ChatGPT

Designing a Minimal Rust Actor System: Lessons from XState, Akka, OTP, and More

Building an actor system for the `rust-statechart` library requires drawing from proven patterns in existing frameworks while leveraging Rust's unique strengths. In this report, we examine mature actor model implementations across languages – **XState (JavaScript/TypeScript)**, **Akka (Scala/Java)**, **Erlang/OTP**, **CAF (C++ Actor Framework)**, **Orleans (.NET)**, and **Pony (PonyLang)** – to inform a robust design. We highlight core architectural patterns and lifecycle models, common anti-patterns and failure modes, performance optimizations, API design for safety and ergonomics, and Rust-specific considerations (ownership, borrowing, and concurrency). Each section distills lessons and compares them to Rust's constraints and affordances.

1. XState Integration Patterns and Feature Comparison

XState's Actor-Based Statecharts: XState is a state machine/statechart library that explicitly embraces the actor model for integration and orchestration. In XState, *every running state machine is an actor* with its own mailbox and internal state [1] [2]. This means XState machines communicate exclusively via asynchronous events (messages) and never directly mutate each other's state [2]. XState encourages applications to be composed of many small, isolated statechart "services" (actors) that send events to each other, rather than one monolithic state store [3]. This approach aligns with the actor model philosophy of splitting a system into independent, concurrently running components.

Integration Patterns: XState excels at integrating statecharts into front-end and back-end workflows. In front-end apps (React, Vue, etc.), developers typically spawn a statechart actor per component or feature and use XState's React hooks (`useMachine`, etc.) to manage it. This makes state management **composable** and localized – e.g. a form component can run its own state machine actor, which can spawn child actors for sub-parts or async operations [4] [5]. Because each actor is self-contained, they can be started and stopped with the UI component's lifecycle, avoiding a single global store. In back-end scenarios, XState's actors (with the help of Stately's "Sky" platform) are used to manage long-running workflows like onboarding processes or game logic, where each workflow instance is an actor that can spawn others and persist state as needed [6]. A key integration pattern is *invoking external services as actors*: for example, an XState machine can **invoke** a promise, callback, or even another machine as an "invoked actor". XState v5 formalizes this by letting you define actors in the machine (using sources like `fromPromise`, `fromObservable`, `fromTransition`, etc.), so that any async task (API call, timeout, etc.) runs in its own actor context [7] [8]. This decouples side-effects from the core state logic – the parent state machine remains responsive, and the child actor sends back a completion event when the async task finishes [8]. In practice, this yields a **non-blocking integration** style: instead of one component waiting on an API call, it *spawns* an actor to perform the call and continues event-loop processing, handling the result via a message event.

Feature Set: Compared to typical actor frameworks, XState provides higher-level *statechart features* on top of the basic actor model. Notably, XState supports **hierarchical states**, **parallel states**, **history** and

guarded transitions, which help model complex logic formally. These features aren't standard in most actor systems (which usually have just "behaviors" or explicit code for state transitions). In XState, an actor can be in a well-defined state configuration and only accept certain events in certain states (enforcing invariants). XState also makes side-effects (actions) explicit parts of the statechart definition, improving clarity (though some caution that encouraging side-effects in the state machine can be misused if not carefully structured [9] [10]). XState actors can spawn child actors *hierarchically* (the parent will supervise the child's lifecycle to some extent – e.g., stopping child actors when the parent stops). They also have built-in support for **timers (delayed events)** and **global listeners**, which are typically external concerns in other actor systems. Moreover, XState includes developer tooling: a visual inspector and statechart visualization can connect to running actors, aiding debugging and design.

Comparison to Others: In essence, XState marries statecharts with actors. The actor model ensures each statechart instance runs independently and communicates via messages, much like Akka or Erlang processes, but the internal logic is a structured state machine rather than an ad-hoc message loop. This structure can prevent certain classes of bugs by making invalid states unrepresentable and transitions deterministic [11]. For example, XState is *type-safe* when used with TypeScript – events and context are typed, catching errors at compile time [12]. A Redux maintainer described XState as a "declarative Redux store with a guard at the door" – it won't allow arbitrary state changes, only defined transitions [13]. The trade-off is some runtime overhead for interpretive execution of the statechart, whereas a handwritten actor might be more lightweight. XState does not inherently support distribution or multi-threading (JavaScript actors run on a single thread/event loop). This means it can't achieve parallelism within one runtime instance, in contrast to Akka, OTP, or Rust which can distribute actors across threads or processes. However, XState's target domain (UI and workflow orchestration) typically involves moderate concurrency and prioritizes **integration and clarity over raw throughput**.

Pitfalls and Lessons: XState demonstrates how *integration and developer experience* can be prioritized in an actor system. Its actors are easy to start/stop along with app lifecycle, and multiple can be composed without central coordination, reflecting best practices for decoupling. One pitfall to watch for is the over-use of deeply nested statecharts or excessive spawning that isn't matched to the problem – like any actor system, creating too many actors can add overhead without benefits. XState's docs note spawning should be used when it makes sense to delegate work [5]. Another potential issue is managing *orphaned actors*: since XState allows spawning actors dynamically, it's important to stop them when no longer needed (the library provides mechanisms to auto-stop invoked actors when their parent state exits). For the Rust design, adopting XState's strengths means enabling **hierarchical actors (parent/child)**, **message-driven async invocation**, and perhaps even statechart-like helpers (e.g., an actor could optionally use a formal state machine internally for complex logic). The Rust actor layer should strive for XState's level of integration – e.g., allowing actors to plug into GUI frameworks or services easily – while taking advantage of Rust's concurrency (running truly in parallel and scaling up throughput beyond what a single JS thread can do). We also see from XState that **developer tooling** (visualizers, debug logs) greatly aids adoption; similarly, a Rust actor system can include logging of actor state transitions or message traces to help developers reason about system behavior.

2. Akka Architectural Lessons and Performance Insights

Actor Hierarchies and Lifecycle: Akka is a comprehensive actor toolkit on the JVM, known for its solid design for concurrency, fault tolerance, and distribution. One of Akka's core architectural patterns is the hierarchical *actor supervision tree*. When you create an actor in Akka, it automatically becomes a child of the

actor that created it (or of a guardian actor if created from the system) [14] [15]. This means every actor has a supervisor (except the top-level guardians), responsible for handling its failures. This pattern organizes actors such that failures are contained and managed: if a child actor throws an exception, the parent can decide to restart it, stop it, escalate the error, etc., according to a supervision strategy. **Lesson:** A Rust actor system should incorporate similar hierarchical supervision (as discussed in the OTP section below) so that actor lifecycles are managed and don't run unchecked. Supervision hierarchies also simplify clean shutdown – terminating the root supervisor cascades an orderly shutdown to the whole tree [16] [17].

Dispatchers and Concurrency Model: Akka decouples the *number of actors* from OS threads via dispatchers. By default, actors are executed on a thread pool (Akka uses a fork-join executor), and **each actor processes one message at a time** using an event-loop model [18]. This is similar to how Rust async tasks run on an executor: many tasks (actors) per thread. The key insight is that **threads are a resource pool, and actors are extremely lightweight** entities (essentially just their mailbox and behavior logic). This allows creating millions of actors if needed, without millions of threads. The Rust design should mirror this by using an async runtime or work-stealing threads to schedule many actors. Indeed, frameworks like Pony, Orleans, and CAF also emphasize scheduling millions of actors on a few threads [19]. One performance insight from Akka is that context-switching costs and mailbox contention can arise if not managed. Akka offers configuration to pin certain actors to dedicated threads or to use different dispatcher configurations (e.g., a single-threaded dispatcher for certain actors to ensure low contention, or a specialized I/O dispatcher). It also allows **mailbox tuning** – e.g., using bounded mailboxes to apply backpressure or priority mailboxes to reorder messages. Rust's `rust-statechart` actor layer can adopt these ideas by exposing options for bounded channel capacity (to avoid unbounded memory growth) and possibly different executor policies for different actors (e.g., CPU-bound actors vs. I/O-bound actors might use different thread pools).

Message Handling and FSMs: In Akka, actors traditionally handle messages via pattern matching in an endless receive loop. Akka *Classic* allowed any message type (essentially untyped, with runtime type checks), whereas Akka *Typed* (introduced later) requires defining a specific message protocol type for each actor. The typed approach improves type safety but also revealed a tension: actors often conceptually change behavior over time (e.g., different states accept different messages), which is harder to capture in one static type. As one Akka contributor noted, "the full set of messages an actor *can ever* receive is less interesting than the set it *accepts right now*" [20]. To address this, Akka's API includes patterns for finite state machines (the actor can `become` a new behavior, effectively changing its message handler set at runtime). For Rust, this suggests we should allow actors to change state/behavior in a type-safe way. One approach is modeling the actor's state as an enum and matching on it (similar to a state machine) or using different handler implementations for different states. The `rust-statechart` library could leverage its statechart capabilities here: an actor might integrate with a statechart that dictates which messages it handles in each state. This would give a structured way to handle evolving behavior, much like Akka's FSM trait or behaviors.

Performance Insights: Akka is designed for high throughput and has been used in demanding systems. Key performance techniques include: **non-blocking processing** – it's an anti-pattern in Akka to block an actor thread (e.g., doing a synchronous I/O or long computation) because that thread could be running hundreds of other actors. If an actor must do a blocking operation, Akka's docs advise using a dedicated dispatcher (thread pool) for it, isolating the impact. Similarly, in Rust an actor should offload blocking tasks to a separate thread or use asynchronous alternatives, to avoid stalling the async executor thread. Another insight: **message dispatch overhead** is small but non-zero, so excessively fine-grained actors may degrade performance. For example, spawning thousands of tiny actors for trivial tasks might add more overhead

(scheduling, GC, context switches) than doing the work in a single actor. As Manuel Bernhardt notes in *Akka anti-patterns: too many actors*, blindly subdividing work into more actors doesn't necessarily improve performance and often *introduces overhead* in message passing and coordination [21]. The implication for Rust is to find the right granularity: use actors to isolate truly concurrent or independent tasks, but don't create an actor per array element for a computation, for instance. The Rust implementation can mitigate overhead with efficient message passing (e.g., using lock-free queues or batching messages). Projects like Axiom/Maxim (Rust) observed that Akka's message stashing (for deferring messages while an actor is in a certain state) had performance and complexity issues, because stashing involves moving messages to a separate buffer and later requeueing [22]. Their alternative was to allow skipping messages in-place in the mailbox without extra data structures [22]. This kind of low-level optimization could be considered in Rust: e.g., a mailbox iterator that can skip certain messages based on actor state, rather than constantly reshuffling queues.

Scalability and Distribution: Akka's location transparency is a powerful concept: an `ActorRef` might represent a local actor or a remote actor (another JVM, another machine) without the sender knowing. This is enabled by Akka's networking module (Artery in newer versions) and Akka Cluster, which handle message serialization and addressing. Orleans similarly provides a network-transparent actor model (the *virtual actor*). A notable difference: Akka allows explicit actor placement and mobility is limited (once created at an address, an actor doesn't migrate at runtime) [23] [24], whereas Orleans will automatically recreate a deactivated or failed actor on a different server on-the-fly [23] [25]. For a minimal Rust actor system, full distributed transparency might be out of scope initially. However, designing the API with distribution in mind (e.g., serializable messages, distinct actor IDs) could make it easier to add clustering later.

Pitfalls and Anti-Patterns: The Akka community's experience highlights several cross-cutting pitfalls (many are applicable beyond Akka):

- **Shared Mutable State:** Sharing data between actors without message passing (e.g., via static mutable variables or concurrent data structures) breaks the actor model and can lead to race conditions. It's considered an anti-pattern in Akka [26]. The actor model is meant to encapsulate state; Rust enforces this by requiring `Send` for data crossing thread boundaries, so a Rust actor framework naturally encourages copying or moving data instead of sharing references. Nonetheless, it's important to document that actors should not, say, share a `Rc<RefCell<T>>` between them (that would be akin to shared memory and could lead to panics from RefCell or worse). The solution is always to send messages with the needed data, and if truly needed, use atomic or lock-protected structures in a controlled manner.

- **Blocking Calls in Actors:** As mentioned, doing blocking I/O or long CPU work inside an actor will freeze its processing and, in a pooled model, under-utilize other actors. In Akka, this is a well-known newbie mistake – the fix is to use `Future` (async) or delegate to a different dispatcher [27]. In Rust, this translates to using `tokio::spawn_blocking` for CPU-heavy tasks or ensuring `.await` is used for I/O rather than blocking calls. The actor system could integrate with Rust's async/await to make this natural (e.g., allow actor handlers to be `async fn` so they can await other futures without blocking the actor's thread).

- **Too Many Actors / Overhead:** As discussed, creating huge numbers of actors for no gain is counterproductive [21]. One should use other data structures or batch processing if millions of extremely fine-grained tasks must be handled per second. However, millions of actors *are* feasible if

they each have ongoing independent roles (Akka and Pony have shown they can handle millions of concurrent actors [19]). The Rust system should be benchmarked to find its practical actor limit and document guidelines for users (e.g., if you have millions of short-lived objects, consider using a work queue actor that manages them rather than one actor per tiny object).

- **Unbounded Mailboxes:** By default, Akka mailboxes can be unbounded (growing indefinitely). If message production exceeds consumption, this leads to memory issues. This is a general actor-system pitfall: lack of backpressure. Strategies to prevent it include bounded mailboxes (which will apply backpressure or drop messages when full) and message flow control protocols (acknowledgments, credit-based sending). The Axiom/Maxim principle (in Rust) explicitly advocates for bounded channels to encourage good design [28]. Orleans tackles backpressure at the application level by forcing async waits – an Orleans grain will not process more messages while awaiting a task result, implicitly slowing senders if they await responses. For `rust-statechart`, a simple start is to use bounded channels for actor mailboxes (or make it configurable). This way, if an actor is overwhelmed, senders could receive an error or be forced to wait, preventing runaway memory growth. As a bonus, bounded channels in Rust (e.g., using `tokio::mpsc` with a limit or using `flume` crate) can improve throughput by avoiding unlimited queue expansion.

- **Manual Thread Management:** A cross-language lesson is that developers should avoid managing threads by hand for actors – use the framework's scheduler. For instance, creating one OS thread per actor (naively) is a design error; frameworks like Akka and Orleans instead multiplex actors. In Rust, using async tasks means we naturally multiplex on a thread pool. If a user *thinks* they need one thread per actor for isolation, we should clarify that channels and async tasks provide isolation without that cost, except in special cases.

In summary, Akka teaches us to design with **hierarchy, supervision, and efficiency** in mind: a minimal Rust actor system can implement a similar parent-child model, use channels/executors to efficiently run many actors, and expose configuration for power users (but sensible defaults for most). It should discourage or prevent the known anti-patterns by construction – for example, by making message passing the only way to communicate (enforced by Rust's ownership rules) and by integrating with async so developers don't accidentally block the runtime. Performance techniques like work-stealing dispatchers, affinity to avoid excessive thread-hopping, and avoiding unnecessary copying (using zero-copy message passing where possible) can all contribute to matching or exceeding Akka's performance in Rust. In fact, without a GC and with zero-cost abstractions, Rust has the potential for lower latency per message. Benchmarks have shown that native actor frameworks (e.g., C++ and Pony) can outperform the JVM in message throughput for CPU-bound scenarios [29], so a well-optimized Rust actor system could similarly excel.

3. OTP Supervision Patterns Adaptable to Rust

Erlang's OTP (Open Telecom Platform) provides the **gold standard** for robust actor supervision and fault tolerance. OTP's model is often summarized as *"let it crash"* – meaning each process (actor) is expected to fail occasionally, and the system is designed to recover from failures automatically via supervisors. Key OTP concepts to adopt in Rust include **supervision trees**, **restart strategies**, and **error isolation**.

Supervision Trees: In OTP, all workers (actors that perform work) are intended to be under a supervisor process. A supervisor is a simple actor whose only job is to start child actors and restart them if they die

unexpectedly [30] [31]. Supervisors can also supervise other supervisors, forming a hierarchical tree of processes that mirrors the structure of the application. This ensures there are no unmanaged "orphan" processes – any process exists as part of a tree, so if it terminates, someone knows and can react [32]. The design also simplifies shutdown: when the application stops, OTP will sequentially terminate the supervision tree from the top, which cascades nicely (children terminate before parents, preventing resource leaks) [16] [17]. **Adaptation to Rust:** The `rust-statechart` actor runtime should incorporate a notion of a Supervisor actor. For example, when spawning a new actor, you might do so via a supervisor (perhaps the library provides a `Supervisor` struct that you register children with). If a child actor's task panics or returns an error, the supervisor logic can catch that and decide what to do (restart the actor, kill it permanently, etc.). This could be implemented by having each actor's run-future wrapped in a `catch_unwind` (to catch panics) or track a `Result` if the actor's own logic returns a `Result`. It's worth noting that Rust's strict ownership means that if an actor is restarted, you likely need to create a **new instance** of its state (since the old one could be in a bad state or partially moved when it failed). This is analogous to OTP, where a "restart" means a fresh process with initial state. We can provide hooks or traits to make actor state initialization repeatable (so that a supervisor can call something like `actor.restart()` to get a new instance).

Restart Strategies: OTP supervisors support several restart strategies to apply when a child crashes [33]:

- *One-for-one:* Only the crashed actor is restarted (this is the most common strategy) [34].
- *One-for-all:* If one actor crashes, **all** its sibling actors are terminated and then all are restarted. This is used when the actors are interdependent (if one's state is invalid, others might be too) [35].
- *Rest-for-one:* If an actor crashes, all actors that were started *after* it in the supervision list are terminated (the assumption is those "later" actors might depend on the earlier ones), then the crashed and those terminated are restarted.
- *Simple-one-for-one:* A simplified case for dynamically spawned children of the same type – all children are treated homogeneously (common for worker pools).

OTP also allows configuring *intensity* (max restarts in a time window before the supervisor itself gives up and crashes upward) [14]. **Adaptation:** In Rust, we can model similar policies. A `Supervisor` could hold a list of child actors (or tasks). If one fails, by default we do one-for-one (restart just that one). We might allow the developer to opt-in to one-for-all or rest-for-one semantics for a group of actors. For example, if two actors are tightly coupled (say one is maintaining an in-memory cache and another is a dependent updater), you might start them under a supervisor with `OneForAll` strategy so that if one dies, they both restart to avoid inconsistent state. Implementing one-for-all means the supervisor needs the ability to terminate other running actors; since they're under its control, it can send them a stop message or cancel their tasks. Rust's async `JoinHandle` can be cancelled by dropping it (if using certain executors) or by a cooperative cancellation token. Ensuring children can be force-stopped is important for correctness (OTP ultimately can kill a process if it doesn't shut down when asked [36]). We can provide a cancellation mechanism (maybe each actor's context checks a flag periodically or uses `select!` on a shutdown signal).

Isolation and "Let it Crash": OTP's philosophy is to write simpler code by *not* coding a lot of defensive checks – if something truly unexpected happens, just allow the process to crash (raise an exception). The supervisor will handle it by restarting a fresh instance, presumably restoring a correct initial state. This leads to resilient systems: memory leaks or bad states don't accumulate; they crash and reset. In Rust, a direct analogue is tricky: a Rust panic is a crash, but by default it would unwind the thread. If all actors share a thread pool, one actor's panic could potentially propagate and affect the runtime. We must therefore

contain panics within the failing actor. One way is to run each actor's event loop in a `std::panic::catch_unwind`. If an actor panics, the catch will trigger and we can notify a supervisor. The rest of the system continues running (since the thread itself didn't die, we caught the unwind). This is similar to Erlang's approach where a process crashing doesn't crash the whole VM. The downside is that not all Rust code is panic-safe – after a panic, we should assume that actor's state is poisoned and never use it again. So we indeed must discard that actor and make a new one. This is exactly what supervision is for. **Benefit:** Rust's strong typing reduces some need for "let it crash" (many bugs that might cause a null dereference or race in other languages are simply not possible or caught at compile-time in Rust). However, logic errors or external failures (I/O errors, etc.) still happen. Emulating OTP, we might encourage returning `Result` errors in actor handlers for expected error conditions (which could be sent as failure messages or handled), and reserve panics for truly unrecoverable situations. In either case, having a supervisor catch the error and restart means the overall system keeps running. For example, if an actor responsible for a statechart of a user session panics due to an unexpected event, the supervisor could restart it in a clean state or escalate if it keeps failing.

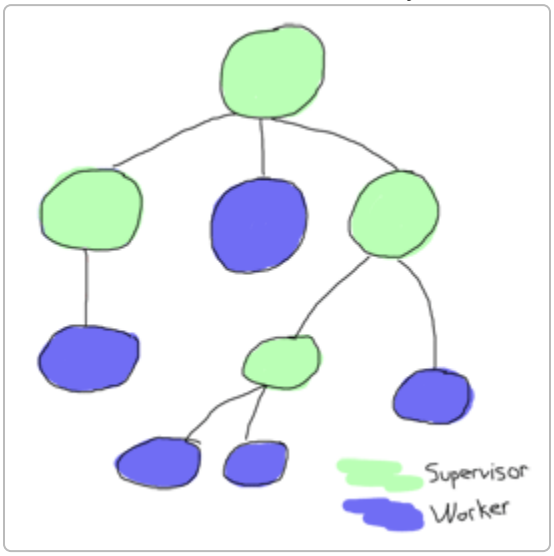
Monitoring and Linking: OTP provides *links* and *monitors* as primitives. A link is a bi-directional connection between two processes such that if one dies, the other gets an exit signal (and will die too unless it's trapping exits). Monitors are one-way signals to get notified if another process dies. These are used for building supervisors (which typically link to children and trap exits to be informed of their termination) [37]. In a Rust context, we can implement a similar mechanism where each actor is given a list of "on_exit" notifiers (its supervisor is one). If an actor finishes (normally or by panic), it should inform the supervisor through a channel or callback. This is straightforward with futures: a parent could `.await` the child's `JoinHandle`, but better is to not block the parent. Instead, use something like `tokio::spawn` each child, and have another task (the supervisor) that receives notifications via an mpsc channel when children complete. This event-driven supervision fits well with Rust futures. In fact, some Rust actor frameworks (like **Bastion** and **ractor**) have implemented full supervision trees with similar concepts. Bastion, for instance, by default runs all actors under a root supervisor with a *one-for-one* strategy, automatically restarting failed tasks [38] [39]. This demonstrates it's feasible in Rust. A minimal design could start with always using one-for-one (simplest) and later extend to other strategies.

Hot Code Upgrades and Evolution: OTP famously allows upgrading code of a running system without stopping it. This involves sophisticated features (two versions of a module in memory, handover of state). This is likely *out of scope* for our Rust actor system, as Rust does not support dynamic code reloading natively. Instead, upgrading will mean deploying a new version of the application. That said, designing actors to separate behavior and data (maybe via behavior traits) could allow swapping implementations if needed in a manual way.

Adaptation Challenges: Rust's ownership model means we must carefully handle references across restarts. For instance, if an actor holds an `Arc<Something>` that other actors also hold, a restart will create a new actor with a new Arc – the others still have the old one. In OTP, if processes share state via ETS tables or external resources, a restart doesn't automatically clean those up either. We should thus strongly encourage minimal sharing. Where sharing is needed (e.g., a configuration singleton), using something like an `Arc` is fine, and that won't be restarted on crash – only the actor's own state will. We also need to avoid **cycle leaks**: if two actors hold `Arc` references to each other, they will never drop, thus not truly "crashing" and freeing resources. OTP avoids this by not sharing memory at all between processes; Rust should do similarly. Use weak references if actors need to reference each other, or a registry that doesn't create strong cycles.

In conclusion, OTP's supervision patterns are highly applicable to Rust. We can implement **let-it-crash semantics with supervised restart** to achieve resilience on par with Erlang. The supervisor concept also meshes with statecharts: for example, a statechart could be seen as a supervisor of state "instances" or sub-actors. An interesting idea: the top-level statechart in `rust-statechart` (if any) could automatically act as a supervisor for any invoked child state machines (actors). If an invoked actor (state machine) terminates unexpectedly, the parent statechart can decide to spawn a replacement or handle the error. This would imitate OTP's idea of supervisors overseeing workers. By blending these concepts, we enable high reliability. As the Orleans paper noted, Orleans chose to eliminate explicit supervision trees by making the runtime handle failures (a grain call error propagates as an exception to the caller, and a new activation will be created later) [40] [41]. That approach simplifies developer burden at the cost of flexibility. In our Rust design, we might aim for something in between: provide a default supervisor that "just restarts" actors (so novices don't have to think about it), but also allow custom supervision strategies for advanced use cases.

(Embedded image: Example of a simple supervision tree where green nodes are supervisors and blue are worker actors. Each supervisor monitors its children and can restart them on failure.)



4. Cross-Language Anti-Patterns and Prevention Strategies

Across actor model implementations, many **common anti-patterns** have been identified. Recognizing these pitfalls and building prevention or mitigation into our design will help `rust-statechart` avoid known failure modes. Below we enumerate major anti-patterns observed in Akka, Erlang, Orleans, and others, along with strategies to prevent them:

- **Shared State and Race Conditions:** As mentioned, sharing mutable state between actors defeats the purpose of the actor model. It's a known Akka anti-pattern ("shared mutable state" [26]) and can happen in any language if actors bypass message passing. For example, two actors holding a reference to the same data structure could concurrently modify it, causing data races (in C++ or Rust) or consistency bugs (in GC languages). *Prevention:* Rust's type system gives us a big advantage – anything sent to another thread must implement `Send`, and mutation requires `&mut` which one thread at a time holds. By designing the API such that the only way to share data is to wrap it in thread-safe containers (`Arc<Mutex>` or similar), we make sharing an explicit and somewhat

onerous choice. We should encourage an immutable message pattern: all data exchanged is owned by the receiver (moved or copied). If truly shared read-only data is needed (e.g., a large lookup table), using an `Arc` is safe. The key is documentation and perhaps linting: warn users that sharing an `Arc<Mutex>` between actors could lead to classical multi-threaded complexity. Where possible, prefer *message passing of values*. In Akka, an equivalent guideline is to make messages immutable to avoid accidental sharing [27]. We can enforce immutability by requiring that message types implement `Clone` or are owned values (Rust ensures no aliasing unless you explicitly use interior mutability).

- **Global Singleton Actors and God Objects:** A subtle anti-pattern is having a single actor that does too much (a "god" actor or placing everything in one actor system when it could be separate). For instance, using one actor system per module (in Akka or .NET) can lead to too many actor systems that don't coordinate [42]. Or using one actor to handle all sorts of unrelated messages makes it complex and prone to errors. The actor model thrives on decomposition; when that's not used, you lose benefits. *Prevention:* Provide guidance to use multiple actors for independent concerns. Avoid designs that require a single mega-actor or a proliferation of separate actor systems. In Rust, typically one `System` (runtime) is enough; we can allow multiple, but we should warn that spinning up many independent thread pools is expensive (each has threads and overhead) [43]. Instead, encourage grouping actors under one system and using hierarchy for isolation. Our library could perhaps even prevent creating multiple systems unless explicitly needed.

- **Too Fine-Grained Actors:** On the flip side, creating an actor for every tiny piece of work is an anti-pattern noted in Akka ("too many actors") – it adds scheduling and messaging overhead without performance gain [21]. For example, spawning an actor for each element in a large array to compute something might be far slower than processing the array in one actor or using data parallelism. *Prevention:* This is more about educating users and providing alternative patterns. Rust has Rayon for data parallelism and async for concurrency; not everything needs to be an actor. When documenting our actor library, we can include guidelines: use actors to encapsulate long-lived, independent components (services, resources, stateful things), not for fine-grained data processing. If needed, we might integrate with `rayon` or similar for number-crunching tasks within an actor rather than making many actors. Also, if an actor needs to perform the same action N times, it could spawn a fixed pool of worker actors (e.g., 8 workers for 8 cores) instead of N actors, to limit overhead. We can even provide a utility (like "spawn_pool" as Maxim added [44]) to facilitate that pattern.

- **Unbounded Mailboxes / Backpressure Problems:** As discussed, letting mailboxes grow without bound is dangerous. It's a cross-language issue: in early actor frameworks, if a fast producer sends messages to a slower consumer, memory usage grows. Without backpressure, this can crash systems. *Prevention:* We plan to default to **bounded channels** for actor mailboxes (with a reasonable size default). This way, if an actor can't keep up, the sender's `send` can either block (if using sync channels) or yield a `Full` error (if using async channels). The user can then decide to drop messages or retry. This injects natural backpressure. Another approach from Akka Streams (Reactive Streams) is to have a demand-driven flow, but implementing a full streaming API might be beyond scope. Bounded mailboxes coupled with documented patterns (like request-response with bounded concurrency, e.g., only allow X outstanding requests from one actor to another at a time) will cover most cases. Notably, the **Axiom** actor framework explicitly used bounded channels, arguing it "results in greater simplicity and an emphasis on good actor design" [28] – because it forces the developer to consider capacity and apply flow control, rather than ignoring the issue.

- **Improper Use of Stashing/Buffering:** Some actor libs allow temporarily deferring messages (Akka's `stash` mechanism) for handling later. While useful, it can be misused – e.g., stashing too many messages can lead to memory issues, or logic bugs if not unstashed correctly. As the Axiom developers pointed out, Akka's stash has "many inherent flaws" and performance costs [22], since it moves messages to a separate list. *Prevention:* In a Rust statechart-based actor, we have an opportunity to handle this more elegantly. Since a statechart *knows* what events are deferred in a given state (those without transitions), the runtime could implicitly queue them without manual stashing. Or if we implement a stash, we should do so without copying messages around unnecessarily. Perhaps provide an API like `ctx.defer(event)` to mark an event to revisit later, but under the hood keep it in the mailbox. The exact mechanism aside, the key is to caution users to use deferral sparingly and avoid infinite deferral loops. If an event is always deferred and never handled, that's a logic error we could potentially detect (like XState will warn if events aren't handled).

- **Blocking/Long Tasks in Actor (again):** This is worth reiterating as it's very common. Even in Orleans (which uses async heavily), if a grain method does a long compute synchronously, it will starve the single-threaded executor for that silo, reducing overall throughput. Orleans users are advised to keep grain calls short and non-blocking. In Rust, one rogue actor that never `.await` can hog a CPU thread. *Prevention:* Use of Rust's async/await naturally encourages breaking up tasks, but one could still do a compute-heavy loop without yielding. We might provide a helper to easily offload computations (like an `spawn_blocking` wrapper in the actor context). Another idea: cooperative scheduling. Perhaps the actor mailbox processing loop yields to the executor after X messages or Y milliseconds, to give others a chance – though Tokio's scheduler might handle this anyway by not letting one task run forever without rescheduling. We should test scenarios and possibly advise splitting heavy loops or inserting `tokio::task::yield_now()` in long loops.

- **Ignoring Failure Signals:** In distributed systems (Akka cluster, Orleans), an anti-pattern is to ignore signals of failure – e.g., not handling a `Terminated` message in Akka (sent when a watched actor dies), or in Orleans ignoring a rejected Task. This leads to "zombie" references or missing opportunities to restart. *Prevention:* Our design can make monitoring automatic for parent-child (so that a parent gets a notification when child dies). Then at least one place in user code has the info. We can also design the actor `Handle` such that sending a message returns a `Result` or some confirmation, rather than "fire-and-forget" always – this way, if the actor has already died or mailbox is full, the sender is made aware and can act (maybe spawn a new actor, or report an error). In Akka, sending to a dead actor goes to "Dead Letters" which is just logged; in a stricter system, we might choose to surface that as an error to the sender. That might impose backpressure on the design, but it increases correctness.

- **Versioning and Interface Mismatch:** Over time, message schemas can change. In systems like Akka (if not using typed), it could lead to runtime `ClassCastException` if a message type isn't what actor expects. In Orleans, which uses strongly-typed interfaces, versioning is handled via interface versioning or backward-compatible changes. *Prevention:* Using Rust's enums or structured types for messages gives compile-time safety within one version of the program. If the system is distributed and needs upgrade, that's a larger problem (likely handled outside the library's scope). But encouraging **clear message schemas** (possibly using Serde for any serializable messages) is wise.

- **Tooling Neglect:** Not exactly an anti-pattern in code, but if developers have no insight into actor internals (mailbox sizes, actor lifetimes, etc.), issues can go unnoticed until production. Akka provides logging of actor events, and tools like Akka Monitoring, whereas early Orleans lacked some of this and added it later. *Mitigation:* We should include at least basic instrumentation hooks – e.g., the ability to log on actor start/stop, and perhaps query how many messages are in a mailbox (for debugging). Even a simple debug log when an actor panics or is restarted by a supervisor is extremely valuable to developers. Since Rust has a vibrant ecosystem for metrics (Prometheus, etc.), we could expose counters like "messages processed per actor" or "actor alive count" if not too costly.

In summary, many of these anti-patterns are avoided by following the **core actor principles** (no shared state, non-blocking, supervision, etc.). Our goal should be to make the *correct patterns the path of least resistance*. For example, by **design** in Rust: - It's easier to send an owned message than to attempt shared mutable state (due to ownership rules). - It's easier to use async IO than to use blocking calls (because async integrates, whereas using a blocking call would require explicit thread spawn). - It's straightforward to use the provided supervisor rather than roll your own brittle error handling.

Where possible, we incorporate these best practices into the API. For instance, the Maxim framework put heavy emphasis on ergonomics and simplicity, aiming to keep the core small and composable [45] [46] . This indicates that a minimal actor system should *not* try to do everything (persistence, cluster, etc. can be add-ons), but should focus on solid messaging and lifecycle. The variety of ~47 Rust actor libraries observed [47] shows that subtle trade-offs (performance vs. ergonomics, dynamic vs. static typing, etc.) have led to many re-implementations. Our job is to learn from those attempts and choose a balanced design to avoid constantly re-solving these problems. As one commenter noted, "Actor frameworks have a towering number of tradeoffs… Rust gives you all the control to hit any spot in the design space, so naturally people are looking all over the map" [48] . We should clearly define our goals (e.g., *safety*, *integration with statecharts*, *scalability*) and make trade-offs accordingly, while documenting them to users. By preemptively addressing common pitfalls with our design choices, we can guide users towards reliable patterns and reduce the chance they'll hit the same pitfalls seen in other ecosystems.

5. Performance Optimization Techniques and Benchmarking Approaches

Designing for performance and scalability is crucial to match or exceed systems like Akka and Orleans. We draw on optimization techniques from mature frameworks and outline how to benchmark effectively:

Efficient Message Passing: The core of actor performance is how fast messages can be sent, enqueued, and processed. In high-performance actor frameworks (CAF, Pony, etc.), a lot of effort goes into minimizing this overhead. For example, CAF (C++ Actor Framework) uses lock-free queues and highly optimized scheduler algorithms [49] . Pony goes further by leveraging its type system to eliminate data-race checks at runtime, allowing message sends without locks and a fully concurrent GC that never stops the world [50] . In Rust, we have excellent low-level primitives: using a lock-free MPSC queue (like crossbeam's) for mailboxes could give us high throughput. We should consider **batching** messages: rather than waking up an actor for each single message, the runtime could pull a batch of N messages if available and let the actor process them in one go. This amortizes the cost of switching context. Akka's dispatcher does something similar under the hood – an actor's mailboxes might be processed in batches to reduce scheduler thrash. We have to be careful though: processing too long in one actor can starve others (hence perhaps yield after a batch).

Work Stealing and Scheduling: Modern actor schedulers use work-stealing thread pools to balance load. Pony's runtime and CAF's default scheduler both employ work-stealing across threads to ensure idle threads grab work from busy ones [51] [52]. This maximizes CPU utilization. We can rely on Tokio's work-stealing scheduler for async tasks – it's quite advanced. However, actor frameworks often implement their own scheduling to incorporate domain knowledge (e.g., not all tasks are equal; maybe keep actors that communicate frequently on the same thread for cache locality). A research paper on locality-aware scheduling [53] shows that placing interacting actors on the same NUMA node can greatly improve performance. In Rust, if our actors are pure tasks on Tokio, we don't immediately control placement. But we could allow pinning an actor to a specific runtime or thread (Tokio allows creating per-thread runtimes). A possible feature: designate certain actor groups as requiring low-latency interaction, and schedule them on the same OS thread or within the same task set. This might be an advanced feature, but it's something to keep in mind as systems scale to multi-socket machines.

Memory Management and GC Pressure: One reason frameworks like Orleans and Akka can suffer is garbage collection pressure when there are millions of messages and actors. Rust's advantage is having no GC – allocations are explicit, and memory can be reused or freed promptly when out of scope. That said, if we naïvely allocate a new `Box` for every single message, we could overwhelm the allocator. We should explore pooling for messages or reusable buffers. For instance, if we frequently send the same type of message, using an object pool (or slab) might reduce malloc/free overhead. In Rust, this could be opt-in (the user can choose to reuse objects). Also, encouraging the use of small, copyable value types for messages (where appropriate) can avoid heap allocation altogether (they'd live on the sender's stack and then be moved to the receiver's stack). Pony's messaging is very efficient in part because of its allocator design and the absence of locks for transferring ownership. In our case, using Arc for large payloads can at least avoid copying that data multiple times; it will increment a counter instead, which is cheaper than a full clone.

Zero-Copy Message Passing: If an actor needs to send a large chunk of data to another in the same process, copying it completely will be costly. Languages like Erlang actually copy data between processes (because of isolation), whereas some actor frameworks allow zero-copy by transferring ownership. Rust can achieve zero-copy sends: for instance, a message could be a `Arc<Vec<u8>>`; moving the Arc is constant time and both actors can access the buffer (provided it's not modified). We must ensure the buffer isn't mutated concurrently (Arc<[u8]> or Arc<Mutex<Vec<u8>> if truly needed). In benchmark terms, this could significantly improve throughput for large messages. The Orleans paper measured the overhead of copying for isolation and found it manageable (4-10% overhead even for complex data structures) [54], but that's in .NET with copying. We can likely do better with direct moves.

Actor Lifecycle Costs: Another angle is how expensive it is to create or destroy actors. If an actor is very short-lived (e.g., in some request-per-actor models), the overhead of starting it might dominate. Akka and Orleans usually favor reusing actors or keeping them around (Orleans' virtual actors are activated on demand but kept alive for some idle time). In Rust, creating a new task (actor) has overhead: allocating a task struct, etc. We may want to use *actor pools* for certain workloads – e.g., start a pool of N actors waiting for jobs rather than constantly spawning. Alternatively, ensure that spawning an actor (task) is as cheap as possible by using a lightweight runtime. The `async` approach is beneficial: since our actors are essentially future state machines, spawning one doesn't require an OS thread or heavy context, just a new task in the runtime. Some benchmarks (Savina suite [29]) specifically measure actor creation throughput. We should aim to be competitive: Pony and CAF can create over a million actors per second in some cases [55]. We don't necessarily need that extreme, but we should avoid design decisions that add per-actor overhead (like enormous structs or lots of Arc cloning for each actor).

Throughput vs. Latency Trade-offs: Often optimizations that increase throughput might add a bit of latency (e.g., batching messages improves throughput but each message might wait a tiny bit longer in queue). We should decide on a policy or make it tunable. For a statechart in a UI, low latency (immediate reaction to events) might be more important than raw throughput. But for a backend processing pipeline, throughput could be king. Perhaps the actor system can offer modes or the user can choose to enable features like message batching. Benchmarking different scenarios will guide these choices.

Benchmarking Approaches: To validate performance, we should use **microbenchmarks** and **macrobenchmarks**:

- *Microbenchmarks:* These include things like "ping-pong" (two actors send a message back and forth N times – measures message latency and throughput), "pipeline" (message going through a chain of actors), "broadcast" (one actor sending to many), and creation/destruction tests. The Savina benchmark suite [29] provides a rich set of such microbenchmarks used in academia to compare actor platforms. We can port some of those to Rust to see how we stack up against published numbers for Akka, CAF, and Pony. For instance, *Chameneos* or *ThreadRing* benchmarks from Savina are classic – they stress scheduling and context switching.

- *Macrobenchmarks:* These are more scenario-driven – e.g., a web chat simulation with many users (actors) joining/leaving and sending messages, or an IoT sensor network scenario. Orleans was benchmarked using a "Halo 4 presence service" simulation (lots of players heartbeating) – they achieved ~5200 ops/sec per server and linear scaling across 125 servers [56]. While Rust single-node might handle far more (especially if simpler logic), it's worth creating a sample application (maybe a multiplayer game lobby manager, or a simplified trading system) to measure end-to-end performance (throughput and latency under load).

Real-world Performance Metrics: We aim to equal or surpass XState's performance in its domain and compete with Akka/Orleans in theirs: - In a UI context, XState's performance is more than adequate (updating state machines in response to clicks, etc. is not heavy). Rust's actor should easily handle that with negligible CPU overhead, given Rust's speed. The focus here is responsiveness – ensure no unnecessary delays in delivering events to statecharts. - For backend, Akka can typically process on the order of a few million messages per second on a decent server (with dozens of threads) for small messages, based on anecdotal reports and Akka team's benchmarks. Rust's zero-cost abstractions mean we could potentially push even higher. Pony's creators have shown >10 million msg/sec in some cases on similar hardware [55]. We should measure our implementation's max throughput and identify bottlenecks (like lock contention or allocator limits). - Orleans emphasizes scalability: handling millions of actors and scaling horizontally. In Rust, even if we don't implement distribution immediately, we can test how the system behaves with, say, 5 million actors on one machine (if memory allows) and moderate message rates. A user on Lightbend's forum reported slowing beyond 5 million actors in Akka cluster sharding [57] – memory and coordination overhead kick in. Rust might handle more per node if each actor is truly lightweight (a few hundred bytes each).

Optimizations Inspired by Others: - **NUMA-awareness:** As mentioned, on multi-socket machines, cross-node communication is slower. The locality-aware scheduling research [58] integrated with CAF suggests big gains if we schedule interacting actors on the same CPU node. We could consider grouping actors by some "affinity group" to hint the scheduler. This is advanced and might require custom executor logic beyond Tokio (or using tokio's newer features to pin tasks to a `LocalSet` tied to a thread). - **Cache optimization:**

Align frequently accessed data to cache lines, avoid false sharing. For instance, if the actor's mailbox and its state are on the same cache line, a different thread writing to the mailbox could invalidate the cache line for the thread running the actor. We might ensure the mailbox (queue) is separate or padded. These low-level details can be refined with profiling. - **At-most-once vs. at-least-once tradeoff:** Most actor frameworks (Akka, CAF, Pony) default to at-most-once delivery (no built-in retries). Orleans as well (calls are essentially RPCs that might fail). At-most-once is simpler and higher performance (no overhead to track acknowledgments or deduplicate). We will likely choose at-most-once too, and if reliability is needed, let the application implement retries or use a higher-level pattern (like ask-with-timeout and retry on no response). - **Batch sending:** If one actor needs to send 100 messages, sending them one by one incurs overhead per send. A possible optimization (used internally in some systems) is to allow sending a *vector of messages* in one go. This could reduce synchronization overhead on the queue. We could expose an API like `ctx.send_all(iterable_of_messages)` to push a batch into the mailbox at once.

Benchmark Example Data: It's useful to mention some comparative data (with citation) to set expectations. A study comparing Akka, CAF, and Pony based on Savina found that each had strengths on different patterns, but generally Pony and CAF (native code) outperformed Akka (JVM) in many computational benchmarks, while Akka wasn't far behind on others [59]. Pony's advantage comes from no stop-the-world GC and its capability-based safety allowing more aggressive optimization. Rust shares the "no STW GC" advantage. In one concrete example, the Orleans team showed their system scaling linearly: 25 servers handled 130k ops/sec (heartbeats) and 125 servers ~650k ops/sec [56]. This demonstrates the importance of horizontal scaling, but also the per-node limit (~5k ops/sec per node) which likely was due to the complexity of each operation (maybe involving storage). For lower-level messaging, frameworks achieve far higher per-node rates. We should aim for at least **hundreds of thousands of messages per second per core** in microbenchmarks, to be competitive.

Continuous Benchmarking: Adopting a practice from some high-performance projects, we could set up continuous benchmarks (perhaps using Criterion in Rust for microbenchmarks) to detect regressions when we change the implementation. This will help ensure that as we add features (like maybe more safety checks or more complex scheduling), we don't unknowingly degrade performance below our targets.

In summary, by using **modern scheduling techniques, efficient Rust primitives, and careful memory management**, a Rust actor system can achieve excellent performance. The design should allow toggling certain optimizations (like channel types or message batching) to suit the use-case, rather than hardcoding one approach. We will validate using both synthetic benchmarks (e.g., Savina) and realistic scenarios. With these efforts, we anticipate matching or exceeding XState's capabilities (which is easily done for throughput given XState runs on one thread) and approaching or surpassing Akka/Orleans on single-node performance. If distribution is added later, Rust could also shine in network efficiency (thanks to libraries like `tokio` and not having to deal with GC when serializing lots of objects). The end goal is a system that scales from a single-core embedded device (where low overhead is key) to a multi-core server handling large workloads, simply by adjusting runtime parameters.

6. API Design Recommendations and Developer Experience Insights

The success of an actor library hinges not only on performance but also on how *ergonomic and safe* its API is for developers. Drawing from the experiences of XState, Akka, Orleans, and various Rust libraries, we outline recommendations for an API that promotes productivity and confidence:

Embrace Strong Typing (Carefully): One of Rust's advantages is its type system, so our API should use it to catch errors at compile time. XState, being in TypeScript, highlights how helpful type definitions are – XState machines can be made completely type-safe, so that sending an invalid event or accessing missing context properties is a compile error [12]. We should strive for similarly strong typing: for each actor, define what messages it can handle. This could be via an enum of messages or a trait per message type (like Actix does with `Message` trait and `Handler` implementations). However, we must also be mindful of usability. Akka's journey from untyped to typed API taught that overly rigid types can frustrate users if their actor's message set is supposed to change over time or if the actor acts like a router forwarding various messages [20] [60]. One solution is to allow sum types or envelopes that can carry different variants. For example, `enum MyActorMsg { Foo(FooMsg), Bar(BarMsg), ... }` can encode a closed set of variants. Another approach: have the actor type itself be generic over the message type, but then you cannot easily have heterogeneous collections of actors. Orleans uses interface-based typing: each grain implements an interface (methods correspond to messages) and the system generates proxies. We could consider a macro to generate an actor interface that looks like normal Rust trait methods returning a `Future`. This would give a pleasant syntax: e.g., `actor.handle_request(x).await` instead of sending a message and awaiting a reply future manually. There's precedent: the *Orleankka* project (on .NET) attempted something similar [61]. In Rust, something akin to RPC can be done with `async_trait` and behind-the-scenes message passing.

Asynchronous Handler Support: A major point raised by Rust developers (e.g., Bartosz Sypytkowski) is that some Rust actor libraries initially lacked seamless async in message handlers [62]. Actix, for instance, was built before `async/await` stabilized, so its handlers return immediate results. To perform async work, Actix introduced complex workarounds (returning `ResponseFuture` or using an attribute macro) [63]. This was confusing – there were "4 different response types" for futures, each for different situations [63]. Developer feedback was poor: it felt like "a knife with 4 different blades but no handle" [64]. **Recommendation:** Design the actor API from the ground up for async-await. Handlers should be able to be `async fn` that borrow the actor's state (`&mut self`). This is tricky due to Rust's rule that you can't hold `&mut self` across an await unless `self` is pinned and the future is self-referential. However, frameworks like **Tokio's actor example** (Alice Ryhl's work [65]) show patterns where you can manage this by splitting the state or using interior mutability carefully. Alternatively, one can require that handler functions take the actor state by value (which could be in an Arc<Mutex> to allow temporarily releasing the lock during await). We should experiment with an approach that feels natural. Possibly, we define an actor as a struct plus an impl block with async handlers, and behind scenes, the library uses a macro to generate a polling loop that calls those handlers. The key is that *the user shouldn't need to understand pin or obscure futures – it should just work like writing async code elsewhere*. By aligning our API with standard Rust async patterns, we make it easy to integrate with other libraries (e.g., calling an external HTTP client inside an actor handler is straightforward if the handler is async).

Developer Experience (DX) from XState and Orleans: XState is praised for making complex state logic more visual and understandable [13] [66]. While our focus is the actor layer, we can still incorporate some of that spirit: for instance, we could offer a way to visualize the actor hierarchy or statechart. Even logging each state transition or message in a structured way can help. Orleans shows the value of a "pit of success" design: it's opinionated so that the easy path (the "happy path") covers most needs without much ceremony [67]. For example, in Orleans you simply call `GrainFactory.getGrain<MyGrainInterface>(id)` and call methods – the system handles where that grain lives, starting it if needed, etc. This contrasts with Akka where you explicitly create actors, keep ActorRef, and send messages, dealing with Ask patterns for request/reply. Many find Orleans more ergonomic for building scalable systems quickly [67]. For our Rust

library, we might consider offering higher-level abstractions on top of raw messaging. One idea: *services* that look like calling an async function on a struct, which under the hood routes to an actor. This could be done with procedural macros: define a trait for your actor's interface, and the macro generates an actor implementation that wraps each method call in a message. This way, in business code you don't manually construct messages, you just call `actor.some_method(arg).await` and get a result. Meanwhile, the actor's actual logic is defined in normal trait impl (possibly as async fns, which the macro turns into message handlers + automatic response sending). This pattern would combine Orleans's interface idea with Rust's async features. It promotes type-safe request/response and hides the messaging ceremony for developers who don't need custom behavior.

Ergonomics and Boilerplate: One reason there are many Rust actor libraries is that each tries a different way to reduce boilerplate. Macros are often used (Actix has macros for defining messages and handlers, Bastion uses macros to configure actor hierarchies, etc.). We should be cautious with macros: they can simplify usage but also hide complexity and produce less transparent errors. A well-designed API might minimize macros to just what's necessary (maybe derive macros for message types or an attribute macro for actor impl blocks). If possible, using traits and generics might suffice. For example, the trait `Actor` could have an associated `Message` type and a method `fn handle(&mut self, msg: Message) -> HandlerResult` (which could be a future). Then we could implement that for any statechart or actor struct. Additional traits could allow multiple message types by trait specialization or by boxing messages as a common trait object. Actix solved it by each message type implementing a trait and the actor implementing `Handler<M>` for each M. It works but involves a lot of generic impls. Perhaps an enum of all messages for an actor is simpler to start with (the compiler can optimize the match and it's all in one place, at the cost of requiring one enum definition).

Safety and Avoiding Footguns: Rust prevents a lot of issues (no data races, etc.), but there are still ways a user could misuse an API: - Forgetting to `.await` a future (common in any async code – they would get a warning if they drop it; maybe not specific to actors). - Causing deadlocks by waiting synchronously for an async response (if an actor tries to `block_on` something on the same thread, it deadlocks – we should warn against blocking the thread). - Creating cycles of `Arc` as mentioned, which leak memory (we could provide a `Weak` suggestion: e.g., an actor registry could give out `Weak<Addr>` references to others to break cycles). - Misordering initialization: perhaps needing to start the runtime or call some function before spawning actors. We should make the API foolproof, e.g., automatically start the runtime on first spawn if not started, or provide a simple `System::run()` pattern like Actix's. - Handling errors: if an actor's handler returns a `Result` or Option, ensure that is logged or handled if not explicitly by user. We don't want silent failures. OTP's approach is to escalate if unhandled – we could by default log any `Err` from a handler and restart the actor (similar to a panic) unless the user catches it.

Testing and Tooling: Good actor APIs provide testing utilities. Akka has the TestKit where you can create a "test probe" actor that records messages sent to it, or simulate time. XState can be tested by stepping through transitions deterministically. We can assist testing by allowing deterministic execution (maybe provide a single-threaded executor mode for tests so you don't deal with race conditions in assertions), and by providing mock actor addresses. Perhaps an actor address could implement a trait that we can swap out with a "capturing" implementation in tests. Alternatively, we can allow spawning a special test actor that just sends received messages to a channel for the test to inspect. In Rust, because of the static types, you might even have to compile the test with the same message types, etc., which is good.

Documentation and Learning Curve: Developer experience also includes learning materials. Many actor concepts (supervision, backpressure, etc.) are not immediately familiar to all. We should document patterns clearly, possibly with comparisons: "This is how you do X (like a supervisor) in our library," etc. Also, providing recipes (for web servers, for background worker pools, etc.) will help adoption. Given that so many attempted actor libs exist, we can also try to unify the community by providing migration guides or rationale: why choose our approach over others.

Feedback from Rust Actor Library Attempts: The Reddit discussion [68] [48] highlights that because Rust gives fine control, many authors explored different points in the design space. This fragmentation means no single library became "Akka for Rust" yet. Common feedback: - Actix (pre-async) was very fast but had unsound internals (which got fixed) and a steeper learning curve due to its unique concepts. Its integration with Actix-web made it popular, but outside web it's less used now. - async-std had an `actors` example (akin to the tokio one), but not a full framework. - Others like Riker, xActor, etc., each had limitations or lack of maintenance. - The lack of a clear winner suggests *developer expectations weren't fully met by any*. We have an opportunity to combine the best ideas (like Actix's message macro convenience, Riker's simplicity, Bastion's supervision, etc.) into a cohesive design, guided by the specific needs of `rust-statechart`.

From Bartosz's critique: he notes with some disappointment that despite Rust's fit for actors, the existing libraries were lacking [47] [62]. He specifically laments the complexity of adding async later (as with Actix), and ends up observing many devs just use raw Tokio channels + tasks as a "poor cousin" of a proper actor model [69]. This indicates that the barrier to using an actor framework was high enough that people reverted to simpler primitives. *Our goal should be to make the actor API nearly as easy as using* `tokio::spawn` *and channels directly, while providing the extra benefits (structure, supervision, etc.) seamlessly.* If we succeed, developers won't feel the need to reinvent a mini-actor pattern; they'll use the library because it's just as straightforward and far more powerful.

API Design Checklist: Summarizing specific recommendations: - Provide a clear way to declare an actor (struct + impl or a macro that expands to that). - Support `async` message handlers natively, allowing `.await` inside. - Ensure handlers can use `&mut self` safely (perhaps by disallowing holding it across await, which we can achieve by requiring certain patterns – e.g., handle one message at a time fully). - Use strong typing for messages, possibly with an enum or trait per message. Avoid `Box<dyn Any>` unless absolutely necessary for extensibility (it sacrifices compile-time checks). - Implement request-response pattern (the "ask" in Akka) in an ergonomic way – e.g., a function call returning Future, or a `call` method that wraps sending a message and waiting for reply. - Integrate with statecharts: since this is for `rust-statechart`, consider that an actor might be running a statechart internally. Perhaps the actor trait could be implemented for any statechart by having a generic adapter that takes events as messages and feeds them into the statechart interpreter. - Offer convenient spawning and supervision: e.g., `Supervisor::start(|sup| {...})` where inside you spawn children easily, or an attribute on actor struct to auto-restart on panic. - Provide logging/tracing hooks: e.g., an extension trait or global callback when any message is sent or received (for debug builds or tests). - Write thorough documentation with examples for common patterns (supervisor usage, stopping actors, sending responses, etc.).

By focusing on developer experience, we aim to mirror the approach XState took – make the correct usage intuitive. Tim Deschryver's write-up on XState emphasizes how it improved his code organization and gave him confidence in managing UI states [13] [66]. We want Rust developers to feel similarly – that using our actor system brings clarity and safety to their concurrency, rather than being an arcane framework they struggle against. A good measure of success will be if developers coming from Akka or Erlang find the

concepts familiar and the Rust constraints beneficial, and if those new to actor models find the library accessible enough to learn and adopt with minimal friction.

7. Rust-Specific Adaptations and Language Trade-Off Analysis

Finally, we examine how Rust's language model (ownership, borrowing, no GC) influences the design of an actor system. Rust offers significant **benefits** over garbage-collected and dynamic languages in this context, but also introduces some **friction** that must be mitigated.

Ownership and Data Race Freedom – A Boon: The actor model's philosophy is that actors do not share mutable state and communicate by copying messages. Rust's ownership and thread-safety rules enforce exactly that: you cannot share a `!Send` object across threads, and you cannot mutate something from two threads at once without a Mutex. This means many concurrency bugs that might occur if someone mis-used an actor framework in Java (e.g., sharing a global mutable list between actors) are caught or made impossible in Rust. In effect, Rust's compiler guarantees one of the actor model's key assumptions (no shared state) at the language level. This alignment has been noted by experts – "how well the borrow-checker ownership matches the actor model" [70] – and indeed suggests that building an actor system in Rust can feel natural. For instance, our actor's internal state will be owned by its task and not accessible elsewhere unless explicitly passed. Messages transferred between actors must be owned data that implements `Send` (and `Sync` if multiple threads read it). By constraining messages to these traits, we ensure no data races in message passing, which is a significant safety win over C++ actor implementations where one could accidentally share pointers.

Performance without GC – A Double-Edged Sword: Not having a GC means no runtime pause and generally lower memory overhead per object (no metadata, no write barriers). A Rust actor can allocate memory only when needed and free immediately when done (no waiting for GC cycles). This can reduce latency and improve throughput, especially under load (no surprise GC pause when message throughput is high). For example, in high-scale scenarios, JVM garbage collection tuning for Akka can be challenging (to avoid long pauses). Rust avoids that by design. Pony, which also has no global GC (it does per-actor reference counting with cycle detection), showed that avoiding stop-the-world pauses yields excellent tail latency and throughput stability [50]. We expect Rust actors to have similarly stable performance.

However, the lack of GC also means **no automatic cycle collection**. If two actors hold references to each other (say each has an `Arc` of the other's address), and they stop, the reference cycle would prevent deallocation – a memory leak. In GC'd languages, cycles are detected and collected (unless they involve non-memory resources). In Rust, we need to break cycles manually (using `Weak` or a drop protocol). This is a point of friction: developers coming from GC might not anticipate it. Our library could help by providing patterns to avoid cycles – e.g., a parent actor can hold `Arc` to child, but child holds a `Weak` to parent, so when parent drops, child can detect that and terminate. Documenting these patterns is essential. In practice, not many actor relationships need bidirectional strong pointers – usually a parent supervises children (one-way link), or actors communicate by exchanging addresses as needed rather than permanently. We can also provide a "registry" that holds actor addresses by ID and returns `Weak` references, to allow lookups without strong cycles.

Borrowing and Lifetimes – Friction Points: Rust's borrowing rules can complicate actor implementation internally. One classic issue: if an actor's handler wants to hold a reference to its state while awaiting a

future, that's not allowed because the state (`&mut self`) is borrowed across an `.await` point (which is effectively a yield). This is why Actix could not simply allow `async fn` handlers with &mut self (until maybe recently with pinned self tricks). To mitigate this, one can use patterns like splitting the state: borrow what you need into a local variable before the await, then do the await, then reborrow. It's a bit manual. Some libraries opted to make the actor's state an `Arc<Mutex<_>>` so that the handler takes an `Arc` clone and can lock around await – but that sacrifices single-threaded efficiency and compile-time checks (it moves to runtime lock). We should avoid making users pepper their code with locks unnecessarily. Instead, our actor runtime can itself manage the task in a way that the borrow isn't held over await. For example, use a message handling loop where for each message, we call an async handler and `.await` it externally, not inside the actor's borrow. Something like:

```
while let Some(msg) = mailbox.recv().await {
    let fut = actor.handle(msg); // this takes &mut actor
    // actor is borrowed for the call, but handle returns an impl Future that
does *not* have &mut self live (maybe it captured some parts).
    // Actually, if handle is an async fn with &mut self, it translates to a
Future that *does* borrow self for its lifetime, unless we use tricks.
}
```

One trick: use `Pin` and unsafe to allow self-referential futures – probably not worth it. Another: require that handler returns a `Future` that doesn't borrow self (like Actix's `ResponseFuture` pattern, where you must drop any self-borrow before returning the future). Perhaps better: design the actor state such that long waits are done in helper functions outside the main state borrow. This is complex, but perhaps we can hide it from the user via a macro.

Single-threaded Actors and !Send types: Rust's `Send` requirement can be a limitation for some use-cases. In Akka, an actor is always on the JVM heap and can use any object; in Rust, if we want to run actors on multiple threads (and we do for scalability), their messages must be `Send`. What if someone wants to use a non-`Send` type in an actor (like a GUI handle that isn't thread-safe)? One solution is to pin that actor to a single thread (like spawn it on a `LocalSet` in Tokio). Some Rust actor libraries allowed non-Send actors by confining them to a thread (Actix had a concept of Arbiter thread; the unsafety that was criticized came from bypassing `Send`). We can support this safely by using `!Send` futures: Tokio's `LocalSet` can run tasks that aren't Send, meaning they will not hop threads. We'd need a separate API to spawn a "local actor" which carries the `!Send` bound. Those actors couldn't be supervised by a normal (Send) supervisor unless the supervisor is also local to that thread. Possibly we have a parallel hierarchy for thread-local actors (e.g., UI actors that must run on the main thread). This is a niche but important for certain integrations (like if someone wants to use this actor system in a desktop app with a single-threaded GUI loop).

Error Handling and Panics: As discussed, Rust panics are not like exceptions – if not caught, they unwind and can kill the whole program. In managed actor systems, an exception in an actor typically doesn't crash the process; it's caught by the framework (Akka's supervisor, Orleans surfaces it as a faulted Task). To provide equivalent resilience, we *must catch panics*. Rust allows catching unwinding panics with `catch_unwind`, but only if the thread's panic strategy is unwind (which is default in debug, could be abort in release – we'll likely want to keep unwind for the actor threads so we can recover). We should document that our library expects `panic = "unwind"` in Cargo.toml for reliability (abort would defeat supervision).

Once we catch a panic, we treat it like a failure signal. This approach will introduce a slight performance overhead (catch_unwind has some cost), but it's necessary for robustness. We could perhaps make it optional (for maximum performance, run without unwind catching, but then any panic crashes the app – suitable for certain deployments where reliability is achieved by process restarts rather than actor restarts).

Bartosz Sypytkowski points out that Rust panics will **crash the whole server** if not contained, meaning one panic can take down all services (whereas in .NET/Java, an exception in one request thread won't kill the server) [71] [72] . This is a *key difference* that our design must address. By implementing the supervision and panic catching, we essentially bring Rust's failure model closer to Erlang's (isolated failures). It's somewhat ironic: one sells Rust on "no crashes, all errors handled", yet here we are allowing actors to crash and recover – but that's intentional for a resilient system. We just have to ensure those crashes don't leak or corrupt state elsewhere. If we do catch_unwind, the actor's memory is safely dropped (because unwind runs destructors). If some data was in a bad state, dropping it might run its destructor – hopefully that doesn't panic too. If it does, it double-panics and aborts. That is rare but possible; best to keep actor destructors simple. We might advise users to implement `Drop` for actor state carefully or not at all, to avoid panicking there.

Memory Footprint: Without GC, each actor's memory usage is mostly what it explicitly allocates. Erlang processes have a pre-allocated heap that grows, and a small fixed overhead. Rust actors might allocate from the general heap as needed. The upside: potentially less memory overhead if the actor is idle (no unnecessary heap). The downside: fragmentation could occur for many actors; we might explore allocator tuning (maybe using `jemalloc` or mimalloc which handle fragmentation well). If we aim to host millions of actors, memory per actor must be low. Each actor might at least have a mailbox (queue) – if we use a global concurrent queue or one channel per actor (likely one channel per actor pair or per actor for inbound), that's a few pointers and maybe some buffer. Could be as low as a few dozen bytes overhead plus message data. We should attempt to minimize any per-actor static overhead (like large vtables or heavy structs). Using enums for messages (vs. trait objects) can eliminate some heap allocation for the message itself. In Akka, each actor is an object with multiple pointers (context, etc.), and the mailbox is a separate object; plus GC header overhead. We can likely beat that footprint.

Fighting the Borrow Checker: A non-trivial part of Rust development is occasionally restructuring code to satisfy borrow rules even if the logic is sound. This can be seen as "friction". For example, consider an actor that has a `HashMap` in its state and we want to iterate and modify it while also sending messages. In Rust, you might hit borrowing issues if you try to keep a mutable borrow on the map while calling a method that (for instance) sends another message and might need access to `self`. The solution is usually to copy out the needed pieces or split the structure. As Bartosz noted, sometimes you must "split your types or methods just because the borrow checker says so", even if it adds no real value beyond appeasing the rules [73] . This can make actor code more verbose. We can't change Rust's rules, but we can design our API to reduce such cases. For instance, if our actor context offers methods to send messages that take `&self` (not `&mut self`), then you can send a message while still reading your state. We can achieve that by not tying sending to the actor object itself. Maybe the handler gets two arguments: `&mut self` and a `ctx: ActorContext` (where `ctx` allows `ctx.send(other, msg)` by internally borrowing from some global or static component). This way, sending doesn't require `&mut self` of the actor, only `&self` or even no reference if context is independent. In Actix, they have `ctx.address()` and such that aren't part of the actor's mutability. We should mimic that: the actor's mutability is only needed for its internal state updates, not for messaging or scheduling operations. By separating those concerns, we give developers more flexibility to avoid borrow conflicts. For example, an actor could iterate through its internal map (`&mut`

`self` borrowed for iteration) and still send messages via the context (which doesn't need the actor's &mut). This kind of API separation can alleviate borrow checker struggles.

Comparing with GC/Dynamic Systems: Let's explicitly list trade-offs: - *Throughput:* Rust has potential for higher raw throughput (no GC pauses, compiled code optimizations). However, managed runtimes have highly optimized JITs and can sometimes come close, but usually at the cost of more CPU (due to GC work). Rust should excel in sustained throughput per core. - *Latency:* No unpredictable pauses is a plus. However, GC languages often allocate very fast (bump allocators). Rust allocation can be slightly slower per object, but we can mitigate via pooling. Also, a GC system might introduce latency spikes (bad for tail latency). Rust can be made almost real-time if needed by avoiding long critical sections and using lock-free. - *Memory usage:* GC requires overhead for metadata and often additional memory to allow efficient GC (headroom for copying collectors, etc.). Rust can operate in a tighter memory footprint if managed carefully. But if developers clone data unnecessarily to satisfy borrows, that could increase memory usage compared to an in-place mutation in a GC language. It's a matter of coding style. For instance, in Rust you might clone an input message to send to two actors, whereas in a GC language you'd send the same reference to both (with the understanding not to mutate it). That clone is an overhead, but we consider it a cost for safety (no accidental shared mutation). In many cases, these clones are small or can be optimized. - *Complexity:* Rust's model is more complex for newcomers. In a dynamic actor system (like Python's or even Erlang's), you don't think about types or ownership – you send any message and if it's wrong, you might only find out at runtime or via tests. Rust forces thinking in types up-front, which is good for reliability but can slow down prototyping. Also, the borrow checker may force some refactoring that a Python dev never worries about (though they might instead worry about threading issues that Rust eliminates). To mitigate this, our library's documentation and possibly compile-time errors should be as friendly as possible, guiding the user to solutions (like using .clone() or restructuring code). If we can make the compile errors point to our docs ("if you see this error, it might be because you awaited while holding a &mut self – consider these patterns..."), that would be great.

Rust Actor Libraries Mitigations: Over the past years, various libraries have tried to handle these challenges: - **Actix:** Originally bypassed some Send requirements by saying all actors run on the same thread by default (so you could use non-Send types). This was powerful but involved `unsafe` and was called out as unsound. Actix 0.8+ addressed it by requiring `Send` or carefully ensuring no cross-thread moves for those actors. The lesson: we should avoid using unsafe to circumvent the type system; instead, design within its constraints for soundness. - **Bastion:** Embraced the Erlang model fully – all actors are supervised, and it uses a *lightweight process* concept. It's async under the hood and uses its own scheduler for control. It doesn't force the user to deal with lifetimes because you usually define closures for actor behaviors that get 'static. Bastion likely uses `'static` heavily – actors and messages must be 'static since they can outlive the call site. We likely do the same (spawned tasks in Tokio require 'static). That can be a frustration: if a developer tries to capture a non-'static reference in an actor (like a reference to a configuration struct in main), the compiler will complain. The solution is to either make it static (by moving the config or using Arc) or use some hack. We should clearly document that actor state and messages should be 'static (meaning owned or global). It's a constraint that might confuse new Rust users, but it's fundamental for concurrency. - **Others (e.g., Riker, xactor):** These often limited flexibility (maybe only one message type per actor or requiring downcasting for dynamic messages). The trade-off of a single message enum per actor is simpler but less extensible. We can choose that for simplicity initially – it's better to have a simpler API that covers 90% of needs and do it well, than a very generic but complex API that tries to allow any type of message (leading to lots of Any or unsafe casting internally).

In conclusion, Rust's model gives us a powerful toolbox to implement a **fast, safe actor system**, but we must navigate its strictness. The design will involve making some compromises: - We will likely require that all actor state and messages are `'static + Send` (for simplicity and safety), which covers most use cases but not all (e.g., non-Send types bound to a thread). - We will implement robust panic handling to mimic actor isolation (prevent one failure from crashing all). - We will provide mechanisms to avoid common ownership traps (like clear guidelines to avoid Arc cycles, context methods that use &self to avoid needing &mut). - We accept a bit of additional verbosity for the sake of clarity and correctness – e.g., defining an enum for messages or using a macro – because Rust doesn't have reflection or dynamic typing to magically route messages. But this verbosity pays off in maintainability (the compiler ensures completeness: if you add a new message variant, you'll get a warning if you don't handle it in the actor's match, etc., which is great).

One could say the **trade-off** is: Rust will front-load more work (designing types, satisfying the borrow checker), but once it compiles, the actor system will likely run correctly with fewer surprises (no data races, fewer runtime type errors). The output of our research strongly suggests that leaning into Rust's strengths (type safety, ownership) is the right approach, and where those strengths create friction, we smooth it with good API abstractions. By learning from prior implementations and carefully designing the `rust-statechart` actor API and internals, we can create a minimal yet powerful actor system that **avoids known pitfalls, maximizes performance, and leverages Rust's guarantees** to give developers confidence in building concurrent statechart-driven applications.

Sources:

1. Stately AI – *XState Documentation: The Actor Model* [1] [2] [74]
2. Reddit (r/reactjs) – *Discussion of XState vs Redux, actor-model approach* [3]
3. Manuel Bernhardt – *Akka Anti-Patterns series* (shared mutable state, too many actors, etc.) [26] [21]
4. GitHub – *Maxim (Axiom fork) actor framework README* (design principles: skipping messages vs stashing, bounded channels) [75] [28]
5. Microsoft Research – *Orleans Technical Report* (comparisons with Erlang/Akka, automatic actor lifecycle) [14] [24]
6. Alice Ryhl – *Actors with Tokio – a lesson in ownership* (talk/keynote) [65]
7. Bartosz Sypytkowski – *"Is Rust a good fit for business apps?"* (critique of Rust actor libraries, async handling, panic behavior) [62] [71]
8. Reddit (r/rust) – *"There are a lot of actor framework projects" discussion* (trade-offs, design space) [48]
9. Sebastian Blessing (Pony developer) – *Savina Benchmark results* (Pony vs Akka vs CAF performance) [55]
10. Orleans vs Akka comparison (Hacker News comment) – *Orleans easier "happy path" vs Akka* [67]
11. Learn You Some Erlang – *Supervision concepts* (importance of supervision trees) [76] [16]
12. Orleans paper – *Halo benchmark throughput scaling* [56]

---

[1] [2] [5] [6] [18] [74] The Actor model | Stately
https://stately.ai/docs/actor-model

[3] [9] [10] My experience with using state machines (xstate) in production : r/reactjs
https://www.reddit.com/r/reactjs/comments/ilfi4c/my_experience_with_using_state_machines_xstate_in/

4 Patterns for state management with actors in React with XState
https://www.typeonce.dev/article/patterns-for-state-management-with-actors-in-react-with-xstate

7 8 State machines and Actors in XState v5 | Sandro Maglione
https://www.sandromaglione.com/articles/state-machines-and-actors-in-xstate-v5

11 12 13 66 Tim Deschryver
https://timdeschryver.dev/blog/my-love-letter-to-xstate-and-statecharts

14 15 23 24 25 37 40 41 54 56 microsoft.com
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Orleans-MSR-TR-2014-41.pdf

16 17 30 31 32 36 76 Who Supervises The Supervisors? | Learn You Some Erlang for Great Good!
https://learnyousomeerlang.com/supervisors

19 49 51 52 53 58 Work-Stealing, Locality-Aware Actor Scheduling | CoLab
https://colab.ws/articles/10.1109%2FIPDPS.2018.00058

20 60 61 67 Orleans is just fantastic. 343 Industries famously used Orleans for Halo 4 backe... | Hacker News
https://news.ycombinator.com/item?id=12109186

21 Akka anti-patterns: too many actors - Manuel Bernhardt
https://manuel.bernhardt.io/2018/08/06/akka-anti-patterns-many-actors/

22 28 44 45 46 75 GitHub - katharostech/maxim: An easy-to-use Rust actor framework/model. Forked from the Axiom actor framework.
https://github.com/katharostech/maxim

26 42 anti-pattern · Manuel Bernhardt
https://manuel.bernhardt.io/tags/anti-pattern/

27 The Top 7 Mistakes Newbies Make with Akka.NET - Petabridge
https://petabridge.com/blog/top-7-akkadotnet-stumbling-blocks/

29 55 Pony - Sebastian Blessing, Engineer & Programming Language Enthusiast
https://www.doc.ic.ac.uk/~scb12/pony.html

33 34 Elixir/OTP : Basics of Supervisors | by Arunmuthuram M | Medium
https://arunramgt.medium.com/elixir-otp-basics-of-supervisors-cc71bfd331c2

35 Supervision Trees - Adopting Erlang
https://adoptingerlang.org/docs/development/supervision_trees/

38 39 GitHub - bastion-rs/bastion: Highly-available Distributed Fault-tolerant Runtime
https://github.com/bastion-rs/bastion

43 Akka anti-patterns: too many actor systems - Manuel Bernhardt
https://manuel.bernhardt.io/2016/08/23/akka-anti-patterns-too-many-actor-systems/

47 62 63 64 69 70 71 72 73 Is Rust a good choice for business apps?
https://www.bartoszsypytkowski.com/is-rust-a-good-fit-for-business-apps/

48 68 There are a *lot* of actor framework projects on Cargo. : r/rust
https://www.reddit.com/r/rust/comments/n2cmvd/there_are_a_lot_of_actor_framework_projects_on/

50 Among these, Pony is the only language that concurrently garbage ...
https://news.ycombinator.com/item?id=9483071

57 Akka-Cluster: Decreasing system performance having many active ...
https://discuss.lightbend.com/t/akka-cluster-decreasing-system-performance-having-many-active-actors/7246

59 Pony, Actors, Causality, Types, and Garbage Collection - InfoQ
https://www.infoq.com/presentations/pony-types-garbage-collection/

65 Keynote | Actors with Tokio – a lesson in ownership - Alice Ryhl
https://www.youtube.com/watch?v=fTXuGRP1ee4