DOI: https://doi.org/10.36910/6775-2524-0560-2026-64-14
UDC 004.75:004.41
Dubynskyi Oleh, Postgraduate student
https://orcid.org/0009-0003-5486-5456
Porev Victor, PhD in Technical Sciences, Associate Professor
https://orcid.org/0000-0001-5250-1454
National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine

EVOLUTION OF ACTOR MODEL IMPLEMENTATIONS IN DISTRIBUTED SYSTEMS: A SYSTEMATIC DOCUMENTARY REVIEW AND MULTI-CRITERIA COMPARISON OF ERLANG, AKKA, ORLEANS AND PROTO.ACTOR

Dubynskyi O., Poriev V. Evolution of Actor Model Implementations in Distributed Systems: A Systematic Documentary Review and Multi-Criteria Comparison of Erlang, Akka, Orleans and Proto.Actor. The paper presents a systematic comparative analysis of four leading Actor-model framework implementations (Erlang/OTP, Akka, Microsoft Orleans, and Proto.Actor) along five architectural dimensions: actor lifecycle management, cluster topology and node discovery, message serialization and inter-node communication, state management and persistence strategies, and the concurrency model with associated mailbox implementations. The work is a documentary review with multi-criteria decision analysis (MCDA): each criterion is tied to a documented framework property, scored on a five-point ordinal scale, and summed under equal and scenario-specific weights. The paper shows that over fifty years of evolution the Actor model, proposed by C. Hewitt in 1973, has split into two architecturally opposite camps: classical actors with explicit lifecycle control (Erlang, Akka, Proto.Actor), and virtual actors with implicit, platform-managed lifecycle (Orleans, Dapr). The paper identifies three evolution points: the shift from monolithic node discovery to externalized coordination (Consul, etcd); the shift from native serialization to cross-platform formats based on gRPC/Protobuf; and the shift from manual persistence to declarative state management. A ten-criterion framework-selection matrix with a declarative decision tree is proposed and observability and security aspects are surveyed. The results can support the design of new distributed systems and the audit of existing ones. The scientific novelty lies in the systematization of the internal mechanisms of Actor-model frameworks in a comparative dimension absent from the literature so far.
Keywords: Software engineering, software architecture, actor model, distributed systems, virtual actors, Erlang, Akka, Orleans, Proto.Actor, serialization, persistence, clustering, mailbox, concurrency.

Дубинський О. І., Порєв В. М. Еволюція реалізації моделі Акторів у розподілених системах: систематичний документарний огляд та мультикритеріальне порівняння Erlang, Akka, Orleans та Proto.Actor. У статті здійснено систематизований порівняльний аналіз чотирьох провідних фреймворків реалізації моделі Акторів - Erlang/OTP, Akka, Microsoft Orleans та Proto.Actor - у п'яти ключових архітектурних вимірах: управління життєвим циклом актора, топологія кластера та механізми виявлення вузлів, серіалізація повідомлень і міжвузлова комунікація, стратегії керування станом і персистентності, а також модель конкурентності та організація поштових скриньок (mailboxes). Робота побудована як документарний огляд із мультикритеріальним аналізом рішень (MCDA): кожен критерій прив'язано до документованої властивості фреймворку, оцінено за п'ятибальною порядковою шкалою та підсумовано за рівними й сценарними вагами. Показано, що за п'ятдесят років еволюції модель Акторів, запропонована К. Хьюіттом у 1973 році, розділилася на два архітектурно протилежні табори: класичні актори з явним керуванням життєвим циклом (Erlang, Akka, Proto.Actor) та віртуальні актори з неявним, платформо-керованим життєвим циклом (Orleans, Dapr). Виявлено три ключові точки еволюції: перехід від монолітного виявлення вузлів до externalized coordination (Consul, etcd); від нативної серіалізації до крос-платформових форматів на базі gRPC/Protobuf; від ручної персистентності до декларативної. Запропоновано матрицю вибору фреймворку з десяти критеріїв із декларативним деревом рішень та розглянуто аспекти спостережуваності та безпеки. Результати можуть бути використані при проектуванні нових розподілених систем та аудиті існуючих. Наукова новизна полягає в систематизації внутрішніх механізмів акторних фреймворків у порівняльному вимірі, який дотепер був відсутній у літературі.
Ключові слова: Інженерія програмного забезпечення, програмна архітектура, модель Акторів, розподілені системи, віртуальні актори, Erlang, Akka, Orleans, Proto.Actor, серіалізація, персистентність, кластеризація, mailbox, конкурентність.

Problem statement

The Actor model of computation, formalized by Carl Hewitt in 1973 [1], has become a common architectural paradigm in Cloud-Native and IoT systems over the past fifteen years. Its core primitive is an isolated entity that holds state, communicates only through asynchronous messages, and can spawn other actors. This maps directly onto the requirements of modern distributed systems: horizontal scalability, fault isolation, and distributed state management. Adoption spans telecommunications infrastructure (Ericsson AXD301 switches built on Erlang [3]), large-scale gaming services (Halo and Gears of War back-ends running on Microsoft Orleans [4]), financial technology with multiple Akka deployments, and polyglot microservice architectures using Proto.Actor [5]. The foundational premise of decoupling the sender from communication was a critical advance, enabling asynchronous control structures that remain essential for highly scalable, inconsistency-robust information systems today [2].

The growing popularity of the paradigm has produced a practical problem: framework proliferation makes the architect's choice harder. From the outside, all Actor-model frameworks look similar; they offer actors, messages, and supervision. Internally they work in very different ways. The choice between Erlang, Akka, Orleans, and Proto.Actor is a choice of underlying philosophy: how state is managed, how nodes are discovered, and how data is encoded. An uninformed choice produces architectural debt that surfaces only under production load. The problem is therefore: how to systematize the internal mechanics of contemporary Actor frameworks so that selecting a tool becomes an informed architectural decision rather than a matter of marketing preference or accidental familiarity?

Analysis of recent research and publications

The foundational reference is Hewitt's original work on the Actor formalism [1]. Armstrong's 2003 doctoral dissertation [3] gives the most complete study of the classical approach and justifies the «Let-It-Crash» philosophy together with the OTP supervision model. The virtual-actor branch originates in the 2014 Microsoft Research report by Bernstein et al. [4], which introduced Orleans in response to the programmability limitations of Erlang-style actors. The Dapr Actors specification [6] is a more recent reference; it generalizes the Orleans concept into a language-neutral sidecar runtime.

Official documentation of the frameworks (Akka [7], Orleans [8], Proto.Actor [9]) provides a thorough reference for their APIs, but does not offer cross-framework comparative analysis, as is typical for vendor documentation. Textbook treatments by Van Steen and Tanenbaum [10] and by Vernon [11] discuss the Actor model in general, but do not cover the internal mechanics of concrete frameworks in enough depth for architectural decision-making. Agha's monograph [12] remains the canonical theoretical reference, but predates most of the frameworks discussed here. Several articles address individual aspects, such as serialization benchmarks in the JVM ecosystem or failure-detector analyses in gossip-based clusters. A systematic cross-framework mapping along the axes of lifecycle, cluster topology, serialization, persistence, and concurrency has not been published.

Identification of previously unresolved parts of the general problem

The literature review reveals the following gap: there is no systematic comparative analysis that examines Actor-model frameworks simultaneously along the five basic architectural dimensions listed above. Existing comparisons tend to be vendor-centric, single-dimensional (for example, only serialization throughput), or focused on behavioural aspects (load balancing, failure recovery time) rather than on static mechanics. The present paper fills this gap and concentrates on the «under-the-hood» mechanics that determine the framework's basic capabilities before any workload is applied.

Research aim

The aim of this work is to systematize the basic mechanisms of state management, communication, lifecycle control, and concurrency in four leading Actor-model frameworks (Erlang/OTP, Akka, Orleans, and Proto.Actor) and to derive a framework-selection matrix usable by distributed-systems architects. The tasks of the study are:

- to present the historical evolution of the paradigm;
- to classify approaches to actor lifecycle management;
- to compare cluster-node discovery mechanisms;
- to analyse serialization and inter-node transport strategies;
- to systematize state persistence approaches;
- to compare concurrency models and mailbox implementations;
- to survey observability and security support;
- to formulate an integrated comparison matrix and framework-selection recommendations.

Materials and methods

This paper combines a documentary review with multi-criteria decision analysis (MCDA). No benchmarks were run; the comparison rests on primary documentation, foundational literature, and peer-reviewed work on related topics. Controlled workload testing is left to future work (see «Prospects for further research»).

A framework entered the set only with documented production use at scale (telecommunications, gaming, fintech, or cloud back-ends), authoritative documentation from its maintainer, and a clear place in one of two lifecycle philosophies: classical explicit lifecycle (Erlang, Akka, Proto.Actor) or virtual implicit lifecycle (Orleans).

Sources fell into three groups: foundational work (Hewitt 1973 [1], Armstrong 2003 [3], Bernstein et al. 2014 [4], Agha 1986 [12]); vendor documentation for Akka [7], Orleans [8], Proto.Actor [9], and

Dapr [6]; and peer-reviewed studies on serialization, failure detection, persistence, and concurrency ([13]–[21]).

The five comparison dimensions (lifecycle, cluster topology, serialization, persistence, concurrency) match the gap stated in «Identification of previously unresolved parts of the general problem». For MCDA, each dimension maps to one or two criteria, giving the ten-row matrix in Table 7 and the scores in Table 8.

Table M1 defines each criterion and names the source type used to score it.

Table M1. Operational definitions of comparison criteria

Criterion	Operational definition	Source class
Lifecycle control	Presence of explicit spawn/create and stop/exit calls in the public API and depth of supervision primitives	Official framework docs [3,7,9]
Preemption guarantee	Documented scheduler interruption of running execution units at bounded intervals independent of cooperative yield	Armstrong [3], BEAM docs, framework concurrency docs [7,8,9]
Cross-language interop	Number of officially supported host languages with native bidirectional message passing without an HTTP or gRPC gateway	Official framework docs [7,8,9]
Cluster discovery flexibility	Range of officially supported discovery backends (built-in registry, gossip, external coordinator)	Official framework docs [3,7,8,9]
Persistence model maturity	Depth of native persistence primitives: manual, snapshot, event-sourced, or hybrid	Official framework docs [7,8,9], [16], [18]
Mailbox flexibility	Number and configurability of documented mailbox variants (unbounded, bounded, priority, control-aware)	Official framework docs [3,7,8,9]
Schema evolution support	Forward- and backward-compatible field identifier scheme in the default wire format	Format specifications (Protobuf, ETF, Orleans IL-gen)
Default serialization safety	Documented absence of remote-code-execution vulnerabilities in the default serializer	CVE registries, vendor security advisories [7]
Practical node limit	Highest documented production deployment size cited by vendor or independent case study	Vendor case studies, [3], [15]
Entry barrier	Time-to-first-deployment indicated by official getting-started guides and tutorials	Official getting-started guides [3,7,8,9]

Scores use a single five-point ordinal scale across all criteria, based on what the documentation states: 1, absent or needs heavy external work; 2, third-party module only; 3, supported but must be configured; 4, first-class with light setup; 5, first-class with no extra setup. The numbers are ordinal, not measured throughput or latency. A later study could swap measured values into the same matrix for criteria where benchmarks exist.

Table M2 gives two kinds of weights. The baseline gives 0.10 to each of the ten criteria. Four domain profiles shift weight toward what matters in that setting: polyglot microservices, .NET game back-ends, event-driven fintech with CQRS, and soft-real-time telecom with strict availability. Weights in every profile sum to 1.00.

Table M2. Per-scenario weighting profiles for MCDA aggregation

Criterion	Equal baseline	Polyglot microservices	.NET back-ends	Fintech CQRS	Telecom high-availability
Lifecycle control	0.10	0.05	0.05	0.05	0.05
Preemption guarantee	0.10	0.05	0.05	0.05	0.40
Cross-language interop	0.10	0.25	0.05	0.05	0.05
Cluster discovery flexibility	0.10	0.15	0.10	0.10	0.05
Persistence model maturity	0.10	0.10	0.15	0.30	0.05
Mailbox flexibility	0.10	0.05	0.05	0.10	0.10
Schema evolution support	0.10	0.15	0.10	0.10	0.05
Default serialization safety	0.10	0.10	0.05	0.10	0.05
Practical node limit	0.10	0.05	0.10	0.05	0.10
Entry barrier	0.10	0.05	0.30	0.10	0.10
Total	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**

For framework *f* under profile *p*, the composite score is a weighted sum over the ten criteria:

$S(f, p) = \sum_i w_i(p) \cdot s_i(f)$, where $s_i(f)$ is the score of *f* on criterion *i* and $w_i(p)$ is the weight of *i* in profile *p*. The highest $S(f, p)$ is the suggested match for that profile. Table 8 lists $s_i(f)$; Table 9 lists the weighted totals.

The MCDA snapshot reflects documentation as of mid-2026 and depends on the rubric and weights chosen here. Throughput, latency, and fault tolerance were not measured. Ordinal scores also collapse differences between frameworks that land on the same number.

Main body

Historical evolution of actor frameworks. Before the analysis of internal mechanics, it is helpful to position the frameworks chronologically, because many architectural decisions were shaped by the computing context of their respective decades. Table 1. summarizes the main milestones.

Table 1. Timeline of the Actor model and its implementations

Year	Milestone
1973	C. Hewitt et al. publish the Actor formalism at IJCAI [1]
1986	Ericsson internal Erlang prototype (J. Armstrong, R. Virding, M. Williams)
1998	Erlang released as open-source software
2003	Armstrong's PhD thesis formalizes Let-It-Crash and OTP [3]
2009	First release of Akka on the JVM
2010	Akka 2.0 introduces Cluster module with gossip protocol
2011	First public Orleans presentation at SOCC [5]
2014	Microsoft Research report «Orleans: Distributed Virtual Actors» [4]
2015	Orleans goes open-source; Halo 4/5 confirmed as production users
2017	Proto.Actor released (Asynkron AB, R. Bogusz) targeting Go, C#, Kotlin
2019	Dapr 1.0 announced by Microsoft; Actors block standardizes virtual actors across
2022	Orleans 7 ships a redesigned code-generation serializer
2023	Akka 2.8 stabilizes the Typed API as the default
2024	Lightbend introduces Akka SDK focusing on polyglot solution and serverless

Two observations follow from the timeline. First, the classical camp (Erlang, Akka) predates the cloud era and therefore carries assumptions about relatively stable, operator-managed clusters, whereas the virtual camp (Orleans, Dapr) was designed for the ephemeral nature of cloud resources. Second, Proto.Actor, although the youngest of the four, inherits the classical philosophy but reimplements it on top of cloud-native transports (gRPC) and coordination services (Consul, etcd). This shows a convergence trend.

Actor lifecycle and entity concept. The first architectural dimension addresses the question of who controls the life of an actor: the developer or the platform? Along this axis the four frameworks split into two distinct camps.

Classical approach with explicit lifecycle in Erlang/OTP, an actor is created by an explicit call to spawn/1, spawn_link/1, or a higher-level OTP primitive such as gen_server:start_link/3 [3]. The lifecycle is governed by supervision trees with four canonical restart strategies (one_for_one, one_for_all, rest_for_one, and simple_one_for_one). Termination is likewise explicit: exit(normal), abnormal exit, or forced kill. The «Let-It-Crash» philosophy assumes that the actor does not attempt self-recovery; the supervisor recreates it instead. This allows processes to safely fail during execution while the underlying framework provides a defined recovery strategy to restart affected entities cleanly [13]. Akka on the JVM follows the same philosophy. Actors are created via actorOf(Props) from a parent context, placed into a hierarchical path space (/user/, /system/, /deadLetters/), and expose lifecycle hooks (preStart, postStop, preRestart, postRestart) [7]. Termination is triggered by PoisonPill, Kill, or gracefulStop, and supervision strategies (OneForOneStrategy, AllForOneStrategy) are configured through a Decider function that maps exceptions to reactions.

Virtual approach with implicit lifecycle in Microsoft Orleans inverts this philosophy [3, 7]. A grain (the Orleans term for an actor) is treated as a logical entity that exists forever from the application's point of view. The developer never creates and never destroys a grain; they obtain a proxy to it through GrainFactory.GetGrain<T>(id). The Orleans runtime handles activation (loading the grain into memory on first reference) and deactivation, also called Grain Garbage Collection (unloading after a configurable idle period, typically two hours by default). The benefits are elasticity and natural migration of state on node failure. The virtual actor paradigm has further expanded into stateful serverless computing, where frameworks expose shared log APIs to functions, decoupling read and write paths to independently scale throughput and fault tolerance [14]. The challenges are cold-start latency on first activation and the need for balanced rebalancing. Dapr Actors [6] extend this philosophy into a language-neutral sidecar model coordinated by a Placement Service.

Hybrid approach. Proto.Actor [4, 8] adopts the Akka philosophy of explicit spawn and supervision, but strips the JVM overhead and supports Go, C#, Kotlin, and Python. Reflection-based dispatch is avoided in favour of generated code. A Cluster Grains mode is available as an optional virtual-actor layer, but it is not the framework's core model.

This aspects are shown in Table 2 which presents the lifecycle comparison between solutions.

Table 2. Lifecycle management comparison

Aspect	**Erlang/OTP**	**Akka**	**Proto.Actor**	**Orleans**
Creation	Explicit (spawn)	Explicit (actorOf)	Explicit (Spawn)	Implicit (GetGrain)
Lifecycle responsibility	Developer + Supervisor	Developer + Supervisor	Developer + Supervisor	Runtime platform
Recovery after failure	Supervisor-driven restart	Supervisor-driven restart	Supervisor-driven restart	Automatic reactivation
Notion of «death»	Explicit (exit)	Explicit (PoisonPill)	Explicit (Stop)	Absent (deactivation ≠ death)
Programming effort	High	High	Medium	Low

The classical approach trades programming simplicity for control and determinism; the virtual approach trades control for operational convenience.

Cluster topology and node-discovery mechanisms. This subsection examines how a cluster of actor-hosting nodes form and maintain themself, excluding the dynamics of load balancing and failure-induced rebalancing.

Erlang nodes discover each other through the Erlang Port Mapper Daemon, running on TCP port 4369 [3]. Authentication is cookie-based, and every node maintains a direct TCP connection to every other node, forming a full mesh of $N \times (N - 1) / 2$ links. This design guarantees transparent location-independent message passing, where a remote send is syntactically identical to a local one. Armstrong [3] and Erlang operational guides treat the full mesh as practical up to roughly 100–200 nodes, where connection and heartbeat cost grows quadratically with cluster size.

Akka Cluster replaces full-mesh connectivity with a gossip protocol loosely based on SWIM [7]. Membership state transitions through a lifecycle of Joining - Up - Leaving - Exiting - Removed. Seed nodes provide the initial contact point, and an adaptive Phi Accrual Failure Detector gives probabilistic, network-condition-aware failure judgments rather than binary alive or dead decisions. By adopting a gossip protocol combined with the actor model, these systems eliminate centralized synchronization bottlenecks, making them highly effective for peer-to-peer data distribution and parallel processing under unstable network conditions [15]. Cluster Sharding is offered as a separate module for distributing actors across nodes. The architecture scales to thousands of nodes.

Orleans has no single coordinator. Grain locations are tracked by a Distributed Grain Directory implemented over a consistent-hash ring, a DHT-style structure [3, 7]. A Membership Provider (Azure Table, SQL Server, ZooKeeper, or Consul) maintains the list of live silos. Placement strategies include Random, PreferLocal, HashBased, ActivationCountBased and Stateless. A grain can migrate between silos as the topology changes.

Proto.Actor relies on external systems (Consul, etcd, or the Kubernetes API) for service discovery and membership [4, 8]. Gossip propagates membership among cluster members; there is no EPMD-style monolithic registry. The virtual-actor layer (Cluster Grains) sits on top of this coordination substrate.

Lets review the Table 3 which summarizes cluster-level properties for each of the frameworks.

Table 3. Cluster topology and discovery comparison

Characteristic	**Erlang/OTP**	**Akka**	**Orleans**	**Proto.Actor**
Discovery mechanism	EPMD + cookie	Gossip + seed nodes	Membership provider (Azure/SQL)	Consul/etcd/K8s
Connection topology	Full mesh	Partial mesh (gossip)	Client-to-silo	Client-to-member
Failure detector	Net kernel ticks	Phi Accrual FD	Lease-based	Gossip heartbeat
Practical node limit	≈ 100–200	Thousands	Thousands	Thousands
External dependency	None	None	Required (database)	Required (Consul/etcd)

The evolution visible in Table 3 is clear: from a self-contained monolithic registry (Erlang), through embedded distributed membership (Akka), to externalized coordination over cloud-native infrastructure (Orleans, Proto.Actor).

Serialization and inter-node communication. The way messages are packed on the wire determines achievable throughput, cross-language interoperability, and schema-evolution flexibility. Five criteria apply here: throughput, on-wire size, cross-language interoperability, schema evolution and deserialization security.

Erlang uses External Term Format (ETF). ETF is the native binary format of the BEAM virtual machine, accessed via term_to_binary/1 and binary_to_term/1. It supports every native Erlang type and requires zero configuration. Its weakness is lock-in: interoperability outside Erlang/Elixir requires explicit BERT-RPC bridges.

Historically, Akka inherited Java Serialization as the default. Akka's documentation now treats it as slow and unsafe and no longer recommends it [7]. JVM serialization studies cited in [7] report Protocol

Buffers reaching roughly an order of magnitude higher throughput than Java Serialization on typical payloads. CVE-2015-4852 and later Apache Commons Collections gadget-chain advisories showed the same default as an RCE-class deserialization risk. Contemporary Akka offers Jackson (JSON or CBOR), Protocol Buffers, Avro, and community Kryo modules, but each must be explicitly registered in application.conf. Remote transport is handled by Akka Artery over TCP or UDP/Aeron.

Proto.Actor was built from day one on gRPC and Protocol Buffers [4, 8]. The consequences are substantial: native polyglotism, a strict message schema as a contract, and HTTP/2 multiplexing without additional configuration. An actor written in Go can send a message to an actor written in C# without any adapter layer. The trade-off is the requirement that every message type be described in a .proto file; untyped or dynamically shaped messages are not first-class citizens.

Orleans 7 and later ship a built-in serializer that generates IL code at build time for every grain message type [8]. Throughput is close to hand-written serializers, and the model supports polymorphism, generics and backward-compatible schema evolution via [Id] field attributes. The cost is tight coupling to the .NET runtime: cross-language access must go through an HTTP or gRPC gateway rather than through native grain calls.

All together this was shown in Table 4 that summarizes the serialization in each of actor frameworks.

Table 4. Serialization and inter-node communication comparison

Parameter	**Erlang (ETF)**	**Akka (Protobuf)**	**Proto.Actor (gRPC)**	**Orleans (IL-gen)**
Default format	ETF	Jackson/JSON	Protobuf	Orleans Serializer
Cross-language	No	Yes (with config)	Yes (native)	No
Schema evolution	Implicit	Protobuf field numbers	Protobuf	[Id] attributes
Transport	TCP (erl dist)	Aeron/TCP (Artery)	HTTP/2 (gRPC)	TCP (custom)
Configuration effort	Zero	High	Medium (.proto)	Low (attributes)
Deserialization safety	High	Historically low	High	High

By the cross-language criterion in Table M1, only Proto.Actor supports native bidirectional messaging across more than two host languages (Go, C#, Kotlin, Python) without an HTTP or gRPC gateway [9]. Orleans 7 generates per-type marshaling code at build time and avoids reflection [8]; vendor docs describe it as the fastest built-in option on .NET. ETF needs no extra setup inside BEAM [3]. Akka asks for the most wiring: every non-default serializer must be registered in application.conf [7].

State management and persistence. The fourth dimension concerns how systems guarantee that actor state survives node restarts. Three canonical approaches exist: event sourcing (store the sequence of events and reconstruct state by replay), snapshot-based (store the current state as a single image), and hybrid (snapshots plus events since the last snapshot).

Erlang ships without built-in persistence. The canonical pattern is a stateful gen_server that periodically persists to Mnesia, ETS or DETS tables. For cluster-wide state, Riak Core is often used as an external framework. The developer bears full responsibility for consistency and recovery logic. Furthermore, advanced distributed document stores frequently adopt Conflict-free Replicated Data Types (CRDT) to handle highly concurrent updates, providing strong eventual consistency globally without forcing developers to implement manual conflict resolution logic [16].

Akka exposes PersistentActor (Classic API) and EventSourcedBehavior (Typed API) as native abstractions. Events are appended to a Journal (Cassandra, JDBC or LevelDB), periodic Snapshots accelerate recovery, and Akka Projection offers a CQRS path. The advantages are a complete audit log, the ability to re-project events into new read models and a natural fit for complex domains. The challenges are a steep learning curve, event-versioning discipline and eventual consistency in the CQRS path [7]. By strictly separating state-mutating commands from queries and relying on an append-only event store, the CQRS and Event Sourcing combination enables completely reactive, scalable architectures that can project multiple, distinct read models over time [17].

Orleans exposes persistence declaratively. A grain holds a property IPersistentState<T> annotated with [PersistentState("profile", "redis")] and the developer explicitly calls await state.WriteStateAsync() after modification. Providers include Azure Table, Azure Blob, SQL Server, Redis, DynamoDB and MongoDB [8]. The advantages are a low barrier to entry and automatic serialization. The challenges are a single-writer bottleneck (only the owning grain writes its state), potential hot partitions under uneven load and the absence of native event sourcing (community add-ons such as Orleans.EventSourcing partially fill this gap). However, the platform has recently evolved closer to an actor-oriented database by introducing distributed ACID transactions across grains, utilizing early lock release to avoid blocking operations across cloud storage systems [18].

Proto.Actor persistence pattern is similar to Akka's: a PersistentActor with an event journal backed by SQL, DynamoDB, or Redis. Event sourcing is the recommended default [4, 8].

The comparative summary is given in Table 5 which summarizes persistence behaviour.

Table 5. Persistence comparison

Framework	**Primary approach**	**Storage backends**	**Snapshot support**	**Event sourcing**
Erlang/OTP	Manual (Mnesia/ETS)	Mnesia, ETS, DETS	Manual	Via external databases
Akka	Event Sourcing (native)	Cassandra, JDBC	Yes (native)	Yes (core feature)
Orleans	Snapshot (native)	Azure, SQL, Redis	Yes (= state)	Via add-ons
Proto.Actor	Event Sourcing	SQL, DynamoDB	Yes	Yes

Akka and Proto.Actor align with event-driven domains such as fintech and audit-heavy systems. Orleans favours CRUD-shaped scenarios with rapid development. Erlang leaves the choice entirely to the developer.

Concurrency model and message mailboxes. Beyond whether messages are persisted or serialized, frameworks differ substantially in how messages are scheduled and queued inside each actor. This dimension strongly influences achievable throughput, back-pressure behaviour and protection against starvation.

Erlangs BEAM virtual machine runs a per-scheduler ready queue and uses preemptive scheduling based on reduction counting: every process is guaranteed execution time after approximately 2 000 reductions, regardless of whether it voluntarily yields [3]. Processes are lightweight (initial heap of roughly 233–338 words), and a single node routinely hosts hundreds of thousands of them. Each process has its own mailbox, implemented as a FIFO queue with selective receive via pattern matching. The selective-receive model is unique: a process can inspect and skip messages without dequeuing them. The approach simplifies protocol design but can create unbounded-scan cost when many non-matching messages accumulate.

Akka separates dispatchers (thread-pool abstraction) from mailboxes (per-actor queues). Four dispatcher types are provided out of the box: Dispatcher (fork-join, default), PinnedDispatcher (one thread per actor), CallingThreadDispatcher (synchronous, for tests), and AffinityPoolDispatcher (thread-actor affinity). Mailbox variants include UnboundedMailbox (default), BoundedMailbox (explicit back-pressure), UnboundedPriorityMailbox, and ControlAwareMailbox (priority for control messages). Scheduling inside an actor is cooperative: the actor is pinned to a thread for a configurable throughput window (default of 5 messages) before the thread is returned to the pool [7]. Java's lack of preemption means a misbehaving actor can block a thread for an arbitrary time. Consequently, accurately modeling the resource consumption and timed behavior of such cooperative scheduling is often necessary to predict system performance under heavy load [19]

Microsoft Orleans has turn-based single-threaded execution. Each grain processes one request at a time on an implicit single-logical-thread model, which removes the need for locks inside grain code [3, 7]. Two escape hatches are provided. The [Reentrant] attribute allows multiple asynchronous turns to interleave on the same grain (useful when none of them mutate state concurrently), and [AlwaysInterleave] marks methods as safe to interleave. Request queuing is transparent; there is no explicit mailbox API exposed to the developer. The runtime uses the .NET task scheduler and may multiplex thousands of grains onto a small thread pool.

Proto.Actor: mailbox-dispatcher model, goroutine-native in Go. Proto.Actor's model is conceptually close to Akka's (dispatcher plus mailbox), but in the Go edition every actor is backed by a dedicated goroutine rather than a shared thread pool, which uses Go's lightweight concurrency [9]. Mailbox types include unbounded, bounded, and priority. Default throughput per mailbox pass is 300 messages. As in Go generally, scheduling is cooperative at the goroutine level and preemptive at the OS-thread level.

Table 6 consolidates the concurrency model comparison for examined frameworks.

Table 6. Concurrency model comparison

Parameter	**Erlang/OTP**	**Akka**	**Orleans**	**Proto.Actor**
Scheduler	Preemptive (reductions)	Cooperative fork-join	.NET task scheduler	Go scheduler / thread pool
Execution unit	BEAM process	JVM thread share	Turn (single-threaded)	Goroutine / thread share
Mailbox variants	FIFO + selective receive	Unbounded / Bounded / Priority / Control-aware	Implicit (runtime-managed)	Unbounded / Bounded / Priority
Reentrancy	N/A (process-per-actor)	Not applicable (one message at a time)	[Reentrant] / [AlwaysInterleave]	Not applicable
Starvation protection	Guaranteed (preemptive)	Throughput-window only	Fair queueing per grain	Throughput-window only
Back-pressure	Process-flag, explicit	Bounded mailbox	Implicit via turn queue	Bounded mailbox

Only Erlang/OTP documents hard scheduler preemption: a BEAM process is cut off after about 2 000 reductions even if it never yields [3]. Orleans keeps grain code on a single logical thread per turn; reentry is opt-in via [Reentrant] and [AlwaysInterleave] [8], which simplifies grain code but limits fine-grained control over interleaving. Akka [7] and Proto.Actor [9] use the dispatcher–mailbox pattern with cooperative scheduling capped by throughput windows; Akka shares JVM threads across actors, while Proto.Actor in Go backs each actor with a goroutine.

Observability and security. For completeness, two cross-cutting concerns deserve a brief mention, as they are often decisive in enterprise contexts.

As per observability: Erlang/OTP ships with the :observer graphical tool, the built-in logger, and SASL reports that expose supervisor events [3]. Akka integrates with OpenTelemetry via the Lightbend Telemetry agent and exposes cluster-state through the Akka Management HTTP endpoint. Orleans ships the OrleansDashboard add-on and pluggable telemetry providers for Application Insights, Prometheus, and OpenTelemetry. Proto.Actor exports metrics and traces through native OpenTelemetry instrumentation. Modern observability is increasingly supplemented by automated chaos engineering techniques, which systematically inject failures - such as network partitions or heartbeat delays-to proactively identify cluster weaknesses and validate recovery mechanisms [20].

Security of inter-node communication. Erlang offers TLS-encrypted Distribution and cookie-based node authentication [3]. Akka supports TLS termination in Artery and pluggable authentication on Akka HTTP management endpoints. Orleans supports TLS per silo connection and integrates with ASP.NET Core authentication for grain-level authorization. Proto.Actor inherits mTLS support from gRPC and can also be deployed behind service-mesh proxies such as Istio or Linkerd for uniform security policy.

Framework-selection recommendations. Table 7 consolidates the five dimensions into a ten-criterion framework-selection matrix.

Table 7. Consolidated framework-selection matrix

Criterion	**Erlang/OTP**	**Akka**	**Proto.Actor**	**Orleans**
Lifecycle philosophy	Explicit, Let-It-Crash	Explicit, Supervision	Explicit (+ optional virtual)	Implicit, virtual actors
Primary language	Erlang/Elixir	Scala/Java/Kotlin	Go/C#/Kotlin/Python	C#/.NET
Clustering	EPMD full mesh	Gossip	Consul/etcd/K8s	Distributed Directory

Serialization	ETF (native)	Protobuf/Jackson (config)	gRPC/Protobuf (native)	IL-gen (native)
Cross-language	No	Partial	Yes (first-class)	No
Persistence	Manual	Event Sourcing	Event Sourcing	Snapshot
Concurrency model	Preemptive per-process	Dispatcher + mailbox	Goroutine + mailbox	Turn-based per grain
Entry barrier	High	High	Medium	Low
Practical node limit	≈ 100–200	Thousands	Thousands	Thousands
Ecosystem maturity	Mature, narrow	Mature, JVM-wide	Young, polyglot	Mature, .NET-wide

Based on the consolidated matrix, the following decision heuristics are proposed:

Table 8 applies the rubric from Table M1 to the qualitative summary in Table 7. Each cell follows the matching subsection above.

Table 8. MCDA score matrix per criterion (ordinal scale 1–5)

Criterion	Erlang/OTP	Akka	Proto.Actor	Orleans
Lifecycle control	5	5	5	2
Preemption guarantee	5	2	3	2
Cross-language interop	1	3	5	2
Cluster discovery flexibility	2	4	5	5
Persistence model maturity	2	5	3	4
Mailbox flexibility	3	5	3	2
Schema evolution support	2	4	5	4
Default serialization safety	5	3	5	5
Practical node limit	2	5	4	5
Entry barrier	2	2	2	5

Table 9 multiplies Table 8 scores by the weights in Table M2. The highest total in each row is shown in bold.

Table 9. Aggregated MCDA weighted totals per scenario profile

Profile	Erlang/OTP	Akka	Proto.Actor	Orleans
Equal weights baseline	2.90	3.80	**4.00**	3.60
Polyglot microservices	2.40	3.70	**4.40**	3.55
.NET game back-ends	2.45	3.55	3.50	**4.15**
Event-driven fintech with CQRS	2.65	**4.05**	3.75	3.85
Soft-real-time telecom high-availability	**3.55**	3.20	3.50	3.10

Table 9 points to four selection patterns. Anyone can recompute them from Tables M1, M2, and 8 using the same weights.

- Polyglot microservices with cross-platform teams: Proto.Actor (4.40). Cross-language interop carries weight 0.25; schema evolution, 0.15.
- Enterprise .NET development, rapid iteration, game back-ends: Orleans (4.15). Entry barrier is weighted 0.30; persistence maturity, 0.15.
- Event-driven domains with CQRS, including financial systems: Akka (4.05). Persistence maturity is weighted 0.30; mailbox flexibility, 0.10 (control-aware mailboxes for priority control traffic).
- Telecommunications and soft-real-time systems that need hard preemption: Erlang/OTP (3.55). Preemption carries weight 0.40; Erlang alone scores 5 on that criterion in Table 8.

Conclusions and scientific novelty

The study has systematized four leading Actor-model frameworks: Erlang/OTP, Akka, Orleans, and Proto.Actor along five architectural dimensions that are usually treated in isolation: actor lifecycle, cluster discovery, serialization, persistence, and concurrency model. Three evolutionary trends have been identified:

1. From monolithic to externalized discovery. Erlang's EPMD gave way first to Akka's embedded gossip and then to the Orleans- and Proto.Actor-style reliance on external coordination systems like Consul, etcd, Kubernetes, cloud databases. This trend aligns Actor frameworks with the cloud-native paradigm.

2. From native to cross-platform serialization. The lineage runs from Erlang's BEAM-only ETF, through Akka's Jackson/Protobuf compromise, to Proto.Actor's fully language-agnostic gRPC/Protobuf model. Orleans retains platform-native speed at the cost of cross-language reach.

3. From manual to declarative persistence. Manual ETS/Mnesia patterns evolved into event-sourced journals (Akka, Proto.Actor) and declarative snapshot providers (Orleans).

The scientific novelty of the work is the systematization of the internal mechanisms of Actor-model frameworks in an integrated five-dimensional comparative setting that has so far been absent from the literature.

On the practical side, Table 7 lists ten selection criteria; Tables M1 and M2 define how to score and weight them; Tables 8 and 9 carry the scores and totals. An architect can rerun the same arithmetic for a given domain instead of defaulting to a familiar stack. The numbers in Tables 8 and 9 come from the ordinal rubric in «Materials and methods», not from benchmarks. Controlled workload tests for criteria such as throughput and fault recovery remain for later work. The same tables can double as an audit checklist for systems already in production.

Prospects for further research

Three main directions emerge for future research.

First, a shared benchmark suite under identical workloads could replace selected ordinal scores in Table 8 with measured throughput, latency, and fault-tolerance values, then rerun the weighted sums in Table 9. That would turn today's documentary MCDA into a mixed documentary-and-measurement study without changing the framework of Tables 7–9.

Second, conducting a systematic experimental study on framework behavior under Gray Failures. Such partial or asymmetric failures violate the assumptions built into traditional failure detectors, exposing the critical vulnerability of differential observability [21], where a system appears healthy but application logic is severely impacted.

Third, researching architectural improvements to maximize overall system resilience and throughput, specifically through the design and implementation of a novel actor entity sharding mechanism to optimize dynamic load distribution and state recovery.

References:

1. Hewitt, C.; Bishop, P.; Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI). 1973. P. 235-245.
2. Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. arXiv. 2010. URL: https://doi.org/10.48550/arxiv.1008.1459 (date of access: 29.04.2026).
3. Armstrong, J. Making reliable distributed systems in the presence of software errors. PhD Thesis. Royal Institute of Technology (KTH). 2003. 295 p.
4. Bernstein, P.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J. Orleans: Distributed Virtual Actors for Programmability and Scalability. Microsoft Research Technical Report MSR-TR-2014-41. 2014. 23 p.
5. Bykov, S.; Geller, A.; Kliot, G.; Larsson, G.; Pandya, R.; Thelin, J. Orleans: Cloud Computing for Everyone. Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11). 2011. Article 16. P. 1-14. URL: https://doi.org/10.1145/2038916.2038932 (date of access: 29.04.2026).
6. Dapr Actors: Official Documentation. Dapr. 2024. URL: https://docs.dapr.io/developing-applications/building-blocks/actors/ (date of access: 20.04.2026).
7. Akka Documentation. Lightbend Inc. 2024. URL: https://doc.akka.io (date of access: 20.04.2026).
8. Microsoft Orleans Documentation. Microsoft Corp. 2024. URL: https://learn.microsoft.com/dotnet/orleans/ (date of access: 20.04.2026).
9. Proto.Actor Documentation. Asynkron AB. 2024. URL: https://proto.actor (date of access: 20.04.2026).
10. Van Steen, M.; Tanenbaum, A. S. Distributed Systems: Principles and Paradigms. 3rd ed. CreateSpace. 2017. 596 p.
11. Vernon, V. Reactive Messaging Patterns with the Actor Model. Addison-Wesley. 2015. 400 p.
12. Agha, G. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press. 1986. 160 p. URL: https://doi.org/10.7551/mitpress/1086.001.0001 (date of access: 29.04.2026).
13. Francalanza, A.; Tabone, G. ElixirST: A session-based type system for Elixir modules. Journal of Logical and Algebraic Methods in Programming. 2023. 135. P. 100891. URL: https://doi.org/10.1016/j.jlamp.2023.100891 (date of access: 29.04.2026).
14. Jia, Z.; Witchel, E. Boki: Stateful Serverless Computing with Shared Logs. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021. P. 691-707. URL: https://doi.org/10.1145/3477132.3483541 (date of access: 29.04.2026).

15. Taamneh, S.; Al-Hami, M.; Bani-Salameh, H.; Abdallah, A. E. A Robust Distributed Clustering of Large Data Sets on a Grid of Commodity Machines. Data. 2021. 6(7). P. 73. URL: https://doi.org/10.3390/data6070073 (date of access: 29.04.2026).
16. Rinberg, A.; Solomon, T.; Shlomo, R.; Khazma, G.; Lushi, G.; Keidar, I.; Ta-Shma, P. DSON. Proceedings of the VLDB Endowment. 2022. 15(1). P. 1053-1065. URL: https://doi.org/10.14778/3510397.3510403 (date of access: 29.04.2026).
17. Debski, A.; Szczepanik, B.; Malawski, M.; Spahr, S.; Muthig, D. A Scalable, Reactive Architecture for Cloud Applications. IEEE Software. 2018. 35(1). P. 62-71. URL: https://doi.org/10.1109/ms.2017.265095722 (date of access: 29.04.2026).
18. Eldeeb, T.; Burckhardt, S.; Bond, R.; Cidon, A.; Yang, J.; Bernstein, P. A. Cloud Actor-Oriented Database Transactions in Orleans. Proceedings of the VLDB Endowment. 2024. 17(1). P. 3720-3730. URL: https://doi.org/10.14778/3685800.3685801 (date of access: 29.04.2026).
19. Schlatte, R.; Johnsen, E. B.; Kamburjan, E.; Tapia Tarifa, S. L. Modeling and Analyzing Resource-Sensitive Actors: A Tutorial Introduction. Lecture Notes in Computer Science. 2021. P. 3-19. URL: https://doi.org/10.1007/978-3-030-78142-2_1 (date of access: 29.04.2026).
20. Kikuta, D.; Ikeuchi, H.; Tajiri, K. ChaosEater: Fully Automating Chaos Engineering with Large Language Models. arXiv. 2025. URL: https://doi.org/10.48550/arxiv.2501.11107 (date of access: 29.04.2026).
21. Huang, P.; Guo, C.; Zhou, L.; Lorch, J. R.; Dang, Y.; Chintalapati, M.; Yao, R. Gray Failure. Proceedings of the 16th Workshop on Hot Topics in Operating Systems. 2017. P. 1-7. URL: https://doi.org/10.1145/3102980.3103005 (date of access: 29.04.2026).

Історія статті:
Отримано: 26.05.2026 Доопрацьовано: 22.09.2026 Прийнято до друку: 23.09.2026 Опубліковано: 29.09.2026