A Survey of Actor-Like Programming Models for Serverless Computing

Jonas Spenger[1,2][0000−0002−7119−5234], Paris Carbone[1,2][0000−0002−9351−8508], and Philipp Haller[1][0000−0002−2659−5271]

[1] Digital Futures and EECS, KTH Royal Institute of Technology, Stockholm, Sweden
{jspenger,parisc,phaller}@kth.se
[2] Computer Systems, RISE Research Institutes of Sweden, Stockholm, Sweden
{jonas.spenger,paris.carbone}@ri.se

Abstract. Serverless computing promises to significantly simplify cloud computing by providing Functions-as-a-Service where invocations of functions, triggered by events, are automatically scheduled for execution on compute nodes. Notably, the serverless computing model does not require the manual provisioning of virtual machines; instead, FaaS enables load-based billing and auto-scaling according to the workload, reducing costs and making scheduling more efficient. While early serverless programming models only supported stateless functions and severely restricted program composition, recently proposed systems offer greater flexibility by adopting ideas from actor and dataflow programming. This paper presents a survey of actor-like programming abstractions for stateful serverless computing, and provides a characterization of their properties and highlights their origin.

Keywords: Actor Model · Active Objects · Serverless Computing · Dataflow · Stateful Serverless · Distributed Programming · Cloud Computing

1 Introduction

Serverless computing has greatly simplified building cloud applications by providing Functions-as-a-Service (FaaS), a programming model consisting of *functions* and *event triggers*. These functions are automatically scheduled for execution on compute nodes, elastically scaling with the load [22]. In effect, the serverless model fully abstracts away the underlying computing infrastructure, billing and running user code on-demand. As a consequence, serverless computing can reduce costs and make scheduling more efficient.

While early serverless models were restricted, recent developments have introduced more flexible abstractions. The first major serverless frameworks, such as AWS Lambda [6] and similar [31,40,51], were restricted to: 1) *stateless functions*; and 2) limited compositional primitives such as no direct function-to-function messaging, often-cited challenges with serverless computing [12,36,42]. Recent developments, however, have seen programming models supporting stateful serverless that overcome these challenges through abstractions closely related

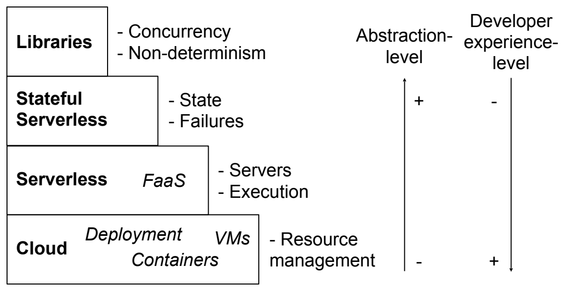


Fig. 1: Levels of abstraction for distributed programming.

to dataflow programming and the actor model [18,17,29,63,61,53,62,47]. We refer to these as *actor-like programming models for serverless computing*, this can also be referred to as *stateful serverless*.

These stateful serverless programming models are an abstraction of the underlying computing infrastructure. Conceptually, we can represent the abstraction levels of utility computing for distributed applications as a step-ladder, as shown in Figure 1, ranging from low-level cloud resources to abstract, virtualized applications.

In this representation, *stateful serverless* is on the third layer, aiming to abstract away application state and masking failures, providing abstractions for deploying failure-free stateful functions with powerful compositional primitives. The stateful serverless layer provides powerful abstractions for building distributed applications and is used increasingly to build libraries or compose stronger abstraction levels (*e.g.,* level 4, abstracting from concurrency and non-determinism). In contrast to lower layers, it abstracts away failure and state management, which are difficult to get right.

This paper surveys actor-like programming models for serverless computing. The purpose is to provide a background on the development of these models; provide a characterization thereof; describe their challenges with respect to a serverless execution (state management and fault tolerance); highlight the similarities and differences of popular implementations; and provide an outlook on research directions. For this purpose, we survey eight implementations in detail [18,17,29,63,61,53,62,47], and include other relevant works in the whole analysis. In particular, we find three key enabling principles for their serverless execution to be of importance: they are virtualized, decoupled; they are data-parallel; they are slightly less dynamic than traditional actors.

Recent surveys have studied serverless computing [22,12,42,36,26,49], the actor model [43], the active objects model [14], and other related fields [11,50]. In contrast to surveys on serverless computing [22,12,42,36,26,49], the presented analysis puts more focus on the programming model and its properties. Actor systems have been studied extensively [43], whereas this survey sheds more light on properties at the intersection of actors and serverless such as per-key execution semantics, fault tolerance, and execution guarantees. Similarly, this applies

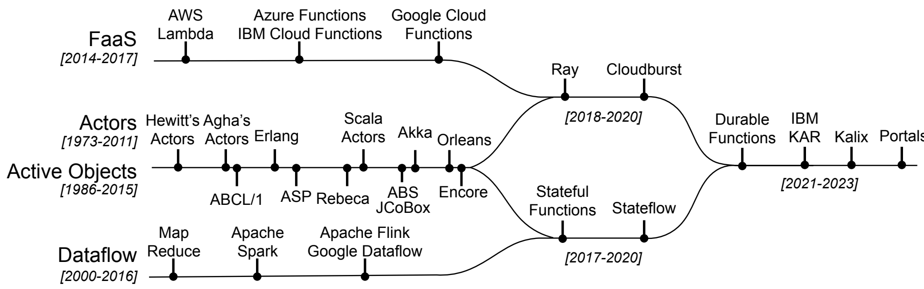


Fig. 2: An overview over related programming models and systems with effective periods.

also to active object languages [14]. Reactive programming [11], and vertex-centric programming [50], also share some similarities with the discussed topics here, yet they lack some of the dynamic messaging properties of actors.

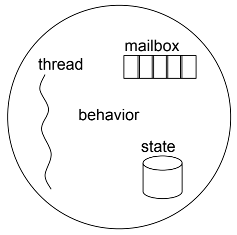
The rest of this paper is structured as follows. In section 2, we provide a background on the development of the actor model, active object model, dataflow processing, serverless, and actor-like serverless models. Section 3 discusses the main challenges of programming systems for stateful serverless computing. Next, in section 4, we analyse the distinctive characteristics of these systems, and compare their properties with respect to programming model (subsection 4.1) and serverless execution (subsection 4.2). Finally, we outline promising research directions (section 5), and provide a conclusion (section 6).

2 Background

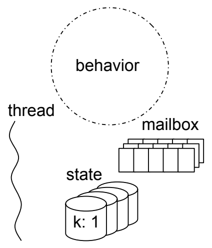
This section provides a background on the development of actor-like programming models in the context of serverless computing, traced back to Actor and Active Object systems, Dataflow platforms, and Functions-as-a-Service (FaaS). To that end, Figure 2 presents a timeline of related systems in their respective areas. We discuss the main directions in more detail with the aim to identify distinct characterizations and their development.

2.1 Actors

The Actor Model is a programming model for distributed, concurrent programming. It was invented in 1973 by Carl Hewitt [39], originally as a formalism for reasoning agents (in the context of artifical intelligence) and distributed parallel computations [38]. Additional significant work on the Actor Model was performed by Gul Agha, who provided a semantic formalization [3], and proposed the model as a "framework for concurrent systems" [1]. Since then it has seen a myriad of implementations with heavy industry adoption [43]. Notable actor implementations include Erlang [9], Scala Actors [34,33], Akka [46], and Pony [24].



(a) Non-virtual actor: coupled mailbox, thread, behavior, and state.



(b) Virtual actor: fully decoupled and externalized mailbox, thread, behavior, and state.

Fig. 3: Regular and virtual actor types.

In essence, an actor is a concurrent object that can perform three different actions [2]: 1) *create* other actors; 2) *message* other actors; and 3) modify its *state* (or behavior) for the next received message. This style of actor corresponds to the non-virtual actor in Figure 3a: an actor consists of an executing thread, mailbox, state and behavior.

A key principle of actor execution is the "isolated turn principle" [43], that is, the processing of a message by an actor (*i.e.,* a turn) can be viewed as a single isolated step. This is because actors do not share state, and actors process one message at a time. As a result, reasoning about concurrent actor programs is simplified. Another key property of actor systems is their hierarchical supervision for failure-management, which greatly influenced the design of fault-tolerant systems [8].

The actor model was later, in 2011, adapted for cloud programming in pioneering work on the *Virtual Actor* model (Figure 3b) in Microsoft's Orleans framework [18,13] (created at Microsoft Research). This influential work proposed three core distinctions: 1) actors are virtual, *i.e.,* they always exist, they are not created; 2) the framework manages the actor life-cycle, *i.e.,* actors are activated on-demand (and passivated when there is no demand), and transparently recover from failures; and 3) actor references are virtual (logical), *i.e.,* they can be created and serialized, and are always valid. Importantly, a virtual actor's *virtual identity* consists of a type/class tag and a key: `identity = type + key`. With this new identity, multiple actor instances (one for each key) can exist for the same type of actor, enabling a form of data parallelism. As a result, virtual actors are suitable for the cloud setting, and have consequently been adopted and further extended in the cloud and serverless realm [17,29,46,56,63,61,47].

Actor Model characterization:

– Actor-to-actor communication
– Stateful computation
– Dynamic topology: actors can create new actors; actors can create new connections

The Actor and Virtual Actor models share similarities with other models. The Active Object model [65] is closely related to the Actor Model, and is discussed in the next section. The Virtual Actor model bears much similarity with the *Entity* model developed by Pat Helland in 2007 [35]. In fact, many incarnations of the virtual actor model bear the name entity [17,46,56].

2.2 Active Objects

The Active Object model is an object-oriented concurrent programming model which evolved from the actor model and was developed in 1986 for the programming language ABCL/1 [65]. The model consists of *active objects* with a single thread of control and local state, which interact through *asynchronous method calls* [14]. These method calls usually return a *future* of the return value (implicitly or explicitly). Within the method, the active objects can suspend and wait (await) for a guard (*i.e.,* a conjunction of futures or boolean expressions) to be satisfied [32]. Important systems in this space include the ABCL/1 language [65], the ABS language [41,32], ASP/ProActive [21,10], Rebeca [60], JCoBox [59], and Encore [15], providing a spectrum of implementations and flavours.

The active object model can be understood as an integration of object oriented concepts with the actor model [14]. This allows for compositional object-oriented program constructions through the supported interface abstractions. Still, there are notable differences. Method calls to active objects are statically guaranteed to be executed. Whereas in the actor model, the actor's behavior and its implicit interface may change dynamically such that a message is ignored. Method calls in the active object model, moreover, are tightly integrated with *futures* [65], whereas futures are optional features in actor systems. One such example is future forwarding (avoiding creating nested futures), and future sharing [14]. Another example are nested blocking receives as available in some actors models![34]. In contrast, active objects process further method calls even when the called method was suspended. Overall, active objects have sophisticated mechanisms for process suspension and process scheduling beyond the run-to-completion model of actors.

We can understand the term actor-like, for the purpose of this survey, to encompass programming models that resemble the actor model and active object model. In fact, Orleans [18,13], Durable Functions' Entities [17], IBM KAR [63], Ray [53], Cloudburst [62], and Kalix [47], resemble the object oriented style in the active object model.

2.3 Dataflow Processing

Dataflow Processing has become the de-facto standard for processing large amounts of data. It defines computations as static, acyclic computational graphs. One of the most influential early systems was *MapReduce* [25], developed in part as a reaction to the complexity of managing computations over large data, dispersed across thousands of machines. The MapReduce framework enabled computations to be programmed as sequences of Map/Reduce steps, introducing two key innovations. Firstly, the framework fully managed *fault tolerance*. If any machine failed, it would recover and redo any lost computations. As a result of this, the system guaranteed *exactly-once processing:* meaning, that everything was processed and delivered exactly once, or, in other words, the system behaved observably equivalent to a failure-free execution [61,17]. Failures, in effect, became completely transparent to the user; a hallmark of dataflow processing systems. Providing exactly-once processing out-of-the-box was a great relief for the programmer because of how notoriously difficult it is to implement manually. Secondly, the computations were performed over data sharded by their keys. This enabled *data-parallelism* by distributing the computation such that data/events for the same keys were processed by the same computing nodes using local state.

Subsequent dataflow processing models have inherited much from MapReduce, such as the vertex centric model [50], Apache Spark [66], Apache Kafka [45], Apache Flink [20], Google Dataflow [5], and Naiad [54]. While these frameworks have improved in terms of performance as well as expressiveness, they still adhere to the same characteristics as MapReduce did: they provide transparent fault-tolerance (typically, a distributed two-phase commit); and computations occur over a per-key context.

Dataflow Processing characterization:

- Transparent fault-tolerance, exactly-once processing guarantees
- Scalability, data-parallelism, computations over a per-key context
- Static, directed acyclic computational graphs (DAGs)

2.4 Functions-as-a-Service (FaaS)

Serverless computing would come to offer even more convenience for developing scalable and distributed services: a fully-managed runtime that would execute Functions-as-a-Service (FaaS). These services are specified by two components [22]: 1) the *functions* which are to be executed; and 2) the types of events that *trigger* the functions. These functions are executed on a serverless platform: the code is run on-demand, the billing is only per-use, abstracting away any of the servers and infrastructure from the user [22]. The computing model no longer requires manual provisioning of virtual machines or servers (hence, “serverless”), instead, the serverless platform fully manages the execution.

The major cloud vendors started adopting this new trend [12], with AWS Lambda [6] introduced in 2014, and other similar services right after [49] (Azure Functions [51], IBM Cloud Functions [40], Google Cloud Functions [31]).

Functions-as-a-Service are typically restricted to stateless functions with limited composition beyond step-like workflows, which are commonly cited challenges with serverless computing [12,36,42]. In reaction to this, recent programming models have started to support stateful serverless applications with more flexible communication and composition primitives [53,29,4,62,17,63,61,56,47]. These models utilize abstractions closely resembling actors, active objects, entities, and virtual actors.

Functions-as-a-Service (Serverless) characterization:

– Stateless functions triggered by events
– Elastic scalability, code is run on-demand, billing is per-use
– Fully-managed runtime/platform

2.5 Actor-Like Serverless Computing

Actor-like programming models for serverless computing, sometimes also referred to as *stateful serverless*, are a combination of actor, dataflow, and serverless principles; they provide the flexibility of stateful computations with actor-to-actor communication; together with the fault-tolerance and data-parallel scalability of dataflow processing; with the serverless, fully-managed execution platform, run on-demand.

These combinations require the *virtualization* (decoupling) of the following components: function, compute, state, and event queue (mailbox) (see Figure 3b). Similarly to FaaS, the functions can be considered stateless: the function signature has both a stateful context and an event as parameters: `F: Ctx => Event => Unit`. The provided context `Ctx` gives the function access to state (`Ctx.state`) as well as the capabilities to interact with its environment (*e.g.,* `Ctx.send`). This decoupling, in turn, enables the on-demand scalability through replicating the functions and migrating state, and the transparent fault-tolerance through capturing any side-effects in terms of state and events from the context.

Systems in this space have adopted some of these new principles. The Virtual Actor model in Orleans [18,13], created at Microsoft Research, provided many of these features but lacked strong fault-tolerance guarantees such as exactly-once processing, or a fully-managed platform. Ray [53] and Cloudburst [62], incorporated actor principles with serverless (FaaS), forming decoupled (non-virtual) actors with automatic failure recovery providing at-least-once (and, tunable, at-most-once) guarantees. Another direction towards stateful functions, as seen on Flink [29,4], merged principles from dataflow processing with actors: scalable, data-parallel, stateful functions with function-to-function messaging and exactly-once processing guarantees.

Listing 1: A *bank account* entity that can get, deposit, withdraw, and transfer.

```
1  class Account(ctx: Context):
2    val balance = PersistentState[Int](ctx).withDefault(0)
3
4    def get(): Int =
5      balance.get()
6
7    def deposit(amount: Int): Unit =
8      balance.set(balance.get() + amount)
9
10   def withdraw(amount: Int): Unit =
11     balance.set(balance.get() − amount)
12
13   def transfer(amount: Int, to: String): Unit =
14     val otherAccount = EntityRef[Account](ctx).withKey(to)
15     if balance.get() > amount then
16       balance.set(balance.get() − amount)
17       otherAccount.deposit(amount)
```

More recently, proposals for stateful serverless programming models have emerged, merging actors, dataflow processing, and serverless, enabling the writing of stateful services with powerful compositional abstractions, while providing exactly-once processing guarantees. Notable systems include Microsoft's Durable Functions [17], IBM KAR [63], Portals [61], Stateful Entities [56], and Kalix [47].

An example entity representing a bank account is shown in Listing 1 in a style inspired by various systems [18,17,47,61,56,63,29,4]. It shows a bank account class that takes the runtime context as a parameter in its constructor (line 1). The runtime context is used to provide access to the side-effects of the entity: the state and the outgoing messages. The persisted state of the entity is explicitly declared on line 2, representing the account's `balance` with initial value `0`. The entity defines methods, for getting, depositing to, and withdrawing from the account. It also defines a method for *transferring* an amount from the account to another account. Creating a reference to the other account (receiving the transfer), is achieved through the `EntityRef` factory, which takes the runtime context as well as a parameter for the other accounts *key* (line 14). This way a reference can be created, and later used for depositing the transferred amount (line 17). This example highlights some of the features of entities: the persistent explicit state, and the per-key identity. Note that the balance is not shared between different keys, rather, every key has its own balance value. The example also highlights potential issues due to the asynchronous nature of the method invocations on these actors: concurrently issued withdraw invocations may cause an overdraft on the account. In order to overcome this, some transactional mechanisms or similar would be needed.

Characterization In general, we would characterize these actor-like serverless systems through five characteristics.

Actor-like serverless computing characterization:

1. Actor-like (Virtual Actors, Entities)
2. Data-parallel, keyed, scalable
3. Transparent fault-tolerance, exactly-once processing
4. Decoupled / externalized state, virtualization
5. Serverless execution, managed runtime

The execution model of serverless actor-like systems resemble the *isolated turn principle* [43] from actors with an additional *per-key* execution context: the execution of actors can be thought of an execution over isolated turns, in which a turn consists of an actor instance, identified by its *type* and *key*, consuming a message from its mailbox (mailboxes are disjoint over keys), executing the statements in the behavior, and possibly producing output messages and/or a state/behavior change. These turns are executed serially for a key, so that no two events are processed at the same time for a given actor type and key.

3 Challenges of Serverless Actors and Active Objects

Stateful serverless programming aims to provide several desirable properties which, in combination, are challenging without sacrificing the fault tolerance, flexibility, or performance. In particular, the following properties are essential: (a) serverless state management, enabling the provisioning of compute resources on demand; (b) fault tolerance with corresponding execution guarantees, providing the illusion of a failure-free execution in the presence of faulty computers and networks. In the following, we discuss the challenges of providing these properties in the context of actor and active object languages.

3.1 Serverless State Management

Serverless computing abstracts from the underlying computing infrastructure, providing load-based scaling of computing resources on demand. The automatic provisioning of compute resources affects the state management of the programming system. To illustrate some resulting challenges, consider the example shown in Listing 1. Suppose the `deposit` method of an `Account` is called by a different entity. When the `deposit` method is invoked on an entity reference, the corresponding entity instance must be activated on a suitable compute resource (*e.g.,* a virtual machine running in a data center). Note that we cannot assume that the entity instance is already loaded into the memory of a specific virtual machine. Instead, load-based scaling requires dynamically *loading/activate* a varying number of entity instances into a varying number of compute nodes.

Listing 2: An *account* entity with a *guard* on its withdraw method (replacing the withdraw method for the account entity from Listing 1).

```
1 class Account(ctx: Context):
2   ... // (see Listing 1)
3   val balance = PersistentState[Int](ctx).withDefault(0)
4   def withdraw(amount: Int): Unit =
5     await balance.get() >= amount
6     balance.set(balance.get() − amount)
```

Likewise, in case demand for requests to certain entities drop, it must be possible to *passivate* entity instances by persisting their state to stable storage and deallocating their memory. This means that all state of an entity must support *serialization* and the runtime system must be able to manage this state to support automatic passivation and activation.

Passivating the state of an entity is challenging in cases the programming model supports *guards* (*e.g.,* ABS [41]), or blocking receive statements (*e.g.,* Erlang [9]). To illustrate this, consider the `withdraw` method in Listing 2. On line 5, an ABS-style guard, `await balance.get() >= amount`, ensures that any call is *suspended* until the guard evaluates to true, ensuring a non-negative balance. This means that passivated entities might contain suspended calls. For this reason, the suspended calls and their *execution state* must be passivated as well, so that they subsequently can fully restore the suspended calls and be activated. Depending on the concrete programming model, execution states of suspended calls might consist of coroutines (*e.g.,* ABS [41], JCoBox [59]) or stackful continuations which are challenging to serialize (*e.g.,* due to embedded, non-portable memory addresses). In case the execution state of a suspended call or a suspended receive statement consists of just a continuation *closure* (*e.g.,* Scala Actors [34]), it is possible to support safe serialization using Spores [52] or other constructs that ensure the serializability of a closure's environment. For the reasons mentioned above, it is challenging to support the passivation and activation of actors and active objects in the context of serverless.

3.2 Fault Tolerance

Building distributed systems, *i.e.,* applications executing across multiple interconnected computers, requires handling faults such as machine crashes and unreliable or disconnected network connections. Consequently, distributed programming systems have long supported this through abstractions and constructs for fault handling. For example, Erlang's constructs for actor monitoring and supervision have been used successfully for building highly available distributed systems in the telecom industry [7]. Despite this, building distributed systems that completely mask failures has remained challenging, except for restricted computation patterns and system architectures (*e.g.,* Dataflow Processing).

The challenges of providing transparent fault tolerance in the context of actors and active objects are due to the combination and interplay of the following dimensions.

Stateful computation. To enable recovering from faults, mutable state must be distributed across multiple *replicas* running on different computers. These replicas must be synchronized whenever the state is updated. Furthermore, state updates must be *transactional:* recovering from faults must not inadvertently repeat a state update that was already applied.

Non-deterministic behavior. General concurrent programming models, such as active objects and actors, support writing non-deterministic programs. For example, when two concurrent active objects each call a method on a third active object, the two method calls are concurrent and thus their execution order is non-deterministic. In general, the behavior may also include non-deterministic computations, such as random number generation or the use of local time/-clocks. Supporting non-determinsitic behavior in fault-tolerant systems is challenging, since computations might have to be re-executed when recovering from faults. However, re-executing non-deterministic code can change the outcome of computations, thereby failing to provide the illusion of a failure-free execution. Supporting non-deterministic behavior thus requires the use of implementation techniques that do not make use of re-execution (such as rollback-recovery [28]), or logging all sources of non-determinism [30], making state management more complex and potentially increasing runtime overhead. This is further complicated by the dynamic topology of actor systems.

Interaction with external systems. In practice, distributed systems typically interact with various *external systems*, such as database management systems, distributed file systems, message queues. Requests submitted to external systems must not be tentative (and subject to potential rollback recovery); since such requests, in general, cannot be undone, they can only be submitted if the present system ensures that they are *never* going to be repeated, even during fault recovery.

Due to the above challenges, some stateful serverless programming systems trade flexibility for fault-tolerance guarantees. For example, instead of providing Exactly-Once Processing, some systems only provide At-Most-Once or At-Least-Once fault-tolerance guarantees. The latter significantly increases the complexity of the programming model, since events need to be either idempotent or deduplicated manually. On the other hand, At-Most-Once requires dealing with dropped events without support from the programming system. Although there are no fundamental limitations to execute the classic actor and active object model serverlessly, doing so comes at a tradeoff between the expressiveness, guarantees, performance, and cost of the model. The next section will explore various systems to highlight their variations among these dimensions.

4 Analysis of Actor-Like Serverless Systems

In this section we analyze the properties of a selection of systems at the intersection of actors, dataflow processing, and serverless. The analysis is structured around two questions. First, we analyze their specific properties with respect to the programming model. Second, we analyze their properties with respect to serverless execution. The purpose is to give an overview of similarities and dissimilarities between the programming models and implementations. The systems under survey are the following.

- The *Orleans* system [18,13], pioneering the virtual actor model.
- *Durable Functions' Entities* [17,16], virtual actors that can be used together with other abstractions such as Orchestrations and Activities within Microsoft's Durable Functions framework.
- *Apache Flink Stateful Functions* [29,4], an abstraction of virtual actor-like stateful functions running on Apache Flink, independently developed by different groups [29,4].
- *IBM KAR* [63], a polyglot scalable and fault-tolerant virtual actor system.
- *Kalix* [47], a serverless platform for deploying microservices consisting of entities, actions, and views.
- *Portals* and *Portals' Actors* [61], a research project and programming model which unifies the actor model with the dataflow processing model.
- *Ray* [53], a framework for scaling actor-like computational tasks, focused on reinforcement learning.
- *Cloudburst* [62], a stateful functions research project which leverages CRDT state for its execution.

Although not all systems fit the characterization from section 2.5, *e.g.,* through a lack of a fully-managed platform with per-use billing, they are included in this survey as they are closely related and provide valuable insights.

4.1 Programming Model

We analyze the programming models of the systems across three categories: actor style; communication; and state and computation. The analysis is reflected in Table 1.

Actor Style. The actor-like systems can be divided into two groups based on their style: virtual; and non-virtual (see Table 1). These two groups differ quite uniformly over the properties in our analysis.

Life-cycle. Virtual actors have a virtual life-cycle, they exist by definition rather than through creation. Non-virtual actors, in contrast, exist through creation.

Identity and References. Identifying a virtual actor is achieved through a *virtual identity. Virtual actor references* are constructed from identities using

Table 1: Programming model properties.

		Dyn. Topology (Int/Ext)			
	Actor Style	Application	Comm.	Ext. State	Fault-Transp.
Orleans	virtual	✗ / ✓	✓ / ✓	✓	✓-
Durable Functions	virtual	✗ / ✓	✓ / ✓	✓	✓
Flink StateFun	virtual	✗ / ✓	✓ / ✓	✓	✓
IBM KAR	virtual	✗ / ✓	✓ / ✓	✓	✓-
Kalix	virtual	✗ / ✓	✓ / ✓	✓	✓-
Portals	virtual	✗ / ✓	✗ / ✓	✓	✓
Ray	non-virtual	✓ / ✓	✓ / ✓	✓	✓-
Cloudburst	non-virtual	✓ / ✓	✓ / ✓	✓	✓-

factories. The virtual actor references are not strictly *always valid* when references can be forged from nonsensical user-provided strings [29,17,63,47]: if there is no corresponding actor definition for the provided string then this may cause a runtime error. Other systems ensure that references are valid either through compilation checks [61] or through reference factories constructed from existing actor types [18]. Non-virtual actors, in contrast, have references bound to lifetimes, which become invalidated if the referenced actor ceases to exist [53,62].

Actor Topology. The topology consists of the actors and how they are connected. We distinguish between *Application* and *Communication* topologies. The application topology consists of the actors, *i.e.,* if actors can be created and destroyed. The communication topology is the set of connections between actors, *i.e.,* if new connections can be formed, through exchanges of actor references. On another dimension, we also distinguish between *Internal* and *External* changes. *Internal* changes are triggered by the actors themselves, *e.g.,* an actor creating another actor; *External* changes are triggered by an outside force, *e.g.,* the driving application creating new actors or creating new connections (dynamic reconfiguration). The non-virtual actor systems are dynamic in all four cases (Table 1) [53,62]. The virtual actor systems, in contrast, have dynamic communication topologies, and partially dynamic application topologies (actors cannot create new actors, but the external force can do so) [18,17,29,63,61,47]. All of the systems have *first-class references.* The Portals system is an exception, it restricts actors from creating new connections dynamically through exchanging references; actor references are only usable by actors with the right capabilities, these capabilities are assigned statically through the actor definitions [61].

Communication. Actors communicate by exchanging messages either in the form of *message sends* or *method calls* (*cf.,* actors / active objects [43]) (see Table 2). Out of the selected systems, five had method-based communication, and three had message-based communication. The difference between the two is mostly syntactical, and some systems even provide both styles of interfaces for the actor communication [17]. For this reason we will not further distinguish between these interfaces; we will consider a *Method Invocation* to correspond

Table 2: Communication properties.

	Msg Ops	Msg Futures	Futures Retrieve Ops
Orleans	Send, Call, Reply	✓	Tunable
Durable Functions	Send, Reply-	✗	-
Flink StateFun	Send, Reply	✗	-
IBM KAR	Send, Call, TailCall, Reply	✓	Blocking
Kalix	Send, Call, Reply, Forward	-	-
Portals	Send, Call, Reply	✓	Non-blocking
Ray	Call, Reply	✓	Blocking
Cloudburst	Call, Reply	✓	Blocking

to a *Send* operation if it does not return a value, and to a *Call* operation if it returns a future of the return value. Similarly, we consider *Return* to correspond to the *Reply* operation.

Message Operations. All systems support the *Send* and *Reply* communication primitives. The exception, here, is Durable Functions [17], which can only reply to calls from *Orchestrations.* The *TailCall* primitive supported by IBM KAR, is for orchestrating guarantees across a chain of invocations: the previous call has to have finished/committed before the subsequent calls in the tail call are executed [63]. This can be used for higher fault-tolerance guarantees beyond what is provided. The *Forward* call in Kalix is a special operation which can forward a replyable message to another service [47].

State and Computation. The serverless computing paradigm is built on the decoupling of execution from side-effects and state. The programming models all provide explicit external state abstractions for this (Table 1), accessible through either a KV store-like interface [62,63], an object-store [53], or typed coarse-grained [18,47] or fine-grained [61,29,17] factories/annotations. Local variables, in contrast, do not survive a crash or migration, and are re-initialized upon activation of the actor.

Shared Memory. Although uncommon in actor-like abstractions, we found some instances of shared memory. Ray [53] has shared memory in the form of an external immutable first-writer-wins object store with distributed futures. Cloudburst [62] functions, on the other hand, share access to an eventually consistent key-value store, with additional mechanisms to enforce causal session consistency. Kalix [47] has replicated entity types backed by CRDTs which can be used as a form of highly available replicated shared state.

Concurrent Processing & Futures. Most systems provide futures for messaging and awaiting the completion of futures as a concurrency abstraction (see Table 2). An exception is Durable Functions [17] which does not provide futures for their Entities but for the Orchestrations. Similarly, Kalix supports futures (async effects) on Actions with operations reminiscent of chaining futures [47], it was unclear if this also applies to Entities, for this reason the entry was left blank. Flink Stateful Functions [29] provides futures for asynchronous operations, but not for asynchronous message calls expecting a reply. In the table, we only consider futures that are created from inter-actor messages / method invocations.

While the actor model is traditionally continuous and non-blocking (re-entrant) to ensure liveness [43], processing an *await* command on a future forces the system to choose between blocking or re-entrant execution. The Blocking mode blocks the execution of further events of the same key until the await command completes [63,53,62]. Whereas the Re-entrant mode interleaves the processing of subsequent events before the await command has completed, enabling increased concurrency and avoiding potential issues associated with blocking [29,61]. This choice is also a tunable setting in some systems [18]. Further, IBM KAR provide a mode for re-entrant execution for method calls on itself with the same session-id [63].

Failure Transparency. Failure transparency enables the developer to write applications without having to reason about certain failures (Table 1). The system is completely failure transparent if it provides exactly-once processing (marked as *ExO* in Table 3): the application does not have to manage anything related to failures [17,29,61]. If the system is partially failure transparent, that is, it provides at-most-once/at-least-once guarantees and some failure support (marked as *AMO*/*ALO*), then the application must manually perform certain actions for failure tolerance. For example, Orleans [18] and Ray [53] provide methods for asynchronously persisting and reading state, and it is up to the developer to implement it for the required guarantees. Whereas IBM KAR [63], Kalix [47], and Cloudburst [62] automatically retry function invocations (at-least-once), and the developer must ensure that the function is idempotent. For these reasons, exactly-once processing make programs significantly easier to write and reason about.

4.2 Serverless Execution

In this section we analyze properties related to the serverless execution and runtime. The analysis is structured around four categories: fault-tolerance; state management; scalability; and platform management (Table 3).

Fault Tolerance & Guarantees. Fault-tolerance guarantees are crucial for distributed systems, commonly expressed as one of the following: Exactly-Once (ExO), At-Most-Once (AMO), and At-Least-Once (ALO) (Table 3). Out of these, *Exactly-Once* is the strongest guarantee, guaranteeing that every event is delivered and processed exactly-once, implemented by three of the studied systems [17,29,61]. Exactly-once can also be regarded as observably failure-free, that is, the execution, and what is observed by the user, behaves as though it is failure-free. This greatly simplifies reasoning about distributed programs, eliminating the need for manual deduplication and the need to ensure that functions

Table 3: Serverless execution properties.

	Proc. Guarantees	State	Parallelism	Plaftform Mgmt
Orleans	AMO/ALO	Ext	Data	✗
Durable Functions	ExO	Embedded	Data	✓
Flink StateFun	ExO	Ext	Data	✓
IBM KAR	ALO	Ext	Data	✗
Kalix	ALO	Ext	Data	✓
Portals	ExO	Embedded	Data	✗
Ray	AMO/ALO	Ext	Task	✓
Cloudburst	ALO	Ext	Task	✗

are idempotent. At-Most-Once, in contrast, guarantees that every event is delivered and processed at most once (failed invocations are not retried); whereas, At-Least-Once, guarantees that every event is delivered and processed at least once (failed invocations are continually retried until success). The choice between the latter two may be tunable in some cases [18,53], whereas others only provide At-Least-Once semantics [62,63,47].

Failure-Recovery. Failure-recovery enables the system to effectively mask failures such as crashes or message loss from the observed execution. The exactly-once processing systems [17,16,29,19,61] use a checkpointing and recovery strategy [28]. This approach involves the system periodically creating checkpoints that comprise: 1) the actor state; and 2) the event queues. In the event of a failure, recovery proceeds by restarting the actors from the most recent checkpointed state and replaying events from the last checkpointed event queues. The challenge of establishing consistent checkpoints lies in taking causally-consistent snapshots of the system. This is done in Flink [19] and Portals [61] with a snapshotting protocol similar to the one presented by Chandy and Lamport [23]. In the Netherite runtime for Durable Functions, in contrast, a distributed snapshot is avoided by isolating the processing nodes and blocking events from being observed until they have been committed [16]. Other implementations that do not provide exactly-once processing guarantees restart from the latest checkpointed state, but may potentially replay events more than once (at-least-once), or drop events (at-most-once) [18,63,62,53,13,47].

State Management. The runtime necessarily manages the state in order to ensure strong fault-tolerance guarantees. This state is either external, *i.e.,* primarily on some external storage, cached locally for quicker access [18,63,53,62,47]; or embedded, *i.e.,* hosted in-full locally on the computing nodes, and persisted externally for durability (Table 3). There is a trade-off between the two. External state offers a higher decoupling, making it easier to scale up and down, as external state does not need to be migrated during reconfiguration. Embedded state, on the contrary, yields higher processing throughput and lower latency for stateful computations [17,16,61].

Scalability and Parallelism. All systems under discussion offer elastic scalability, enabling the runtime to flexibly scale up or down in response to demand. In this context, two forms of parallelism emerge (Table 3). The Non-Virtual Actors frameworks facilitate task-level parallelism. This is achieved by spawning new actors assigned to perform specific tasks and subsequently terminating them upon task completion. The Virtual Actors frameworks, on the other hand, support data-level parallelism, as a single actor definition is applied to many events but over different keys.

Platform Management. Platform management entails managing all aspects related to the runtime and the servers. That is, the user should only need to supply the stateful functions definitions and event triggers, and the platform should manage everything else, billing per-use. Of the selected systems, only Durable Functions [17] and Kalix [47] are purposefully built for that (Table 3). Concerning the other systems, both Flink (the Ververica Platform) and Ray (the Anyscale Platform) have hosted platforms available.

4.3 Related Work

This section concludes the analysis through summarizing other systems that were not included in the main analysis. *Data-Parallel Actors* [44] is a research project for writing data-parallel query systems, it is used to distribute otherwise non-distributed systems, such as databases and analytics systems. It does so by using an actor-like abstraction, which manages a partition of the wrapped system, for which the data-parallel actor must serve user-defined composable queries over the partitioned data, such as *Map*, *FlatMap*, *Scatter*, *Gather*; these queries apply to all partitions. *Crucial* [55] is a stateful serverless system for programming parallel applications. It executes on top of existing FaaS platforms, and provides a shared memory abstraction for fine-grained synchronization primitives and sharing larger state which can be used by the deployed functions. It executes on existing FaaS infrastructure with at-least-once guarantees. *Beldi* [67] enables writing fault-tolerant stateful serverless functions. It does so by providing primitives for consistently reading and writing from a shared memory, for transactional workloads with locking and transactions, and invoking other functions from within the function. It provides exactly-once processing guarantees using existing FaaS frameworks together with a strongly consistent storage provider. *A.M.B.R.O.S.I.A.* [30] is a system for transparent fault-tolerant non-deterministic applications. Ambrosia services are executed by actor-like abstractions called "immortals" using event sourcing and replay recovery for exactly-once guarantees, for which non-determinism are captured through impulses. For a replay to recover to a consistent state, it is important that the application adheres to a "weak language binding contract": from some state, any execution of inputs must result in an equivalent final state, outgoing events must be for the same destinations and in the same order, but may differ in content. *AEON* [57] is a scalable and elastic actor framework which guarantees strict serializability for events across actors using an ownership hierarchy. In follow-up work [58], the runtime is extended with programmable elasticity policies. The ABS model [41] has been used in the context of modelling distributed computing models, for example Spark Streaming [48], and Kubernetes deployments [64], making use of suspending guards for expressing the logic.

5 Research Directions

In this section we highlight some research directions with actor-like models for serverless computing.

Static guarantees, formal proofs. There are two main challenges with deploying serverless actor-like models: 1) ensuring that the user application is statically sound, and 2) ensuring that the runtime is fault-tolerant. For the first, a common error in user code is due to user-defined functions capturing non-serializable state from the environment [52]. This may cause errors which are hard to debug due to the distributed execution, and consequently crash. This and similar issues, such as well-typed channels, existing dependencies, may be caught statically at compile-time, and thus prevent the user from deploying the application. For the second challenge, it is important to provide formal proofs for the fault-tolerance guarantees. There has been some work in this area proving the failure-guarantees [63,17]; future research efforts should focus on formally proving more implementations, and providing new proof-techniques and frameworks. Beside formal proofs, it is also an important research direction to continue the exploration of fault-tolerance mechanisms used in this context.

End-to-end exactly-once processing with external systems. Distributed systems are rarely used in isolation. Especially the types of systems presented in this survey are likely to be used together with other services. For this reason it is important to ensure that certain guarantees, such as exactly-once processing, are provided end-to-end, across the external systems. The problem, however, is that the side-effects to external systems are typically not tracked by the system itself. In the context of dataflow processing, end-to-end guarantees are achieved through using transactional sinks, for example, connecting Flink [20] and Kafka [45]. Similarly, Portals suggests connecting external systems via atomic streams (transactional streams) for preserving the end-to-end guarantees [61]. More research in this area is warranted, both formal proofs and abstractions, as well as implementations and primitives for interacting with external systems.

New abstractions and primitives. First, the presented actor-like abstractions may not be suitable for all types of applications. This is especially problematic as the serverless paradigm restricts the application developer from implementing their own core abstractions. Examples include the *Orchestrations* in Durable Functions used for orchestrating workflows with the capability to perform blocking calls to and transactions across entities [17]. Another abstraction is a stateless function or actor as a way to distributed work in a task-parallel fashion [18,17,47]. Second, new communication and compositional primitives can be explored. One example here, are data-parallel operations over actors [44], which allow applying Map, FlatMap, Scatter, and Gather operations over all instances of a data-parallel actor. Third, libraries present an interesting opportunity to leverage the implementations and provide higher-level abstractions (*e.g.,* level four in Figure 1). Examples of this include numerous machine learning libraries on top of Ray [53], transactional libraries for Orleans [27] and Flink Stateful Functions [37], and streaming libraries [13]. Lastly, with the advancement of machine learning models, the importance of incorporating robust model serving capabilities within serverless frameworks has grown. Exploring this avenue presents interesting research opportunities.

6 Conclusion

This study has explored actor-like programming models within the context of serverless computing by providing a background, extracting distinctive features, outlining challenges, analyzing popular implementations within the field, concluding with suggestions for research directions.

In this paper, the development of actor-like models for serverless computing is traced back to the roots of Actors, Active Objects, Dataflow Processing, and Functions-as-a-Service, and presents a case for how merging principles of these three fields are expressed in the actor-like stateful serverless programming models as seen today: actor-like, virtual, data-parallel with decoupled state and transparent fault-tolerance in a serverless execution model. The core challenges of implementing serverless actors are the serverless state management and the fault tolerant execution. This includes managing the execution state of suspended objects, and providing fault-tolerance of non-deterministic functions in a dynamic environment. The analysis highlights similarities and differences between the implementations. One important difference is the choice between different fault-tolerance levels provided by the systems: at-least-once; at-most-once; exactly-once, as it affects many aspects surrounding the programming model and the implementation thereof. As future research directions, we suggest further exploring methods for static guarantees, end-to-end fault-tolerance, and new programming abstractions.

In summary, this survey of actor-like models in serverless computing has revealed a diverse and evolving field. Further work in this field can make future serverless systems more expressive and robust, resulting in more reliable and efficient software.

Acknowledgements This work was partially funded by Digital Futures, the Swedish Foundation for Strategic Research (under Grant No.: BD15-0006), Horizon Europe (SovereignEdge.Cognit under Grant No.: 101092711), as well as RISE AI.

References

1. Agha, G.: Concurrent object-oriented programming. Commun. ACM **33**(9), 125–141 (1990). https://doi.org/10.1145/83880.84528, https://doi.org/10.1145/83880.84528
2. Agha, G.A.: Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence). Ph.D. thesis, University of Michigan, USA (1985), http://hdl.handle.net/2027.42/160629
3. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. Series in Artificial Intelligence, The MIT Press, Cambridge, Massachusetts (1986)
4. Akhter, A., Fragkoulis, M., Katsifodimos, A.: Stateful functions as a service in action. Proc. VLDB Endow. **12**(12), 1890–1893 (2019). https://doi.org/10.14778/3352063.3352092, http://www.vldb.org/pvldb/vol12/p1890-akhter.pdf
5. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (2015). https://doi.org/10.14778/2824032.2824076, http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf
6. Amazon Web Services: AWS Lambda. https://aws.amazon.com/lambda/ (2023), accessed: 2023-03-20
7. Armstrong, J.: Erlang–a survey of the language and its industrial applications. In: Proc. INAP. vol. 96, pp. 16–18 (1996)
8. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, KTH Royal Institute of Technology, Stockholm, Sweden (2003), https://nbn-resolving.org/urn:nbn:se:kth:diva-3658
9. Armstrong, J., Virding, R., Williams, M.: Concurrent programming in ERLANG. Prentice Hall (1993)
10. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, composing, deploying for the grid. In: Cunha, J.C., Rana, O.F. (eds.) Grid Computing: Software Environments and Tools, pp. 205–229. Springer (2006). https://doi.org/10.1007/1-84628-339-6_9, https://doi.org/10.1007/1-84628-339-6_9
11. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 52:1–52:34 (2013). https://doi.org/10.1145/2501654.2501666, https://doi.org/10.1145/2501654.2501666
12. Baldini, I., Castro, P.C., Chang, K.S., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless computing: Current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer (2017). https://doi.org/10.1007/978-981-10-5026-8_1, https://doi.org/10.1007/978-981-10-5026-8_1
13. Bernstein, P., Bykov, S., Geller, A., Kliot, G., Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. Tech. Rep. MSR-TR-2014-41 (March 2014), https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

14. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017). https://doi.org/10.1145/3122848, https://doi.org/10.1145/3122848
15. Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E.B., Pun, K.I., Tarifa, S.L.T., Wrigstad, T., Yang, A.M.: Parallel objects for multicores: A glimpse at the parallel language encore. In: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures. Lecture Notes in Computer Science, vol. 9104, pp. 1–56. Springer (2015). https://doi.org/10.1007/978-3-319-18941-3_1, https://doi.org/10.1007/978-3-319-18941-3_1
16. Burckhardt, S., Chandramouli, B., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C., Zhu, X.: Netherite: Efficient execution of serverless workflows. Proc. VLDB Endow. **15**(8), 1591–1604 (2022), https://www.vldb.org/pvldb/vol15/p1591-burckhardt.pdf
17. Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S.: Durable functions: semantics for stateful serverless. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021). https://doi.org/10.1145/3485510
18. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: cloud computing for everyone. In: Chase, J.S., Abbadi, A.E. (eds.) ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011. p. 16. ACM (2011). https://doi.org/10.1145/2038916.2038932, https://doi.org/10.1145/2038916.2038932
19. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: Consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (2017). https://doi.org/10.14778/3137765.3137777, http://www.vldb.org/pvldb/vol10/p1718-carbone.pdf
20. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015), http://sites.computer.org/debull/A15dec/p28.pdf
21. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. pp. 123–134. ACM (2004). https://doi.org/10.1145/964001.964012, https://doi.org/10.1145/964001.964012
22. Castro, P.C., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Commun. ACM **62**(12), 44–54 (2019). https://doi.org/10.1145/3368454, https://doi.org/10.1145/3368454
23. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985). https://doi.org/10.1145/214451.214456, https://doi.org/10.1145/214451.214456
24. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: Boix, E.G., Haller, P., Ricci, A., Varela, C.A. (eds.) Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015. pp. 1–12. ACM (2015). https://doi.org/10.1145/2824815.2824816, https://doi.org/10.1145/2824815.2824816

25. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492, https://doi.org/10.1145/1327452.1327492
26. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer (2017). https://doi.org/10.1007/978-3-319-67425-4_12, https://doi.org/10.1007/978-3-319-67425-4_12
27. Eldeeb, T., Bernstein, P.: Transactions for distributed actors in the cloud. Tech. Rep. MSR-TR-2016-1001 (October 2016), https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/
28. Elnozahy, E.N., Alvisi, L., Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002). https://doi.org/10.1145/568522.568525, https://doi.org/10.1145/568522.568525
29. Foundation, T.A.S.: Apache flink stateful functions. https://nightlies.apache.org/flink/flink-statefun-docs-stable/ (2023), accessed on 2023-05-18
30. Goldstein, J., Abdelhamid, A.S., Barnett, M., Burckhardt, S., Chandramouli, B., Gehring, D., Lebeck, N., Meiklejohn, C., Minhas, U.F., Newton, R., Peshawaria, R., Zaccai, T., Zhang, I.: A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. Proc. VLDB Endow. **13**(5), 588–601 (2020). https://doi.org/10.14778/3377369.3377370, http://www.vldb.org/pvldb/vol13/p588-goldstein.pdf
31. Google Cloud: Google Cloud Functions. https://cloud.google.com/functions (2023), accessed: 2023-05-28
32. Hähnle, R.: The abstract behavioral specification language: A tutorial introduction. In: Giachino, E., Hähnle, R., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures. Lecture Notes in Computer Science, vol. 7866, pp. 1–37. Springer (2012). https://doi.org/10.1007/978-3-642-40615-7_1, https://doi.org/10.1007/978-3-642-40615-7_1
33. Haller, P.: On the integration of the actor model in mainstream technologies: the Scala perspective. In: Agha, G.A., Bordini, R.H., Marron, A., Ricci, A. (eds.) Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA. pp. 1–6. ACM (2012). https://doi.org/10.1145/2414639.2414641, https://doi.org/10.1145/2414639.2414641
34. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2-3), 202–220 (2009). https://doi.org/10.1016/j.tcs.2008.09.019, https://doi.org/10.1016/j.tcs.2008.09.019
35. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings. pp. 132–141. www.cidrdb.org (2007), http://cidrdb.org/cidr2007/papers/cidr07p15.pdf
36. Hellerstein, J.M., Faleiro, J.M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless computing: One step forward, two steps back. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings. www.cidrdb.org (2019), http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

37. de Heus, M., Psarakis, K., Fragkoulis, M., Katsifodimos, A.: Transactions across serverless functions leveraging stateful dataflows. Inf. Syst. **108**, 102015 (2022). https://doi.org/10.1016/j.is.2022.102015, https://doi.org/10.1016/j.is.2022.102015
38. Hewitt, C., Baker, H.G.: Laws for communicating parallel processes. In: Gilchrist, B. (ed.) Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977. pp. 987–992. North-Holland (1977)
39. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Nilsson, N.J. (ed.) Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973. pp. 235–245. William Kaufmann (1973), http://ijcai.org/Proceedings/73/Papers/027B.pdf
40. IBM Corp.: IBM Cloud Functions. https://cloud.ibm.com/functions/ (2020), accessed: 2023-05-28
41. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers. Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2010). https://doi.org/10.1007/978-3-642-25271-6_8, https://doi.org/10.1007/978-3-642-25271-6_8
42. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.J., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud programming simplified: A berkeley view on serverless computing. CoRR **abs/1902.03383** (2019), http://arxiv.org/abs/1902.03383
43. Koster, J.D., Cutsem, T.V., Meuter, W.D.: 43 years of actors: a taxonomy of actor models and their key properties. In: Clebsch, S., Desell, T., Haller, P., Ricci, A. (eds.) Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016. pp. 31–40. ACM (2016). https://doi.org/10.1145/3001886.3001890, https://doi.org/10.1145/3001886.3001890
44. Kraft, P., Kazhamiaka, F., Bailis, P., Zaharia, M.: Data-parallel actors: A programming model for scalable query serving systems. In: Phanishayee, A., Sekar, V. (eds.) 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022. pp. 1059–1074. USENIX Association (2022), https://www.usenix.org/conference/nsdi22/presentation/kraft
45. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB. vol. 11, pp. 1–7. Athens, Greece (2011)
46. Lightbend, Inc.: Akka. https://akka.io/ (2022), accessed: 2022-07-07
47. Lightbend, Inc.: Kalix. https://www.kalix.io/ (2023), accessed: 2023-05-18
48. Lin, J., Lee, M., Yu, I.C., Johnsen, E.B.: Modeling and simulation of spark streaming. In: Barolli, L., Takizawa, M., Enokido, T., Ogiela, M.R., Ogiela, L., Javaid, N. (eds.) 32nd IEEE International Conference on Advanced Information Networking and Applications, AINA 2018, Krakow, Poland, May 16-18, 2018. pp. 407–413. IEEE Computer Society (2018). https://doi.org/10.1109/AINA.2018.00068, https://doi.org/10.1109/AINA.2018.00068

49. Mampage, A., Karunasekera, S., Buyya, R.: A holistic view on resource management in serverless computing environments: Taxonomy and future directions. ACM Comput. Surv. **54**(11s), 222:1–222:36 (2022). https://doi.org/10.1145/3510412, https://doi.org/10.1145/3510412
50. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 25:1–25:39 (2015). https://doi.org/10.1145/2818185, https://doi.org/10.1145/2818185
51. Microsoft: Azure Functions. https://azure.microsoft.com/en-us/products/functions (2023), accessed: 2023-05-28
52. Miller, H., Haller, P., Odersky, M.: Spores: A type-based foundation for closures in the age of concurrency and distribution. In: Jones, R.E. (ed.) ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8586, pp. 308–333. Springer (2014). https://doi.org/10.1007/978-3-662-44202-9_13, https://doi.org/10.1007/978-3-662-44202-9_13
53. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: A distributed framework for emerging AI applications. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. pp. 561–577. USENIX Association (2018), https://www.usenix.org/conference/osdi18/presentation/nishihara
54. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. pp. 439–455. ACM (2013). https://doi.org/10.1145/2517349.2522738, https://doi.org/10.1145/2517349.2522738
55. Pons, D.B., Sutra, P., Artigas, M.S., París, G., López, P.G.: Stateful serverless computing with crucial. ACM Trans. Softw. Eng. Methodol. **31**(3), 39:1–39:38 (2022). https://doi.org/10.1145/3490386, https://doi.org/10.1145/3490386
56. Psarakis, K., Zorgdrager, W., Fragkoulis, M., Salvaneschi, G., Katsifodimos, A.: Stateful entities: Object-oriented cloud applications as distributed dataflows. In: Tanca, L., Luo, Q., Polese, G., Caruccio, L., Oriol, X., Firmani, D. (eds.) Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28. pp. 15–21. OpenProceedings.org (2024). https://doi.org/10.48786/edbt.2024.02, https://doi.org/10.48786/edbt.2024.02
57. Sang, B., Petri, G., Ardekani, M.S., Ravi, S., Eugster, P.: Programming scalable cloud services with AEON. In: Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016. p. 16. ACM (2016), http://dl.acm.org/citation.cfm?id=2988352
58. Sang, B., Roman, P., Eugster, P., Lu, H., Ravi, S., Petri, G.: PLASMA: programmable elasticity for stateful cloud computing applications. In: Bilas, A., Magoutis, K., Markatos, E.P., Kostic, D., Seltzer, M.I. (eds.) EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020. pp. 42:1–42:15. ACM (2020). https://doi.org/10.1145/3342195.3387553, https://doi.org/10.1145/3342195.3387553
59. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010.

Proceedings. Lecture Notes in Computer Science, vol. 6183, pp. 275–299. Springer (2010). https://doi.org/10.1007/978-3-642-14107-2_13, https://doi.org/10.1007/978-3-642-14107-2_13

60. Sirjani, M., de Boer, F.S., Movaghar-Rahimabadi, A.: Modular verification of a component-based actor language. J. Univers. Comput. Sci. **11**(10), 1695–1717 (2005). https://doi.org/10.3217/jucs-011-10-1695, https://doi.org/10.3217/jucs-011-10-1695
61. Spenger, J., Carbone, P., Haller, P.: Portals: An extension of dataflow streaming for stateful serverless. In: Scholliers, C., Singer, J. (eds.) Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022. pp. 153–171. ACM (2022). https://doi.org/10.1145/3563835.3567664
62. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Gonzalez, J., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow. **13**(11), 2438–2452 (2020), http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf
63. Tardieu, O., Grove, D., Bercea, G., Castro, P., Cwiklik, J., Epstein, E.A.: Reliable actors with retry orchestration. Proc. ACM Program. Lang. **7**(PLDI), 1293–1316 (2023). https://doi.org/10.1145/3591273, https://doi.org/10.1145/3591273
64. Turin, G., Borgarelli, A., Donetti, S., Damiani, F., Johnsen, E.B., Tarifa, S.L.T.: Predicting resource consumption of kubernetes container systems using resource models. J. Syst. Softw. **203**, 111750 (2023). https://doi.org/10.1016/j.jss.2023.111750, https://doi.org/10.1016/j.jss.2023.111750
65. Yonezawa, A., Briot, J., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Meyrowitz, N.K. (ed.) Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1986, Portland, Oregon, USA, Proceedings. pp. 258–268. ACM (1986). https://doi.org/10.1145/28697.28722, https://doi.org/10.1145/28697.28722
66. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Gribble, S.D., Katabi, D. (eds.) Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012. pp. 15–28. USENIX Association (2012), https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia
67. Zhang, H., Cardoza, A., Chen, P.B., Angel, S., Liu, V.: Fault-tolerant and transactional stateful serverless workflows. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. pp. 1187–1204. USENIX Association (2020), https://www.usenix.org/conference/osdi20/presentation/zhang-haoran