BEAM Patterns & Anti-Patterns

The BEAM for Developers

Erik Stenman

Module Overview

- **Module 6: Designing Systems**
- **Content**
- Part 1: Gnomes
- Domains
- Flows
- **Q&A and Discussion**
- ✅ **Module 6 Checklist**

Module 6: Designing Systems

Goal

Understand the design framework:

- **Gnomes (Processes)**: When to spawn processes vs use functions
- **Domains**: Define bounded contexts, types, and transformations
- **Flows**: Design how data, messages, processes, and calls move through the system

Outcome

- Decompose systems using domain-driven design principles
- Design clear boundaries between runtime (processes) and structure (modules)
- Architect four flow dimensions: data, message, process, and call
- Apply process archetypes to build scalable, fault-tolerant systems
- Recognize and eliminate anti-patterns that fight BEAM's design

Content

Overview

You've learned how BEAM processes work, how data flows between them, how memory is managed, and how the scheduler operates. Now it's time to apply this knowledge through a design framework.

The Design Framework: Gnomes, Domains, and Flows

Gnomes = Processes:

The runtime workers. Isolated, lightweight, each with private state and mailbox. They execute code and react to messages.

Domains = Code organization:

Boundaries around data and behavior. A domain includes types (records, specs) and functions (transformations). Both data and code belong to a domain. Domains nest: functions compose into modules, modules compose into applications.

Flows = How things move:

Four dimensions describing movement through your system:

Data Flow: Payload design (references vs values), copy costs, binary handling, when data moves between heaps.

Message Flow: Protocols between processes, timeouts, retries, backpressure, mailbox health, ordering guarantees.

Process Flow: Topologies (per-entity, sharded, pooled, fan-in/fan-out), supervision trees, failure containment boundaries.

Call Flow: Cast vs call vs send, synchronous vs asynchronous patterns, coupling, latency budgets, circuit breakers.

The Relationship

Domains define entities, types, and transformations. What each piece of code is responsible for.

Processes are runtime instances that execute domain code. Where state lives and work happens.

Flows describe how data, messages, processes, and calls move across domain boundaries. The interaction patterns.

Speaker notes

Functions transform domain data. Modules group domain logic. Processes run that logic and own runtime state. Flows connect processes across domain boundaries.

Part 1: Gnomes

When to Use Gnomes vs Functions

Nouns become gnomes. Verbs become functions.

Speaker notes

Use processes (gnomes) for:

- Concurrent activities (one gnome per user connection, per room, per session)
- Fault isolation (crash one without affecting others)
- Stateful entities (shopping cart, game state, connection)

Use functions for:

- Pure computation (calculate, format, validate)
- Transformations (map, filter, reduce)

Simple rule: if you need isolation or concurrency, spawn a gnome. If you need transformation, write a function.

In the machine park, you pool threads and reuse them because they're expensive. In the gnome village, you spawn a gnome per entity because they're cheap.

One user, one gnome. One connection, one gnome. One task, one gnome. Clear ownership, clear boundaries.

This heuristic guides system decomposition. Start by identifying entities (nouns) that need isolation or state. Each becomes a process. Then identify operations (verbs) on those entities. Each becomes a function or message handler.

Processes Have Lifetimes

Hire fast, fire cleanly, rehire on failure.

- Per connection: one process per TCP/WebSocket session.
- Per call (end to end): one process per HTTP/RPC request.
- Per job (work item): one process per background task.

Speaker notes

A worker's end removes only that bench and mailbox. The process heap is freed. The PID becomes invalid. If the process was supervised, the manager spawns a replacement.

In the machine park, one broken cog can stop the whole mechanism. Threads are expensive to create and destroy. You pool them and reuse them, which means errors persist across tasks.

In the gnome village, processes are cheap. Spawning costs microseconds. Terminating is clean. Restarting gives you a fresh slate with no leaked state.

This lifecycle model is why "let it crash" works. Crashes are not catastrophic. They are transitions. A process ends, a new one starts, work continues.

Clear lifetimes prevent resource leaks and state corruption.

Process Archetypes

- **Worker:** Handles one task. Dies after. You can pool them if needed.
- **Resource owner:** Keeps state. A cart, a session, a WebSocket.
- **Router:** Receives, decides, forwards. Never stores.
- **Gatekeeper:** Limits something. One message at a time, access to a resource.
- **Observer:** watches processes, reacts to conditions

Speaker notes

Mixing these roles leads to trouble. A resource owner that also routes? A router that supervises? Possible, but you'll regret it when it's 3am and something needs refactoring.

Worker: Short-lived workers

spawn, do one job, exit. Use these for things like sending emails, updating logs, writing audit trails.

Worker: pooled workers:

Lives in a pool when not running, do one job, back to pool. Can keep some state, like config.

Router: Broadcaster

Fan out of messages

Router: Director

Route messages to the right receiver

Gatekeeper: Flow controllers

processes that gate, synchronize, or sequence other flows.

They manage contention, enforce ordering, or regulate access.

Related to gatekeeper.

Gatekeeper: Rate limiters

Provides back pressure

Gatekeeper:circuit breakers

When subsystems stall or crash, they keep the damage local.

Resource owner: Entity owner

Processes that wrap state.

A payment intent. A socket. A user.

Can be implemented with a gen_server.

Resource owner: aggregator

Collector, windowing, rollups

Observer: Sentinel

Watchdogs, deadlines, cancellations

Observer: Supervisor

Watches others. Restarts them. Never does real work.

They know who to restart and when to walk away.

Anti-Pattern: God Process

```elixir
# BAD - One process doing everything
defmodule SystemManager do
  use GenServer

  def init(_) do
    {:ok, %{
      users: %{},       # User management
      products: %{},    # Inventory
      orders: %{},      # Order processing
      payments: %{},    # Payment handling
      logs: []          # System logging
    }}
  end
end
```

Speaker notes

Why it's bad: Single scheduler thread, large heap, restart affects everything.

GenServer vs. Simple Process

When to use what

- Start with `spawn`, `receive`, clear purpose
- Need monitoring? `Task`
- Need state? `Agent`
- Need loop and mailbox? `GenServer`

Speaker notes

Don't start with GenServer. Start with `spawn`, `receive`, and a clear purpose. If you need monitoring, maybe a `Task`. If you need state, maybe an `Agent`. If you need a loop and a mailbox, then maybe GenServer.

GenServer gives you a loop. It gives you structure. But it also gives you the temptation to cram everything into one process. If it's handling state, routing, timeouts, and monitoring, you've built a God process. Those don't scale. They just grow.

Be careful with GenServers. They are useful, but not harmless. They make serialization easy. They also make bottlenecks easy. Reach for them when you need call/response logic or mailbox-driven control, not because you're used to the pattern.

Use the simplest thing that works. Keep one idea per process.

Domains

What Are Domains?

Domains define bounded areas of state and behavior in your system. This is domain modeling applied to Erlang.

Domains

Domain entities: user, room, connection, message (chat); game, player, world, entity (shooter)

Type boundaries: Moving from loose maps to typed records with specifications

Clear ownership: Each domain owns its data and transformations

Self-documenting code: Domain types make intent explicit

From Loose Maps to Typed Records

```erlang
%% Bad: Loose map, no structure
User = #{name => "Alice", age => 30, role => admin}.

%% Good: Typed record with clear fields
-record(user, {
    id :: binary(),
    name :: binary(),
    role :: admin | user | guest,
    created_at :: calendar:datetime()
}).

User = #user{
    id = generate_id(),
```

Domain types prevent bugs and make code self-documenting.

Domain Ownership

Each domain owns its data and transformations. A module per domain entity.

```erlang
%%% user_domain.erl
-module(user_domain).
-export([new/2, promote/1, can_access/2]).

-record(user, {id, name, role}).

new(Name, Role) ->
    #user{id = generate_id(), name = Name, role = Role}.

promote(#user{role = user} = User) ->
    User#user{role = admin};
    promote(User) ->
        User.
```

Behavior and data live together. Clear boundaries prevent coupling.

Domains Map to Modules and Processes

One module per domain entity: `user_domain`, `order_domain`, `session_domain`

One process instance per entity instance: One user process per connected user, one order process per active order

Domains organize the what. Processes handle the how.

System Decomposition

Example: chat server, payment router

- How to split into independent, cooperating processes
- Avoid serialization traps
- Find the boundaries

The key is finding boundaries where failure can be contained and work can be distributed.

Speaker notes

Let's look at decomposing actual systems. Take a chat server, payment router, or job queue. The goal is to split functionality into independent, cooperating processes without creating global locks or single GenServer bottlenecks.

Chat server example:

- One process per connection (handles WebSocket)
- One process per room (routes messages to members)
- One supervisor for rooms (restarts failed rooms)
- Message persistence in separate worker pool

Chat server example:

- Entity Owner: WebSocket
- Broadcaster: room
- Supervisor: room processes
- Pool worker: persistent writer

Payment router example:

- One process per payment intent (tracks state: pending, processing, completed)
- Pool of workers for API calls to payment providers
- Supervisor for each provider integration (isolate provider failures)
- Rate limiter process per provider (respects API limits)

Payment router example:

- Entity owner: payment intent
- pool worker: payment providers
- Supervisor: provider integration
- Gatekeeper Limiter+Circuit breaker: per provider

Process Ownership and Boundaries

When does a domain need its own process?

- Per-user session
- Per-order worker
- Per-connection handler

Speaker notes

System boundaries define where one concern ends and another begins. Process ownership makes those boundaries explicit.

When to create a process:

- Per-user session: isolates user state, crashes don't affect others
- Per-order worker: parallel processing, clear lifecycle
- Per-connection: one crash = one connection lost, not all

Pushback and Backpressure

Strategies:

- Drop (shed load)
- Queue (buffer temporarily)
- Reject (return error)
- Slow down (rate limit)

Speaker notes

Preventing overload by pushing back to producers. When a process receives faster than it can handle, something has to give.

Drop: Discard oldest or newest messages. Good for metrics, logs where some loss is acceptable.

Queue: Buffer in separate process or ETS. Temporary spike absorption. Risk: unbounded growth.

Reject: Return error to sender immediately. Forces sender to handle backpressure. Honest approach.

Slow down: Rate limiter upstream. Best for external APIs, databases.

Flows

Data Flow

What gets passed between processes. message

```
# Bad: Copy entire user record on every
Room ! {msg, #user{id=123, name="Alice", ...}, "Hello"}
# Good: Send references, not blobs
Room ! {msg, UserId, "Hello"}
```

Guidelines:

Prefer IDs/atoms over full records Binaries ≥64 bytes are reference-counted but still copied on send Slim messages = less copying = better performance

Message Flow

Messages carry full context, a letter should include everything to do the work safely.

```
send(pid, {:make_panini, toppings: [:mozzarella], heat:
:high})
```

Speaker notes

No reliance on remembered steps. Retry is safe. Order doesn't matter for independent tasks.

In the machine park, you configure then act. setTemperature(HIGH), then cook(). Settings interleave and bleed across threads. Thread A sets HIGH, Thread B sets LOW, Thread A's cook runs at the wrong temp.

In the gnome village, send one complete order. The gnome doesn't need to remember what you sent before. Every message is self-contained.

This eliminates races, makes retries safe, and allows message replay for debugging.

Stateful protocols (like TCP) can exist, but they should be exceptions, not the default. Even then, state lives in one well-defined place.

Complete orders over remembered steps.

Patterns:

- Timeouts on receives (avoid hangs)
- Retries for transient failures
- Backpressure (receiver tells sender to slow down)
- Monitor message_queue_len (mailbox health)

Process Flow

How many processes, how they connect.

Topologies:

- Per-entity: One process per connection
- Pooled: Fixed workers, queue incoming tasks
- Sharded: Partition by key
- Fan-out/Fan-in: One-to-many or many-to-one
- Pipeline: Staged processing

Call Flow

Synchronous vs asynchronous patterns.

```
gen_server:cast(Worker, {log, Event})
{ok, Result} = gen_server:call(Worker, {get, Key}, 5000)
Worker ! {task, Data}
```

Tradeoffs:

- Cast: Fast, no coupling, no feedback
- Call: Slower, couples sender to receiver, provides feedback
- Latency budgets: How long can you wait?
- Circuit breakers: Fail fast when subsystem is down

Supervision and Restart Strategies

Good patterns for fault tolerance

Restart strategies:

- `one_for_one`: restart only failed child
- `rest_for_one`: restart failed and all started after it
- `one_for_all`: restart all children

Speaker notes

Supervisors and restart strategies are the foundation of fault tolerance in BEAM systems.

one_for_one: Most common. Each child is independent. When one crashes, only it restarts. Use for workers that don't depend on each other (user sessions, connection handlers).

rest_for_one: Children have startup order dependencies. When one crashes, restart it and all children started after it. Use for pipelines (logger, database, cache where cache depends on database).

one_for_all: Children are interdependent. When one crashes, restart all. Use when state must be consistent across children.

If DatabaseConnection crashes, CacheServer and APIHandler restart too (they depend on DB). If APIHandler crashes, only it restarts.

Using "Let It Crash" Responsibly

When to crash, when to handle

Speaker notes

"Let it crash" doesn't mean ignore errors. It means design for recovery instead of defensive programming.

Crash on:

- Invalid input (bad data from client)
- Programming errors (nil dereference)
- Transient failures (network timeout)

Speaker notes

When to crash:

- Invalid input: `String.to_integer("abc")` should crash. Invalid data indicates caller bug or attack.
- Programming errors: Nil dereference, pattern match failure. Fix the code, don't hide the bug.
- Transient failures: Network timeout, database unavailable. Supervisor restarts with clean state.

Handle explicitly:

- Expected errors (user not found)
- Business logic (insufficient funds)
- Graceful degradation (fallback to cache)

Speaker notes

- Expected errors: User lookup returns `{:ok, user}` or `{:error, :not_found}`. Caller decides what to do.
- Business logic: Insufficient balance in transfer. Return error, don't crash payment service.
- Graceful degradation: API down, serve stale cache. Explicit fallback logic.

The key: crashes reset state. If clean restart solves it, crash. If business logic needs to respond, handle.

Common Anti-Patterns

Patterns that fight BEAM's design

- God process (single bottleneck)
- Process dictionary abuse (hidden state)
- Single-process bottlenecks (serializes work)
- Overusing GenServer (when spawn would work)

Speaker notes

God process: One GenServer handling multiple domains (users, orders, payments). Becomes serial bottleneck, large heap, restart affects everything. Solution: split by domain or shard by key.

Process dictionary: `Process.put(:user, data)` hides state. Makes code hard to test and reason about. Prefer explicit GenServer state or function arguments. Only use for library context (Logger metadata).

Single-process bottlenecks: All requests funnel through one GenServer. Prevents parallel processing. Solution: pool workers, partition by key (user_id, order_id), or rethink (do you need a process?).

Overusing GenServer: Wrapping pure functions in GenServer. Adds overhead, serializes calls. Use GenServer for stateful, long-lived processes. Use plain functions or Tasks for computation.

Decoupling Processes

Avoid the "big GenServer" trap

Symptoms:

- GenServer with >500 lines
- Handles multiple domains
- Many `handle_call` clauses
- Large state map

Solution: Split responsibilities

Speaker notes

The "big GenServer" anti-pattern happens when one process grows to handle too many concerns.

Symptoms:

- File >500 lines
- State map with unrelated keys (`%{users: ..., orders: ..., cache: ...}`)
- Many `handle_call/cast/info` clauses
- Timeout handling, validation, business logic, I/O all in one module

Solution approaches:

- **Domain split:** Separate UserServer, OrderServer, CacheServer
- **Worker pool:** One router GenServer dispatches to N workers
- **Functional core:** Move pure logic to modules, keep GenServer thin
- **Composition:** GenServer delegates to domain modules

Example refactor:

```
# BEFORE: God GenServer
def handle_call({:create_order, data}, _from, state) do
  # validate
  # check inventory
  # calculate total
  # update state
  # send email
  # log event
end

AFTER: Thin GenServer
```

```
def handle_call({:create_order, data}, _from, state) do case Orders.create(data) do {:ok, order} -> Task.start(fn -> Mailer.send_confirmation(order) end)
{:reply, {:ok, order}, state} error -> {:reply, error, state} end end
```

Domain logic in `Orders` module, side effects in `Task`, GenServer just coordinates.

Summary

Key takeaways from patterns and anti-patterns

- Design with flows and domains first
- Use process archetypes appropriately
- Decompose systems at fault boundaries
- Supervise for fault tolerance
- Avoid God processes and hidden state

Speaker notes

We've covered comprehensive patterns for BEAM development:

Design principles:

- Start with flows and tasks, not objects
- Let domains guide logic, processes execute it
- Use the five core archetypes (worker, supervisor, resource owner, router, gatekeeper)

System decomposition:

- Split at fault boundaries
- Use registries for discovery
- Implement backpressure strategies

Supervision:

- Choose appropriate restart strategies
- Use "let it crash" responsibly
- Know when to crash vs handle

Anti-patterns to avoid:

- God processes
- Process dictionary abuse
- Single-process bottlenecks
- Overusing GenServer

Message design:

- Keep messages small
- Avoid selective receive pitfalls
- Handle binaries carefully

Q&A and Discussion

1. "When should I split a GenServer?"

- Signs of overload
- Decomposition strategies
- When to keep things together

2. "How do I handle backpressure?"

- Drop, queue, reject, or slow down
- GenStage and Flow
- Custom rate limiters

3. "What's the performance cost of processes?"

- Spawn time and memory overhead
- Message copying costs
- When to optimize vs spawn more

4. "How do I test supervised processes?"

- Testing crash scenarios
- Mocking and stubbing
- Integration vs unit tests

5. "Biggest production mistake?"

- Unbounded mailboxes
- Missing supervision
- Binary leaks in long-lived processes

Thought Exercises:

- How would you design a rate limiter that works across a cluster?
- Where should validation live: in the process or before messaging?
- When is a single GenServer actually the right choice?

✅ Module 6 Checklist

Process Design Patterns

- Can design systems starting with flows and tasks
- Understands process archetypes and their proper use
- Identifies when to use short-lived vs long-lived processes
- Knows when to use GenServer vs simple processes
- Can split domains into appropriate processes

System Decomposition

- Decomposes systems at fault boundaries
- Uses process registries for discovery
- Implements appropriate supervision strategies
- Applies backpressure patterns
- Avoids single-process bottlenecks

Fault Tolerance Patterns

- Chooses correct restart strategies
- Uses "let it crash" responsibly
- Knows when to crash vs handle errors
- Designs process hierarchies for isolation
- Implements graceful degradation

Anti-Pattern Recognition

- Identifies and eliminates God processes
- Avoids process dictionary abuse
- Recognizes overuse of GenServer
- Fixes binary reference leaks
- Uses explicit state management

Memory and Message Patterns

- Designs efficient message protocols
- Avoids selective receive pitfalls
- Applies proper binary copying strategies
- Uses appropriate message sizes
- Understands reduction costs

Production Configuration

- Can configure BEAM startup parameters
- Understands build-time options
- Knows when to tune scheduler settings
- Can set appropriate memory limits

Further Reading

- *The BEAM Book*
- Erlang Efficiency Guide - Common pitfalls and optimizations
- Fred Hebert: "Erlang in Anger" - Production debugging patterns
- Saša Jurić: "Elixir in Action" - GenServer and supervision
- `recon` - Runtime inspection and debugging
- Richard Carlsson, et al: "Erlang and OTP in Action"

Question 1

What is the "God process" anti-pattern?

- One process handling multiple unrelated domains
- A process that never crashes
- Single serialization bottleneck for all operations
- A supervisor process that restarts too frequently

Question 2

When should you use a short-lived temporary worker?

- For one-off tasks that exit after completion
- When you want immediate heap reclamation
- For stateful long-running services
- Only when you have fewer than 1000 processes

Question 3

What restart strategy should you use for independent workers?

- `one_for_one` - restart only the failed child
- `one_for_all` - restart all children
- `rest_for_one` - restart failed and subsequent children
- No supervisor needed for independent workers

Question 4

What causes a binary leak in BEAM?

- Keeping a small sub-binary slice in long-lived state
- The sub-binary reference keeps entire original binary alive
- Creating too many small binaries
- Concatenating binaries with `<<B1/binary, B2/binary>>`

Question 5

How do you fix a binary reference leak?

- Use `binary:copy/1` to create independent copy of slices
- Call `erlang:garbage_collect/0` to clear accumulated references
- Copy the slice when storing in long-lived state
- Convert to a list and back to binary

Question 6

What is backpressure in process communication?

- Compressing messages before sending
- Rejecting or slowing work when a process is overloaded
- Preventing mailbox from growing unbounded
- Using higher priority processes

Question 7

When should you use GenServer instead of a simple `spawn`?

- When you need stateful request-response patterns
- Always, GenServer is always better than spawn
- When you need a structured message handling loop
- For all computational tasks

Question 8

What makes a good message design?

- Send references to large data instead of copying it
- Keep messages small when possible
- Always use GenServer.call for reliability
- Pack as much data as possible into each message

Question 9

Which supervision strategy for this pipeline: Logger -> Database -> Cache?

- `one_for_one` - they're independent
- `rest_for_one` - Cache depends on Database, both depend on Logger
- `one_for_all` - restart everything on any failure
- No supervision needed

Question 10

What's wrong with using process dictionary for state?

- Hidden state makes code hard to test and reason about
- State is implicit rather than explicit
- Process dictionary is slower than GenServer state
- Process dictionary doesn't work with supervisors

Navigation

← Previous Module: Debugging & Observability | Back to Course Structure | Next Module: Projects & Practice →