]

    # If any child crashes, restart only that child
    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

The `:one_for_one` strategy means: if the `PaymentProcessor` crashes, restart it. Leave `NotificationService` and `MetricsCollector` alone. Other strategies exist:

- `:rest_for_one` - if a child crashes, restart it and all children started after it. Used when later children depend on earlier ones.



Supervisors also enforce intensity limits. You can say “restart this child up to 3 times within 5 seconds - if it keeps crashing after that, terminate this entire subtree and let my parent supervisor decide what to do.” This prevents crash loops from consuming resources indefinitely.

```
# Restart up to 3 times in 5 seconds, then give up
Supervisor.init(children, strategy: :one_for_one, max_restarts: 3, max_seconds: 5)
```

The supervision tree isn’t just an error-handling mechanism. It’s your application’s architecture diagram. When you look at a well-structured Elixir application, the supervision tree tells you:

- What components exist
- What depends on what
- What happens when each component fails

This is information that in most codebases lives in documentation (if it exists at all) or in the heads of senior engineers. In an OTP application, it's encoded in the code itself.

OTP: patterns, not a framework

OTP stands for Open Telecom Platform - a name from its Ericsson origins that nobody takes literally anymore. What OTP actually is: a set of battle-tested patterns for building concurrent systems.

The most important ones:

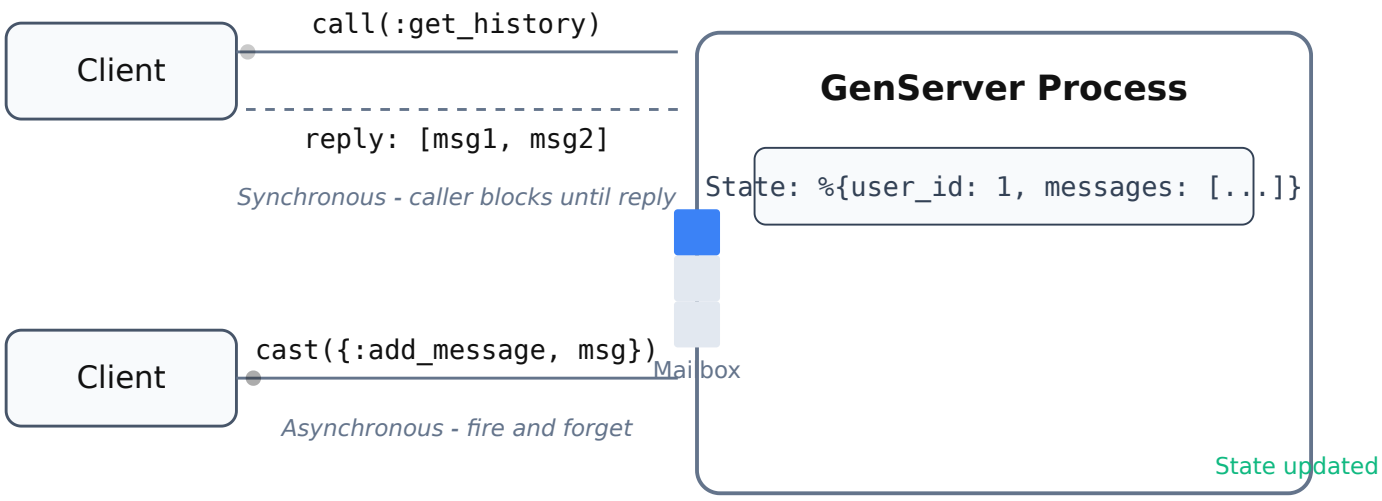
GenServer - the general-purpose stateful process

Most processes in an Elixir application are GenServers. A GenServer is a process that:

- Holds state
- Handles synchronous calls (request → response)
- Handles asynchronous casts (fire and forget)
- Handles arbitrary messages (system signals, timers, etc.)

```elixir
defmodule SessionStore do
  use GenServer

  # Client API
  def start_link(user_id) do
    GenServer.start_link(__MODULE__, %{user_id: user_id, messages: []})
  end

  def add_message(pid, message) do
    GenServer.cast(pid, {:add_message, message})
  end

  def get_history(pid) do
    GenServer.call(pid, :get_history)
  end

  # Server callbacks
  def init(state), do: {:ok, state}
```

```
  end

  def handle_call(:get_history, _from, state) do
    {:reply, Enum.reverse(state.messages), state}
  end
end
```



This is a process that holds a conversation history. You can spawn one per user session. Each one is isolated - its own memory, its own mailbox, its own lifecycle. A thousand concurrent users means a thousand of these processes, each consuming ~2KB plus whatever state they hold. The scheduler handles the rest.

Compare this to the typical approach: a shared data structure (Redis, a database table, or an in-memory map) that all request handlers read from and write to. That works, but now you need to think about cache invalidation, race conditions on writes, connection pooling to your state store, and what happens when the store goes down.

With GenServer, the state *is* the process. No external store to manage. No cache to invalidate. The process is the single source of truth for its own state.

tree, its own configuration, and its own lifecycle. Your Elixir project is itself an Application, and it depends on other Applications (Phoenix, Ecto, Oban, etc.).

When your application starts, the supervision tree starts from the root. Every process is accounted for. Nothing is floating - every process is supervised, and every supervisor is supervised, all the way up to the application root.

This is in contrast with most web frameworks where you start a server, and then various things happen at import time, module load time, and initialization time in ways that are difficult to reason about. In OTP, the startup order is explicit and hierarchical.

The runtime: why BEAM can't be replicated with libraries

Other languages can implement the actor model as a library. Akka does it for the JVM. Asyncio with some discipline can approximate it in Python. But there are runtime-level properties of the BEAM that can't be replicated without modifying the VM itself.

Preemptive scheduling

The BEAM scheduler counts reductions (roughly, function calls) for each process. After approximately 4,000 reductions, the scheduler preempts the process and switches to the next one. The process doesn't get a choice. It doesn't need to yield cooperatively.

This means: **no process can starve the system.** If one process enters an infinite loop, runs an expensive computation, or blocks on a slow operation, every other process continues running normally.

Node.js can't do this. Its event loop is cooperative - if a callback takes 500ms of CPU time, nothing else runs during those 500ms. Python with asyncio has the same limitation. Go is better (goroutines are preemptively scheduled as of Go 1.14), but goroutines share memory, which reintroduces the class of problems isolation solves.

Per-process garbage collection

Each BEAM process has its own heap and its own garbage collector. When a process's heap needs collection, only that process pauses. Every other process continues executing.

handling thousands of concurrent connections, even a 10ms pause affects every single one.

On BEAM, a process's GC pause affects exactly one connection, one session, one agent. And because processes are small (remember, ~2KB), individual collection events are tiny.

Soft real-time guarantees

The combination of preemptive scheduling and per-process GC gives the BEAM something unusual: soft real-time guarantees. Not hard real-time - this isn't an RTOS. But consistent, predictable latency across thousands of concurrent operations.

This is why WhatsApp ran 2 million connections per server on Erlang. Why Discord handles millions of concurrent users with Elixir. Why telecom switches - the original use case - require this level of reliability. And why the BEAM is naturally suited for AI agent systems where thousands of concurrent agents need responsive, isolated execution.

Hot code swapping

You can deploy new code to a running BEAM system without stopping it. Running processes continue executing old code until they make a new function call, at which point they transparently switch to the new version. No disconnected WebSockets. No dropped agent sessions. No downtime.

This isn't theoretical. Ericsson built this because telephone switches can't go down for deployments. In practice, most Elixir teams use rolling deploys instead. But the capability exists in the runtime, and for systems where connection continuity matters - long-running AI agent sessions, real-time collaborative tools, financial systems - it's a genuine differentiator.

Where this shows up today

The patterns aren't academic. They're running production systems right now.

Phoenix and LiveView

Phoenix handles HTTP and WebSocket connections as BEAM processes. Each connection is isolated. Phoenix Channels routinely handle 100,000+ concurrent WebSocket connections on a single server. LiveView - server-rendered interactive UIs - maintains a stateful process per connected user. That

Background job processing

Oban - the dominant background job library in Elixir - runs jobs as supervised processes. Failed jobs get retried by their supervisor. Job queues have backpressure through process mailboxes. Scheduled work uses OTP timers. The entire system is a supervision tree.

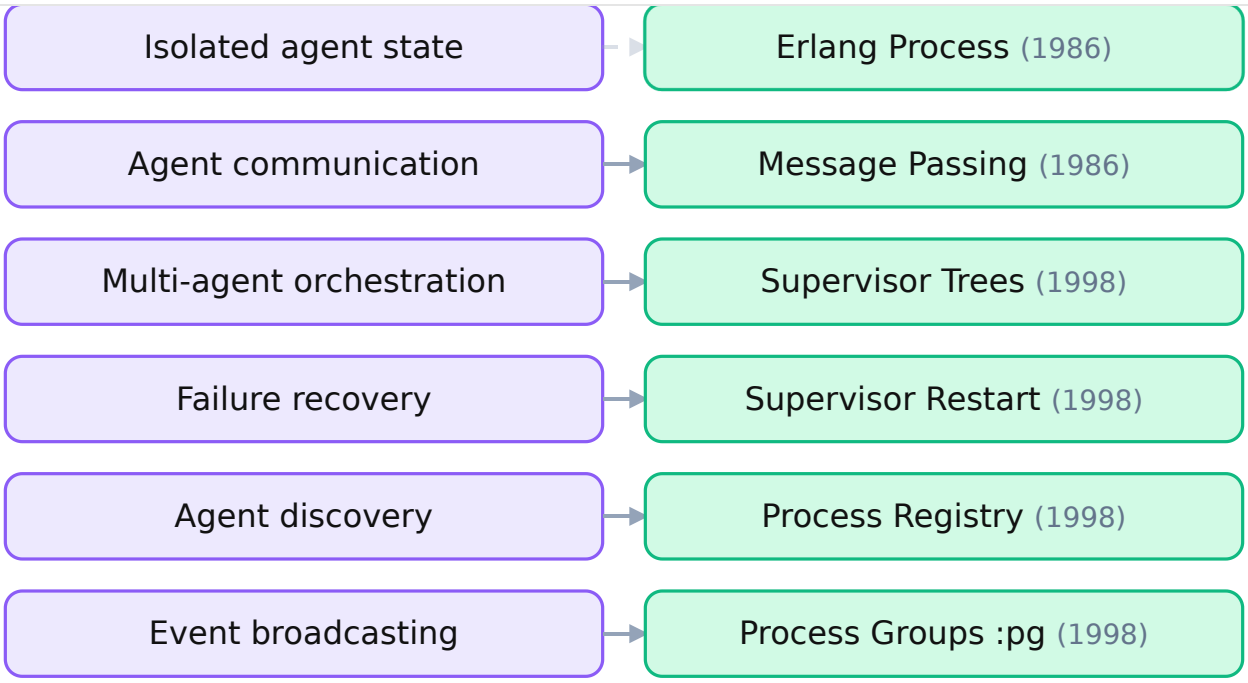
AI agents

This is the connection everyone's making right now. An AI agent is:

- **Long-lived** - maintains conversation state across multiple interactions
- **Stateful** - tracks context, memory, tool results
- **Failure-prone** - LLM API calls time out, rate limit, return malformed JSON
- **Concurrent** - you need to run thousands simultaneously

This maps directly to BEAM processes. One process per agent session. State lives in the process. Failures crash the process - a supervisor restarts it. Thousands of concurrent agents are just thousands of 2KB processes on a VM built to handle millions.

The Python ecosystem is building this with asyncio, Pydantic state models, try/except chains, and custom retry logic. It works - with significant engineering effort. But the result is the actor model implemented in userspace on a runtime that wasn't designed for it. The BEAM gives you this at the VM level, with guarantees that can't be bolted on.



George Guimarães **mapped the correspondence precisely**: isolated state is a process, inter-agent communication is message passing, orchestration is a supervision tree, failure recovery is a supervisor, agent discovery is a process registry, event distribution is process groups. All built into the runtime since the 1990s.

Elixir-native AI tooling is emerging to capitalize on this: **Jido** for agentic workflows, Bumblebee for running transformer models inside supervision trees, and LangChain bindings with step-mode execution for controlled agent pipelines.

The 30-second request problem

Traditional web frameworks (Rails, Django, Express) optimize for requests that complete in milliseconds. AI agent interactions take 5-30 seconds - an LLM call alone can take several seconds, and an agent might chain multiple calls.

Most web servers weren't built for this. A thread-per-request model with 30-second requests means you need vastly more threads to maintain throughput. Connection pools exhaust quickly. Timeouts cascade.

BEAM was designed for telephone calls - the original long-lived connections. A phone call holds state, runs for minutes, and the system handles millions of them concurrently. Replace "phone call" with "AI agent session" and the architecture is identical.

Let's be honest about the tradeoffs.

What other languages can do with effort:

- Actor model semantics (Akka, asyncio patterns, custom frameworks)
- Supervision-like patterns (process managers, health checks, Kubernetes restarts)
- Message passing (channels, queues, event buses)

A disciplined team can get 70% of what BEAM provides using Python, TypeScript, or Go with the right libraries and architecture. For many applications, that's enough.

What requires BEAM's runtime:

- True preemptive scheduling with sub-millisecond fairness
- Per-process garbage collection with zero system-wide pauses
- Process isolation enforced at the VM level (not by convention)
- Hot code swapping without disconnecting active sessions
- Millions of lightweight processes on a single node

These aren't features you can add to a runtime. They're properties of how the runtime is built. The JVM can't add per-process GC without fundamentally changing its memory model. Node.js can't add preemptive scheduling without replacing its event loop. Python can't remove the GIL without... well, they're working on that.

What BEAM doesn't do as well:

- Raw computational throughput. BEAM is not the fastest VM. For CPU-bound number crunching, the JVM or native code wins. Elixir addresses this with NIFs (native implemented functions) and libraries like Nx for numerical computing.
- Ecosystem size. Python's library ecosystem dwarfs Elixir's, especially in machine learning and data science. This is the real reason most AI frameworks are built in Python - not because Python's concurrency model is better, but because that's where PyTorch, Transformers, and the training infrastructure live.
- Learning curve. Process-based thinking requires a mental model shift. Developers from imperative backgrounds need time to internalize "let it crash" and stateless function composition. The payoff is

The recurring reinvention

The pattern keeps repeating because the problem keeps appearing.

In the 1990s, Java's threading model was supposed to be the answer to concurrent computing. It wasn't enough. Akka brought the actor model to the JVM in 2009.

In the 2010s, Node.js bet on the event loop and single-threaded async. It worked for I/O-bound web servers. It didn't work for CPU-bound work or true parallelism. Worker threads were bolted on. Still not enough for isolated, stateful concurrency.

In the 2020s, AI agent frameworks need isolated, supervised, concurrent stateful processes. AutoGen describes itself as an "event-driven actor framework." LangGraph builds state machines with shared reducers. CrewAI chains task outputs. Each one is building toward something that looks more and more like OTP - but on runtimes that weren't designed for it.

Erlang's insight in 1986 was that concurrent, fault-tolerant systems need isolation as a foundational property, not an afterthought. Every runtime that tries to bolt isolation onto a shared-memory model ends up with a system that's more complex, less reliable, and harder to reason about than one that started with isolation as the default.

The BEAM isn't the only way to build concurrent systems. But it's the most coherent one. The runtime, the language, the patterns, and the philosophy are all aligned toward the same goal. When the rest of the industry keeps independently arriving at the same architecture, that's not coincidence. That's convergence on a correct solution.

Variant Systems

Product engineering and code audits for startups and enterprises.

Company

About
Work
Blog
Contact

Explore

Services
Technologies
Industries

Connect

Twitter
LinkedIn
Email