ChatGPT

Panic-Aware Actor Supervision in a Dual-Runtime Rust System

1. Supervision Decision Patterns in Actor Systems

Modern actor systems use **failure context** (panic vs error vs normal exit) to decide whether to restart an actor or escalate the fault. **Erlang/Elixir OTP** introduced a classic model: each supervised child is assigned a restart policy of **permanent**, **transient**, or **temporary** [1]. A **permanent** child is always restarted on termination (regardless of exit reason). A **transient** child is restarted **only** if it exited abnormally (i.e. due to a panic or error), but not if it stopped normally (with a normal or expected shutdown signal) [1]. And a **temporary** child is **never** restarted – even if it crashes, the failure is not recovered by its supervisor [2]. This classification lets the supervisor distinguish *transient, recoverable faults* from planned shutdowns or non-recoverable errors. For example, a configuration error might mark an actor as temporary (no restart on failure), whereas a network request handler could be transient (restart on unexpected panic).

In addition to per-child restart policies, supervisors in production systems also enforce **restart intensity limits**. OTP supervisors define a maximum number of restarts within a time window, after which the supervisor itself gives up and fails (often called *meltdown*) [3]. For instance, a supervisor might allow 3 restarts in 5 seconds; if a child keeps crashing faster than that, the supervisor terminates all its children and then exits itself [3]. This prevents infinite restart loops for persistent faults. When a supervisor dies in this way, its **parent supervisor** (if any) is notified of the failure. In a typical 2–3 level hierarchy, a higher-level supervisor may either restart the failed supervisor (to attempt a full subtree reboot) or escalate the problem up further [4]. In practice, this means transient, localized errors get handled by restarting the individual actor, but a rapid series of failures triggers an **escalation** – the supervisor is considered to have failed, and the next level up must intervene. This pattern localizes minor faults while ensuring major or persistent failures don't go unnoticed.

Rust actor frameworks mirror these concepts. For example, the *Ractor* framework's supervisor supports **Permanent**, **Transient**, and **Temporary** restart policies with semantics identical to OTP [2]. Panics or error returns are treated as "abnormal termination" for transient actors, triggering a restart, whereas a graceful stop would not be restarted under a transient policy [2]. The *Bastion* framework similarly allows defining a restart policy: **Always** restart, **Never** restart, or restart a limited number of times (*Tries(N)*) [5]. Bastion combines this with strategies for how to restart (immediately or with backoff – discussed later). These designs all use the **cause of failure** (normal exit vs. panic) and a **hierarchical limit** on frequency to decide between **restart** vs. **escalation**. In practice, a supervisor often uses a *One-for-One* strategy (only the failed actor is restarted) for most cases [6]. But if actors share state or have dependency ordering, strategies like **One-for-All** (restart all siblings on one failure) or **Rest-For-One** (restart the failed and any started **after** it) are used [6]. These strategies, combined with per-actor transient/permanent settings, let a 2–3 level tree react appropriately: a minor crash in one worker doesn't take down the whole system, while a serious fault can cascade upwards if it cannot be handled locally.

2. Trait-Based Supervision Hooks for Panic Analysis

To customize how panics and errors are handled, actor systems often provide **hook traits** or callback methods. In **Actix**, for example, actors can implement the `actix::Supervised` trait to tweak their restart behavior. Actix will automatically restart a failed actor, but before doing so it calls the actor's `restarting(&mut self, ctx)` hook to let it reset any state as needed [7] [8]. This is a simple hook allowing actor-specific logic during a restart. The `lit-bit` library's design takes a similar approach: the base `Actor` trait defines an `on_panic(&self, info: &PanicInfo) -> RestartStrategy` callback [9]. This allows each actor to specify a recommended restart strategy if it panics. By default, `on_panic` returns `RestartStrategy::OneForOne` (only restart this actor) [9], but an actor could override it. For example, an actor managing a critical resource might override `on_panic` to request an `OneForAll` strategy, indicating that if it panics, all its sibling actors should also be restarted (perhaps because they share state that may now be corrupted). This trait-based design cleanly injects domain-specific policy: the actor itself can analyze the panic (via the `PanicInfo`) or its internal state and decide how the supervisor should treat the failure.

Beyond per-actor hooks, one can design a **pluggable panic analysis trait** at the framework level. For instance, a `trait PanicAnalyzer` could be defined for supervisors, with a method like `fn should_restart(&self, child_id: ChildId, panic_info: &PanicInfo) -> bool`. This trait could encapsulate logic to inspect the panic details or error type and decide if the actor should be restarted or if the error is considered unrecoverable. Such a trait might examine, for example, the panic message or type of `ActorError` – perhaps choosing not to restart on certain fatal errors (like an out-of-memory or configuration error). Since panic payloads in Rust can be any type, a custom analyzer might attempt to downcast the payload to recognize known error strings or types [10]. The key benefit of a trait-based hook is **extensibility**: users of `lit-bit` could provide their own `PanicAnalyzer` implementation to implement project-specific policies (for example, never restart on a `ActorError::ShutdownFailure`). The supervisor would call this hook when a child fails to determine next steps. This is analogous to how one might implement custom supervision strategies in OTP by writing specific callback modules.

To support the **platform duality** (Tokio vs. Embassy) without incurring runtime overhead, `lit-bit` can leverage conditional compilation or trait specialization techniques. In fact, the library already conditionally compiles different supervision internals for each platform. For example, the `ChildInfo` struct includes a Tokio `JoinHandle` for std builds, versus a simple `is_running` flag in no-std builds [11]. This pattern continues throughout: e.g. getting the current time uses `std::time::SystemTime` on std but uses `embassy_time::Instant` when running on Embassy [12] [13]. Similarly, a `PanicAnalyzer` or hook trait could have different implementations under the hood for each platform. On a Tokio (std) build, it might utilize `PanicInfo` (which includes file/line info and the panic payload) to make decisions, whereas on an Embassy build (no-std), it might default to a simpler policy (since unwinding is typically not available). One could even use conditional methods or impl blocks for a trait – for example, an implementation of `should_restart` that only exists when the `std` feature is enabled, providing more detailed analysis, while the no-std version always returns a conservative default. The goal is to allow **platform-specific logic without duplicating the entire supervisor code**. By using feature flags and possibly generic trait bounds, `lit-bit` can invoke a panic analysis hook that is optimized for each runtime. This keeps the steady-state overhead zero-cost (the logic only runs on failure) and lets advanced users plug in analysis of panics if needed. In summary, trait-based hooks (like `Actor::on_panic` or a dedicated `PanicAnalyzer`) are a best-practice for making supervision **extensible** – enabling custom restart decisions – while conditional compilation ensures each target (Tokio or Embassy) only pays for what it can actually use.

3. Deterministic Restart Policies with Backoff

Deterministic and adaptive restart policies help prevent rapid crash-restart cycles (the classic "crash loop" scenario) in a controlled, predictable way. A common pattern is to base restart decisions on the *type or frequency of actor errors*. For example, a supervisor might maintain a **failure count** for each child and use it to decide how long to wait before restarting. This can be done in a deterministic way – e.g. an **exponential backoff** algorithm that increases the delay after each successive crash of the same actor. In a deterministic backoff, the delay might be calculated as `base_delay * 2^(failure_count)`, capping at some max. The key is that given a known failure count, the delay is predetermined (no randomness), which is important for reproducibility in embedded or real-time systems.

Production systems like **Bastion** already incorporate such backoff strategies. Bastion's `RestartStrategy` lets you specify an `ActorRestartStrategy` of *Immediate*, *LinearBackOff*, or *ExponentialBackOff*, along with parameters like timeout intervals [14]. For instance, you could configure a policy "restart at most 3 times, with a 1 second linear backoff between tries" [15] [16]. If the actor still fails after 3 attempts, Bastion will stop restarting it (consider it a fatal error) [17]. This approach ensures that transient errors trigger quick retries, but persistent failures cause an eventual give-up, avoiding endless loops. The **Ractor** supervisor crate shows another take: it allows an optional `backoff_fn` to be provided per child [18]. This is a user-defined function that the supervisor will call to compute a delay before restarting the child, enabling completely custom backoff logic (exponential, jitter, etc.) [18]. Ractor also defines a `reset_after` duration – if a child runs successfully for a certain period, the supervisor can reset that child's failure count to avoid penalizing it for ancient history [19]. All these measures contribute to a **deterministic yet adaptive** policy: the rules (max N restarts in M seconds, backoff delays) are fixed in configuration, so the behavior is predictable, but they adapt to the runtime conditions by, say, lengthening the pause after repeated failures.

For `lit-bit`, which targets embedded determinism, an adaptive backoff can be implemented without introducing nondeterminism. Using a monotonic timer (via the `SupervisorTimer` trait or embassy's time API), the supervisor can schedule restarts at calculated future times. For example, on each failure, record a timestamp of the first failure in a window and a count. If the actor fails *n* times quickly, the supervisor might decide to **wait** (back off) instead of immediate restart. This could be as simple as not restarting until a certain interval has passed since the last failure. Because `lit-bit` is zero-cost in steady state, it likely wouldn't spawn a separate timer task for backoff; instead, it can check the elapsed time in its normal polling loop to decide if the backoff period has elapsed. Another deterministic strategy is to escalate after a fixed number of failures: for instance, "if this actor fails 5 times consecutively within 60 seconds, do not restart again" – which is exactly the OTP intensity mechanism [20]. Indeed, `lit-bit` already has fields like `max_restarts` and `restart_window_ms` in its `SupervisorActor` configuration [21] to support this. The next step is to incorporate **delays** between restart attempts. An exponential backoff could be implemented by simply not invoking the restart immediately in `execute_restarts`, but instead scheduling it (perhaps by re-inserting a child with a timer). In an async runtime, this can be achieved with a delayed future (e.g. `tokio::sleep`) before actually respawning the actor. In an Embassy context, one might use `embassy_time::Timer` for the same effect. Notably, any such delay should itself be deterministic and based on counters/timestamps – no random jitter unless explicitly desired (and even then, one could use a pseudo-random generator with a fixed seed for reproducibility if needed).

The benefit of a backoff is seen in scenarios like hardware flakiness or network outages in embedded systems. A sensor actor that panics due to a transient I/O error might be restarted immediately the first time, but if it panics repeatedly, an exponential backoff will give the hardware time to recover (or avoid flooding logs). Meanwhile, **in steady state (no failures), these backoff mechanisms impose no cost** – e.g. the supervisor just tracks a few integers per actor. Only when a failure occurs do we incur the overhead of computing a delay or scheduling a timer. This aligns with `lit-bit`'s design principle of zero-cost in the common case. In summary, adopting deterministic restart policies means defining exact rules (like "exponential backoff with base 100ms" or "at most 5 restarts per minute") so that anyone reading the supervisor configuration can predict its behavior. This determinism is crucial for embedded and safety-critical applications where random exponential jitter might be unacceptable. By leveraging the approaches proven in systems like OTP (intensity limits) and Bastion (backoff strategies) [22] [23], the Phase 2 implementation can ensure actors are restarted in a controlled manner and that chronic failures trigger either longer waits or graceful escalation rather than a tight crash loop.

4. Hierarchical Escalation and Supervision Tree Integration

In a robust actor system, supervisors themselves can be children of higher supervisors, forming a **supervision tree**. This hierarchy means that if a lower-level supervisor cannot handle a failure, it will escalate the problem to its parent. The classic pattern in OTP is that if a supervisor hits a meltdown (too many restarts in a short time) or a child with an unrecoverable policy (e.g. a **temporary** child that died) causes it to terminate, that supervisor's **exit** is treated just like any other process exit [4]. Its parent (another supervisor) sees that the child (which in this case is itself a supervisor process) has failed. The parent can have its own strategy about this: it might be configured to always restart that child supervisor (treating it as permanent), or it might decide not to restart (if that supervisor was marked temporary at the higher level). In practice, one common approach is to **not automatically restart a failing sub-supervisor** unless there's a clear recovery strategy. Instead, you might want a more managed approach to bringing that subtree back.

A known OTP pattern is the *"supervisor who supervises supervisors"* with mixed restart settings. For example, one might mark worker processes as **permanent/transient** (so they always get restarted by their immediate supervisor on failure), but mark the **supervisor of those workers as temporary** in its own parent's child spec [24] [25]. That means if the whole group of workers fails too often and the worker-supervisor gives up (dies), the higher-level supervisor will **not** try to restart it automatically [24] [26]. Why do this? It allows a deliberate escalation: a higher-level logic can decide *when or if* to restart that part of the system. In OTP, this is often accomplished with an external "manager" process. As described in *Adopting Erlang*, one can insert a **manager process** alongside a temporary supervisor [24] [27]. If the temporary supervisor dies (after too many failures), the top-level supervisor simply leaves it dead (because it's temporary, it won't be auto-restarted) [24]. The separate manager process (not a supervisor, but a regular process that is monitoring the supervisor) detects this and can then apply a custom policy: for example, wait a certain amount of time (backoff), check external conditions (e.g. is a required service back online), then explicitly request the higher supervisor to start a new instance of the subtree [27]. This "manager/supervisor" pattern effectively grafts a smarter brain onto the supervision tree: simple supervisors handle quick restarts, but a higher-level manager can handle complex recovery like exponential backoff or coordinated restarts across subsystems [27] [28].

Translating these ideas to `lit-bit`: we can support hierarchical escalation by allowing a **SupervisorActor** to be supervised by another supervisor. Since `SupervisorActor` itself implements the `Actor` trait [29], one can spawn a supervisor under a higher-level supervisor. If the child supervisor decides not to restart further (e.g. returns None from `handle_child_failure` because max restarts exceeded) [30], it will remove the child and likely emit a `ChildPanicked` for itself (or simply terminate). The parent supervisor would then get a failure notification for that supervisor (as a child_id). At that point, the parent might have that child configured as *temporary* (meaning do nothing – effectively isolating the failure) or perhaps transient (meaning try to restart it if it was abnormal). In designing Phase 2, it would be wise to allow **policy overrides at each level**. The lower-level (child) supervisor already decides how to handle its children. The higher-level supervisor doesn't need to know those details, it only sees if the whole supervisor died. The higher level can have a separate policy for that event. For example, you could allow the top-level to specify "if the AuthService subtree crashes (its supervisor dies), restart it after 30 seconds" – a kind of manual backoff escalation. Achieving this might involve extending `SupervisorMessage` or the supervisor interface to carry *why* a supervisor died (e.g. did it exceed restart intensity?). Currently, `SupervisorMessage::ChildPanicked { id }` is used for any child failure [31], but we might extend it to include an error or reason code in the message. That way a parent supervisor's `on_child_failure` could behave differently if the failing child was itself a supervisor and perhaps pass along context. Even without that, a simple approach is: if a child is known to be a supervisor, the parent might choose a different default strategy (perhaps restart all siblings if an entire subtree went down, or mark it temporary to require external intervention).

Another aspect of hierarchical integration is **policy inheritance vs override**. In OTP, each supervisor in the tree has its own strategy and intensity settings – they don't automatically inherit those from their parent. This is good because different layers often have different expectations. For instance, a top-level supervisor might have a very lenient policy (never give up, always restart the major subsystems, since if those all fail the whole application might as well restart), whereas mid-level supervisors might be more strict about containing faults (escalating to top after a threshold). `lit-bit` should allow similar flexibility: each `SupervisorActor` can have its own `max_restarts` and window. There may be cases, however, where a parent wants to **override** the child's decision. Perhaps a child actor signals it wants to restart (via `RestartStrategy`) but the parent decides to kill it instead (maybe because the parent knows this type of error should bring down the whole subsystem). One way to support that is to let the parent's `on_child_failure` inspect the error or child metadata and choose to *escalate* further. For example, if `on_child_failure` returns no restart strategy (or a special strategy like "Escalate"), the framework could interpret that as an instruction to treat the parent itself as failing. In Phase 2, you could introduce an explicit **"Never" or "Escalate" strategy** to the enum or via a separate flag. This would make it clearer when a supervisor should not attempt a restart. (OTP achieves "never restart" by the temporary flag on the child spec [32], and escalate by the intensity mechanism [33] – both could have equivalents in lit-bit.)

In summary, supporting hierarchical escalation means designing supervisors that can supervise not just workers but other supervisors, and handling those cases thoughtfully. Patterns from OTP show that often the *combination* of policies at different levels yields resilience: e.g. a middle supervisor might **tolerate** a certain rate of child failure, but if exceeded, it dies; the top-level then waits (backoff or external signal) before reviving that subtree [25] [27]. Phase 2 can incorporate these ideas by allowing supervisors to be configured as permanent/transient/temporary even when the "child" is a supervisor, and by perhaps providing a hook or manager-like mechanism (even if it's just the user's own code using the provided APIs) to handle restarting of large subtrees with custom logic. Clear messaging between supervisors (e.g. a supervisor could send an explicit "I am shutting down due to X reason" message to its parent before dying) would also help coordinate the escalation. The end goal is a **predictable tree**: each node in the supervision hierarchy knows what to do when a child fails, and when in doubt, it escalates the problem upward. By documenting and exposing these patterns (policy per level, optional backoff manager, etc.), `lit-bit` can guide users to construct supervision trees that are both robust and tailored to the platform constraints.

5. Platform-Dual Implementation Strategies (Tokio vs. Embassy)

Designing the supervision system to work both on Tokio (std) and Embassy (no-std) requires leveraging each platform's mechanisms for task management and failure signaling. The Phase 1 implementation of `lit-bit` already laid the groundwork: in a Tokio environment, each actor runs in its own async task (with a `JoinHandle`), whereas in an Embassy environment, actors are typically spawned as static tasks without a join handle. The strategy for **Tokio** should center on using the `JoinHandle` to detect panics or errors in child tasks. Tokio's handle provides methods like `is_finished()` and when joined, yields a `Result` that indicates if the task panicked. In the current code, the supervisor calls `join_handle.now_or_never()` to poll completion non-blockingly [34]. If the join returns an error (`JoinError`), the code checks `join_error.is_panic()` to distinguish panics from cancellations. Right now, any panic (or cancellation) is mapped to an `ActorError::Panic` for the supervisor's purposes. Going forward, this could be extended to extract more info from the panic – for example, Tokio's `JoinError` doesn't directly give the panic message, but since the actor task itself could catch unwinds (as seen in the example usage of `panic::catch_unwind` [35] [10]), one approach is to have the actor future catch its own panic and return an `Err` variant that encodes something about it. In any case, Tokio allows *graceful handling* of panics at the task boundary: the supervisor task can ensure the runtime doesn't crash even if a child actor panics, and can use the join handle result to drive the supervision logic. The recommendation for Tokio is to continue leveraging this: spawn each actor with a result-returning future (e.g., `Result<(), ActorError>`), so that if the actor finishes with an error or panic, the supervisor can capture it. This may involve wrapping the actor's `on_event` loop in `catch_unwind` (only in std builds) to translate panics into an `ActorError::Panic` instead of an outright thread abort [36]. The code already demonstrates this pattern – the example wraps the actor's message handling in `catch_unwind` and converts a requested panic into a logged error, letting the task finish normally [37]. The supervisor then simply sees the task ended (likely with Ok), so in that particular example the restart decision was left to the actor (which chose to end itself). In a more general design, we might actually prefer the task to return a distinct error in such a case so that the supervisor *knows* the actor failed. Phase 2 can refine this by standardizing how an actor signals a controlled failure: e.g. perhaps have the actor return `Err(ActorError::Custom("..."))` when it catches a panic or hits an unrecoverable branch, so that the supervisor's join handle polling will treat it as a failure and invoke the restart policy.

On **Embassy (no-std)**, the challenges are different. In a no-std environment, panics typically result in an immediate abort (unless using unwinding with a custom panic handler, which is often not done on embedded). This means we cannot rely on a Tokio-like join mechanism to catch panics – if a panic occurs on embedded, it may reset the whole device. Therefore, the strategy in Embassy is to design actors to avoid panicking and instead use error reporting. Embassy's executor can spawn tasks, but those tasks are `'static` and usually run until completion without an external handle. One pattern is to use **message-based signaling** for actor termination. The `SupervisorActor` in `lit-bit` already defines `SupervisorMessage::ChildStopped` and `ChildPanicked` messages [38]. In an Embassy context, an actor can explicitly send a `ChildStopped` message to its supervisor when it completes work or is about to shut down. For example, if an Embassy actor decides to stop (perhaps it encountered an error it cannot handle), it could obtain a handle/address to its supervisor (passed in at creation) and send `SupervisorMessage::ChildPanicked` or `ChildStopped` before exiting its loop. Since Embassy does not support unwinding, “panic-aware” logic on embedded likely means designing all failure paths to go through Result. In practice, this might mean every actor’s `on_event` returns a `Result<(), ActorError>` instead of panicking, and if that Result is an Err, the actor can notify the supervisor and then terminate. The supervisor, upon receiving the message (delivered via the same mailbox mechanism), would then invoke the restart logic. This is in line with the **deterministic design**: no unpredictable panics, just error messages and coordinated restarts. It’s worth noting that in a no-std build, `lit-bit` uses a fixed-size list of children and tracks an `is_running` flag rather than join handles [11]. To detect when a child stops, the supervisor could periodically check if a child’s `is_running` flag has flipped to false. However, a more event-driven approach is to have the child’s *inbox loop* break out on termination and then send a message. For instance, if using Embassy’s `Spawner`, one could wrap the actor’s future like: `spawner.spawn(async move { actor_task().await; supervisor_addr.send(ChildPanicked{id}); });`. This way, when the actor future ends (either normally or due to an internal `Result::Err` causing it to break), a message is queued to inform the supervisor. This is the **message-based shutdown notification** strategy alluded to in the question. It avoids needing a join handle, using the actor’s own communication channel to propagate the event.

To implement this cleanly, Phase 2 might introduce an abstraction for child task handles that works for both platforms. For example, define a small trait or struct `ChildMonitor` that on std holds a `JoinHandle<Result<(), ActorError>>` and on no-std holds perhaps an `Option<Address<SupervisorMessage>>` to send notifications. But even without such abstraction, conditional code can handle each case. The **shared logic** across runtimes should be kept in the high-level policy (when to restart, how many times, etc.), while the **platform-specific bits** are confined to how we detect failure and how we spawn/restart. We see this approach in the code: e.g., `SupervisorActor.poll_children()` is compiled only for Tokio and uses join handles [39], whereas Embassy might not need an equivalent polling (it would react to messages instead). The restart mechanism (`execute_restarts`) can be mostly common, except that Tokio uses the stored factory to spawn a new task (via `tokio::spawn`) whereas Embassy’s factory likely directly returns an `Ok(())` after starting the task via the Embassy `Spawner` [40] [41]. By **abstracting the spawn/restart call** behind the `RestartFactory` (which is already a type alias that differs for Tokio vs not-Tokio) [42], `lit-bit` achieves a unified interface to “restart child”. Phase 2 can extend this to incorporate panic details: for example, the `RestartFactory` might be augmented to know how to obtain a fresh actor instance. Perhaps the user provides a closure that constructs a new actor (since cloning might not be possible). This closure can be stored in `ChildInfo` so that on restart, a brand new actor is spawned. (In the current code, `add_child_with_handle` uses a no-op factory by default [43], indicating that full restart capability is still being fleshed out.)

Finally, **testing** panic and escalation behavior across both platforms will be crucial. For Tokio (std), one can write unit tests that deliberately cause an actor to panic (or return an error) and assert that the supervisor sends the correct restart and that the actor is indeed restarted the expected number of times. The example `supervision_and_batching.rs` already demonstrates testing a “controlled failure” – they set a flag to simulate a failure and ensure the system handles it without crashing [37]. We should create similar tests where an actor panic is not caught internally, to ensure the supervisor’s join-handle logic catches it and triggers a `ChildPanicked` flow. For embedded/Embassy, testing is trickier since we can’t truly catch a panic. Instead, we simulate failures via error returns. We should test that an actor returning an error leads to a supervisor message and a restart. This can be done in an integration test configured for no-std (maybe using QEMU or a native no-std environment, or a special test harness where `panic!` is set to not actually abort). Additionally, tests for **escalation** can simulate many rapid failures to see if the supervisor stops restarting after `max_restarts` and then perhaps ensure a higher-level supervisor got the failure. Using a small `restart_window_ms` (e.g. 100ms) in tests can force the meltdown behavior quickly for verification. By covering these cases in both Tokio and Embassy configurations, we can be confident that the panic-aware supervision works deterministically and as designed on both platforms. In summary, the platform-dual strategy is: use **Tokio's robust task primitives** (join handles, unwinding) to fully capture panic info and error results in the supervisor, and use **Embassy's messaging and error-return conventions** to achieve similar supervision semantics in no-std (where panics are avoided in favor of controlled error paths). This ensures that `lit-bit`'s supervision remains consistent – an actor crash triggers the same conceptual supervisor behavior – regardless of which runtime is underpinning the execution.

Sources: Production actor frameworks and OTP concepts were referenced for best practices (OTP supervision strategies [1] [3], Ractor and Bastion restart policies [2] [17]). The design and code of `lit-bit` Phase 1 provided context on the current supervision implementation [9]. The *Adopting Erlang* guide on supervision trees was used to illustrate advanced escalation patterns like the temporary supervisor with an external manager [24] [27]. These informed the recommendations for trait hooks, restart backoff, and hierarchical strategies tailored to a dual-runtime Rust environment.

---

[1] [3] [4] [20] [32] [33] Erlang -- Supervisor Behaviour
https://www.erlang.org/docs/18/design_principles/sup_princ

[2] [6] [18] [19] ractor_supervisor - Rust
https://docs.rs/ractor-supervisor/latest/ractor_supervisor/

[5] [14] RestartStrategy in bastion::supervisor - Rust
https://docs.rs/bastion/latest/bastion/supervisor/struct.RestartStrategy.html

[7] [8] Supervisor in actix - Rust
https://docs.rs/actix/latest/actix/struct.Supervisor.html

[9] actor-overview.md
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/docs/actor-overview.md

[10] [35] [37] supervision_and_batching.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/examples/supervision_and_batching.rs

[11] [12] [13] [21] [29] [30] [34] [39] [40] [41] [42] [43] supervision.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/src/actor/supervision.rs

[15] [16] [17] [22] [23] Celebrate the 1st Bastioniversary with the 0.4 release! :tada: :confetti_ball: :rocket: - Bastion-rs blog
https://blog.bastion-rs.com/2020/07/21/celebrate-first-bastion-aniversary-with-release.html

[24] [25] [26] [27] [28] Supervision Trees | Adopting Erlang
https://adoptingerlang.org/docs/development/supervision_trees/

[31] [38] mod.rs
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/lit-bit-core/src/actor/mod.rs

[36] 2025-05-30.md
https://github.com/0xjcf/lit-bit/blob/bca14f6e715fddde190461bc6a191de5960da709/docs/PROGRESS/2025-05-30.md