# Serverless State Management Systems

Tianyu Li
litianyu@csail.mit.edu
MIT CSAIL

Badrish Chandramouli
badrishc@microsoft.com
Microsoft Research

Sebastian Burckhardt
sburckha@microsoft.com
Microsoft Research

Samuel Madden
madden@csail.mit.edu
MIT CSAIL

## ABSTRACT

Modern cloud developers face many distributed systems complexities when building disaggregated applications from cloud building blocks. We propose a new class of cloud services, called Serverless State Management Systems (SSMS), that abstracts away these complexities and transparently manages fault-tolerance, deployment, and scaling of a logical cloud application on physical cloud resources. An SSMS, analogous to a DBMS, provides three important abstractions for disaggregated applications: 1) a logical application model, similar to relational algebra, that describes application semantics but abstracts away the deployment details, 2) strong resilient programming primitives, similar to ACID transactions, that simplifies fault-tolerant programming in the cloud, and 3) smart, cost-based optimization schemes that automates scheduling, placement, and other details, similar to a query optimizer. We present a preliminary design for SSMS and associated research challenges.

## 1 INTRODUCTION

The cloud is undergoing a major shift. Developers increasingly build *disaggregated* applications by composing managed cloud services (e.g., Amazon Aurora, Azure EventHubs) and fine-grained disaggregated resources (e.g., serverless compute, cloud storage). This new reality, broadly described as "serverless", claims to allow cloud developers to quickly assemble flexible applications that only consume resource as needed, and quickly scale to meet fluctuating demands. In reality, developers have a different experience: consider a workload from Amazon Prime Video that monitors a video/audio stream, converts the stream into frames and audio buffers, analyzes the stream for defects, and then send real-time notification for detected defects [8]. In a serverless architecture, developers ended up using AWS Step Function [3] to orchestrate unreliable compute units and S3 storage buckets for data sharing, which proved to be both expensive and unscalable. The Prime Video team had to rewrite their applications as a monolith that is scaled and deployed onto elastic containers for performance, and reported as much as 90% savings on infrastructure cost and better scalability. In short, the current disaggregated stack cannot automatically achieve the serverless vision. Distributed system expertise and careful engineering is often required to make disaggregation work.

What is needed is a way to *automate* such engineering efforts using strong, novel abstractions – the prime video team reported that most of their application logic and components remained unchanged, and much of the work was in deciding how to deploy, compose, and scale individual services. The right abstraction should be able to separate the *logical* cloud application, consisting of business and coordination logic, from the *physical* execution layer that resiliently deploys, and automate the execution layer to transparently perform the kind of adjustment detailed by the prime video team. We postulate that such an abstraction layer would require the following three cornerstones, drawing analogies from Database Management Systems (DBMSs) that have proven a successful and enduring abstraction for data processing workloads:

***Logical Application Model.*** DBMS users define their data and workload using logical schema and relational algebra, without reference to how data is laid out on disk or how queries are executed. This has allowed the same SQL query[1] to execute on vastly different undelrying engines (e.g., row vs. column stores, embedded engine vs. globally distributed service). In contrast, cloud applications today are written specifically to deployment paradigms (e.g., VMs, Kubernetes) or even implementations of these paradigms (e.g., AWS's management API). Cloud user should be able to write their programs once in a logical model of the cloud, which is then mapped onto a variety of backends automatically.

***Fault-Tolerance Primitives.*** ACID transactions are central to usability of DBMSs. Most users are able to enjoy strong guarantees without relying on application-level failure-handling logic or concurrency control. In the modern cloud, while many building blocks are fault-tolerant in isolation, there are few cross-service guarantees in a composed application; users must devise custom solutions to resiliently compose them, which is non-trivial. The ideal cloud abstraction provides transaction-like primitives for resilient composition. Users achieve fault-tolerance by building their multi-service applications with the primitives, and infrastructure providers take care of implementing the primitives.

***Automatic Cost-based Optimizations.*** Cost-based query optimizers are one of the most important components of a traditional DBMS, as they transform declarative user queries into highly efficient physical execution plans using a combination of search-based techniques and statistics about the data. Meanwhile, automatic optimization of cloud applications are still in its early days, limited to treating workloads as blackboxes and implementing only rudimentary knob-tuning or auto-scaling. The ideal cloud abstraction similarly makes dynamic adjustments to applications based on collected statistics and workload patterns, and intelligently navigates the performance-cost trade-off curve according to high-level user input (e.g., whether to prioritize performance or cost-savings).

Earlier solutions address these challenges to varying degrees, but none to our knowledge combines all three aspects into a viable solution. For example, cluster management tools such as Kubernetes[7] make some intelligent placement decisions and offer fault-tolerance in the form of backup and recovery, but provide little help for application-level tasks such as resilient orchestration of workflows; actor systems such as Microsoft Orleans [14] and Ray [32] provide strong and intuitive logical programming models, but require users to manually checkpoint state to storage for fault-tolerance.

In this paper, we propose a new cloud programming abstraction called the Serverless State Management System (SSMS) that addresses these challenges simultaneously. On a high level, SSMS

[1] not considering, for the sake of argument, evolution of SQL syntax over the years and discrepancies between various SQL dialects.

incorporates an expressive, actor-like programming interface, automatically manages state for fault-tolerance, and transparently applies application-level optimizations based on runtime metrics for cost savings and performance boosts. Our key insight is that strong fault-tolerance is the basis for simple programming abstractions and transparent optimizations. Specifically, given fault-tolerance across composed components, application designers don't need to worry about tricky issues like retries, recovery and logging, and the optimization layer can also rely on the fault tolerance layer to safely migrate/change the deployment without impacting application correctness. To summarize, we make the following contributions:

- We propose the SSMS abstraction that combines strong programming abstractions, transparent fault-tolerance, and cost-based optimizations to serve as the "narrow waist" between disaggregated cloud applications and cloud infrastructure.
- We sketch out the architecture of an initial SSMS platform and argue for its generality, performance and extensibility.
- We describe a number of transparent optimizations achievable in SSMS and provide preliminary evidence for their effectiveness.

## 2 OUR PROPOSAL

Users build SSMS applications using logical message-passing *automata* – arbitrary programs written as stateful message handlers, inspired by classical modeling of distributed systems using I/O automata [30]. For most developers, this experience will be similar to actors in Ray or Microsoft Orleans. In contrast to actors, however, SSMS automata are generally more coarse-grained, longer-living, and manipulate messages explicitly instead of through methods or RPCs. Like Orleans actors, SSMS automata are *virtual*. SSMS manages the location and resources for each automata, and users only refer to them by SSMS-managed unique ids. To interact with the system, users send messages to some automata, which may then kick off further internal interactions.

Importantly, SSMS focuses on automatically and resiliently managing the state of each automaton. In contrast, users of Orleans or Ray must manage custom checkpoints on either external storage or a distributed object store for resilience. We formalize the SSMS model using *fail-restart* automata. On a high-level, this means each SSMS automaton logically has two copies of its state — one volatile and one persistent. Failures result in the automata losing its volatile state and recovering from the (unavoidably stale) persistent state. Users supply application-specific checkpointing/recovery logic, but it is up to SSMS to manage checkpoints and control when to invoke them. To correctly orchestrate complex workflows across multiple automata, SSMS relies on the Composable Resilient Steps (CReSt) model [27]. In CReSt, each automaton receives some messages, updates its local state, and then sends some messages as an atomic, all-or-nothing unit that is recoverable after failure. We design SSMS to implement CReSt support by mediating all message and storage interactions, and expose it as a programming primitive to users. We require SSMS components to be written using CReSt *by default*, which guarantees that the resulting composed applications are resilient – external users are not be able to distinguish between an execution trace with failures and one without (except through possible performance degradations during recovery). SSMS mitigates

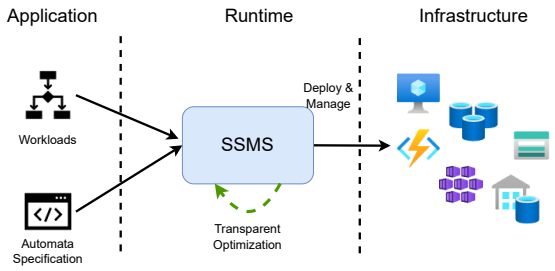



**Figure 1:** SSMS Overview

the resulting performance challenge with the performant DARQ system proposed in [27], discussed in detail in Section 3.

Resilience in SSMS enables more aggressive *online* reconfiguration and optimization of applications. Because application components can now be restarted after failure without impacting correctness of the entire system, SSMS can also restart them for performance improvements, cost savings, or both. Examples include switching between serverless and provisioned backends for automata, shutting down compute capacity for inactive automata, co-locating automata that frequently communicate, or combining multiple fine-grained automata into one equivalent "super" automata for batching opportunities. SSMS is uniquely positioned to implement these optimizations, as they are often per-automaton decisions that are too fine-grained for manual decision, and based on dynamically changing messaging patterns that a traditional static scheduler may not be aware of a priori.

Putting it together, we propose for SSMS to be the mediating runtime layer between disaggregated cloud applications and cloud infrastructure, as shown in Figure 1. SSMS exposes a virtual automata interface to users and transparently deploys automata to underlying cloud infrastructure. Unlike previous work, SSMS integrates state management of automata and controls when and where to persist state. This allows SSMS to support strong resilience guarantees and effectively hide failure recovery from users. SSMS takes advantage of this to implement a number of automatic online optimizations for both performance and cost in a cloud environment.

## 3 DESIGNING SSMS V0.1

We now sketch a preliminary design for an SSMS Figure 2, building on the DARQ system proposed by [27]. We start by introducing DARQ, and then outline how SSMS can be implemented as a management layer on top of a DARQ cluster.

### 3.1 Background: DARQ

DARQ is a cloud-native storage service with built-in CReSt support. Each DARQ instance is essentially a log that encapsulates the state of a fail-restart automaton. Users of DARQ write applications as stateful message handlers that update their local state as reactions to external messages, and (optionally) send out additional messages to other automata. DARQ records messages in the log, and then automatically checkpoints/recovers the log and reliably delivers messages and exactly-once to enforce CReSt semantics. Importantly, DARQ is designed to support compute-storage separation; DARQ message handlers run on logically separate, ephemeral compute nodes and rely solely on CReSt for resilience, and are only

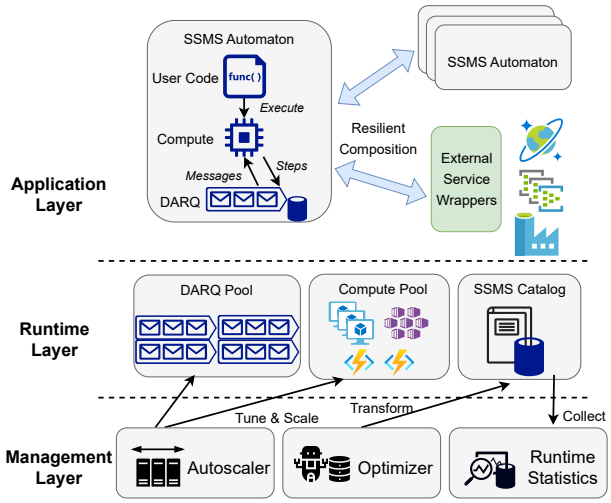



**Figure 2:** SSMS v0.1 Architecture

co-located with storage for performance where appropriate. Developers can treat DARQ as a write-ahead log, using self-messages (similar to write-ahead log entries) to encode state updates and submitting them as part of a step to DARQ. When a failure occurs, the recovered message handler then consumes all self-messages to replay and reconstruct its in-memory state before resuming operations. Because CReSt is atomic, if some external effect of a step (i.e., outgoing DARQ messages) survives the failure, so must the self-messages that allow reconstruction of the state as of the step.

## 3.2 Automata Implementation

Each SSMS automaton is implemented as an ephemeral compute node attached to a DARQ, and user code is written as DARQ message handlers. Non-DARQ services compose with SSMS using CReSt-compatible wrappers – for example, a SQL database can emulate CReSt semantics using the well-known transactional outbox pattern [11]. External services register such wrappers with SSMS, and the rest of the system is able to communicate with them as if they are other automata. SSMS does not manage such external services and it is up to the users to correctly register and operate them. At any given time, SSMS maintains a pool of DARQ instances and a pool of ephemeral compute nodes. When the user requests an automata, SSMS serves the request by picking one of each and loading user code into the compute node. Both pools are *heterogeneous* and consist of instances that occupy different points in the cost-performance trade-off space (e.g., fast but expensive dedicated VMs vs. slow but cheap serverless functions).

## 3.3 SSMS Catalog

To manage the various DARQ instances, SSMS implements a catalog service. Because catalog content is unlikely to change rapidly, we implement it using reliable cloud data stores (e.g., a cloud-native SQL database). Logically, the catalog stores 3 tables: 1) a table mapping automata types to user definitions (i.e., code files), 2) a list of available compute and DARQ nodes, their types, how to reach them, and their current load, and 3) an currently-active automata table, mapping automaton ID to automaton type, compute backend, and DARQ backend. Each DARQ uses the catalog to translate logical IDs in user code to physical addresses for message routing and caches this information for performance. We adopt a *lazy* protocol for handling outdated cache entries. First, each message between DARQs are explicitly tagged with the automaton ID they are intended for, and each DARQ instance checks whether they are currently serving the intended automaton upon receiving a message. A DARQ instance will then be able to detect stale cached entries when it receives a message for an automaton it does not currently service, and send a signal for invalidation.

The catalog also requires some compute capability to perform metadata operations; this includes instantiation of a new automaton, deallocation of an an inactive one, and recovery (from failure or re-deployment). The key challenge here is again fault-tolerance, as even though the catalog content is the source of truth, metadata operations has non-atomic side effects (e.g., loading user code into a compute node). Failure during a metadata operation may cause resource leakage or other anomalies. To overcome this, each metadata operation is written as a *resilient workflow*, achievable with DARQs as shown in [27], ensuring that all necessary steps of an operation completes regardless of metadata worker failure.

## 3.4 SSMS Management Layer

Finally, a management layer is responsible for tuning, scaling, and optimizing an SSMS layer at runtime in the background. The SSMS management layer can be broadly categorized into 3 components as shown in Figure 2, an autoscaler that controls the size and composition of DARQ and compute pools, an optimizer that makes placement and migration decisions, and a monitoring component that collects runtime statistics to support the first two components.

As mentioned, we envision for SSMS's optimization to be *cost-based* and *online*, inspired by both adaptive query processing [13] and more recent work in self-tuning and self-driving databases [31, 33, 40]. On a high-level, we propose to implemenet a cost model that, given an SSMS deployment and a predicted workload, can forecast the performance and cost of the system in the near future. The cost model relies both on hard-coded rules and observation of active deployments. Users specify a custom weighting function (e.g., minimize cost so long as performance does not drop below some level) that rolls multiple objectives into a single scalar optimization target. SSMS will attempt to launch new automata in "safe" configuration depending on the target metric (e.g., over provision if users value performance), and then incrementaly *refine* the deployment as it collects information about the behavior of deployed automata.

The SSMS autoscaler is responsible for managing provisioned resources in the cluster. Even though SSMS can scale out quickly using serverless offerings such as FaaS, much prior work has demonstrated that provisioned resources beat serverless functions in both performance and cost if utilization is high. The SSMS autoscaler is therefore required to detect stable parts of the workload, and launching the approrpiate amount of provisioned resources to support that workload. The primary challenge here is that the optimal provisioning depends is conditioned on the optimizer – a perfectly valid provisioning would appear under-utilized if the optimizer does not yet place much work on provisioned VMs. As a stopgap

solution, we expect to invoke the autoscaler much less frequently than the optimizer, so the optimizer has time to converge on a good configuration before the infrastructure shifts under it. For a more complete and sophisticated solutions, however, SSMS must resort to learning-based approaches similar to BRAD [25]. We leave a more detailed discussion of the solution for future work.

# 4 OPTIMIZING SSMS APPLICATIONS

In this section, we present a (non-exhaustive) list of possible optimizations in the SSMS. For each optimization, we briefly discuss their implementation and illustrate how much benefit they can bring using microbenchmarks.

## 4.1 Optimization through Placement

The most basic and broadest class of optimizations for SSMS is *placement*, which selects the DARQ instance and compute instance for each automaton. Every placement decision occupies a different point in the cost-performance trade-off space, and SSMS navigates the application cost-performance curve by varying placement choices. For DARQs, SSMS can choose between high-performance, low-latency replicated DARQ, mid-tier instances with dedicated servers but various cheaper cloud storage as backend, or cheap instances that exist solely on cloud storage and must be loaded into the compute node when used. For compute nodes, SSMS may utilize spare compute capacity on dedicated DARQ machines, dedicated compute-intensive VMs, or serverless functions that are launched on-demand. Within each of these categories, there may be further variations based on machine types, pricing (e.g., spot instances), or specialized hardware available (e.g., GPUs). The following class of optimizations can all be captured as placement decisions:

***Scale Up/Down.*** To scale-up or down an automaton, SSMS chooses more powerful compute/storage to deploy it onto. Note also that scaling of compute and storage is separate, and it is possible to redeploy a I/O heavy automaton to faster storage without changing its compute instance, and vice-versa.

***Automatic Deactivation.*** Automatic deactivation is a staple of any serverless offering, and allows users to pay nothing or very little for inactive deployments in exchange for a slower spin up time. In SSMS, if an automaton is inactive, its compute node maybe deallocated transparently, and only replaced with an on-demand FaaS when a request arrives. Additionally, if an automaton has no unconsumed (self or incoming)messages, it becomes essentially stateless and can be unassigned from a DARQ.

***Co-location.*** Physical co-location of disaggregated resources is a key design principle for performance of serverless computation [37]. SSMS is able to capture this by having multiple compute instances that are co-located with DARQ instances, or multiple DARQ instances that are co-located on the same VM/storage backend. Messages between such co-located instances are logical and can be implemented with highly efficient local operations.

## 4.2 Multiplexing DARQs

As shown in [27], each DARQ instance supports up to 750k steps per second on fast storage; to saturate one DARQ with a single sequential compute node, each compute steps cannot spend more than a few microseconds. It would be ideal to *multiplex* automata onto a single physical DARQ to increase utilization. However, multiple compute nodes may submit parallel steps that conflict with each other. The original DARQ system sidesteps this by enforcing that only one compute node is allowed to connect to DARQ at any given time. To support multiplexing in DARQ, we modify our earlier safeguards to allow for *partitions* – DARQ users explicitly tag each message with an additional partition ID, and DARQ allows multiple compute nodes as long as they work on disjoint partitions. By definition, partitions do not share state and cannot conflict with each other. Every partition on a physical DARQ can then be considered a *logical* DARQ that has the same guarantees and semantics of a DARQ, but shares resources with its peers on a physical DARQ. For simplicity, we enforce that logical DARQs are the smallest unit of operation in SSMS – each automaton corresponds to one logical DARQ, and multiplexing beyond this level must be done in user application. The SSMS layer hides logical multiplexing of DARQ from users, as users only refer to DARQs using logical IDs.

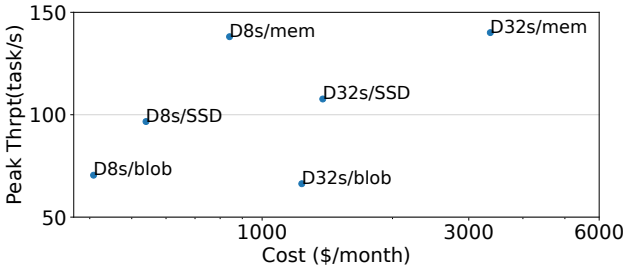
For added runtime flexibility, DARQ must also support dynamic movement of logical DARQs. Logically, a migration is a step that consumes all previously unconsumed messages of a logical DARQ, and copies them as outgoing messages to their new location. Migration is complete when all such messages have been received. The primary challenge here is guarding against concurrent steps and new messages during a migration. To address this, the migration must only begin after the local DARQ has been configured to reject requests against the migration target. This means that a logical DARQ appears as temporarily unavailable during migration, which may cause some messaging delays, but is ultimately safe as long as the migration eventually completes. DARQ has built-in epoch protection capabilities to support this [29], and our scheme is similar to earlier systems that implements dynamic key migration [26, 28].

## 4.3 Transparent Replica and Stand-Bys

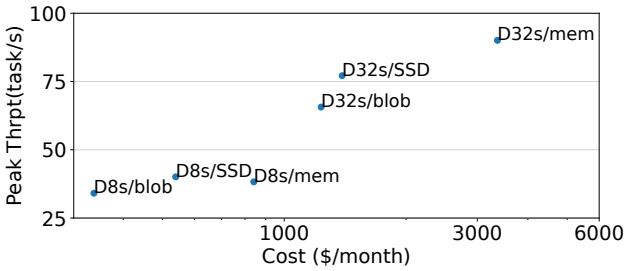
Lastly, DARQ applications can be transparently replicated for high availability. To do so, SSMS allocates more than one compute node to an SSMS automaton, with one being designated the primary. Because SSMS controls messaging, it can enforce primary/backup semantics by only streaming external messages to the recognized primary and only allowing the recognized primary to submit steps. Backup compute nodes receive self-message for replaying, which allows them to build up the same local state as the primary, in classical replicated state machine fashion [34]. To switch to a backup, SSMS merely needs to make a local decision to re-route messages and step priviledge to a backup node. Note here that this only requires client-supplied recovery logic, and clients need not implement additional mechanisms such as consensus or heartbeat. When SSMS detects that certain compute nodes are failing more than others, it may decide to transparently spin up / increase replicas. Alternatively, SSMS can also spin up temporary replica before a scehduled automata relocation, or in response to pre-emption on a spot instance, to reduce impact on operations.

## 4.4 Preliminary Evaluation

We now show the effectiveness of these optimization techniques using experiments. We conduct our experiments on the Azure public cloud using a simple application that repeatedly computes pi to




**(a)** c=1m




**(b)** c=10m

**Figure 3: SSMS Cost-Performance Trade-Off Space**

some precision (compute scale factor $c$). Each completed task then enqueues a new computation task to a different DARQ to continue the computation. We deploy this application on various compute node types, DARQ backends and (manual) placement configurations to showcase the potential SSMS might bring in automatically optimizing these decisions.

***Cost-Performance Trade-off.*** We deploy the simple application on two VM sizes: D32s v3 and D8s v3 [9] and three DARQ storage backends: hot-replicated (simulated with volatile memory), managed cloud SSD [4], and Azure storage blob [6]. Each computation step computes pi in parallel to take full advantage of VM compute capacity. We then compute the monthly cost of such deployments using Azure's current pricing information (assuming 3-way replication for simulated replication backend). We report the result of in Figure 3. As seen, the options chosen span a large area in the cost-performance trade-off space. The sweet spot for each application is also highly variable – note that in the first non compute-intensive scenario, upgrading to faster storage are much more cost-efficient than upgrading VMs, whereas in the compute-intensive scenario, fast VM matters more than fast storage. Note here also that we calculated the price of blob-based configurations assuming a 10% utilization rate, as blobs charge users per request. Assuming peak utilization throughout the month, blobs turned out to be the most expensive, but quickly become cost-efficient with low utilization.

***Co-location.*** We now show the impact of co-location on performance. We run the same workload as before, but when co-locating DARQ we no longer force the next task to enqueue onto another DARQ (co-located DARQ-only), and we further take advatange of DARQ's co-located compute API to elide communication overhead between DARQ and the compute node when noted (co-located both). We show our results in Figure 4, which illustrates that co-location can have orders-of-magnitude of impact on overall performance.

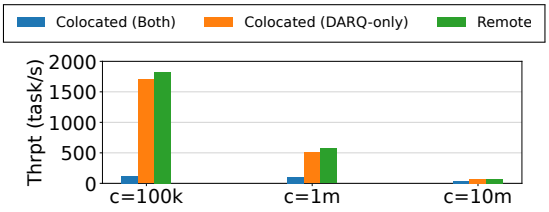



**Figure 4: Benefits of Co-location in Different Scenarios**

However, it is again highly dependent on the application – if the application is compute-intensive, co-location has some benefits, but is much more limited than I/O intensive ones.

Overall, these experiments show that by picking the right configuration, SSMS can potentially have large performance improvements or cost-savings. Such decisions are sufficiently complicated and sensitive to changes in the workload, humans cannot be expected to make case-by-case manual decisions. Therefore, it is both beneficial and necessary to have an automated solution like SSMS.

## 5 RELATED WORK

***Orchestration Systems.*** Traditionally, cloud providers offer resources in coarse-grained bundles as statically provisioned VMs.Most modern users manage VMs through higher-level orchestration systems such as Kubernetes [7], Amazon ECS [1], or Apache Mesos [20]. These systems typically employ some intelligent cluster scheduling algorithm to place workload [22], but fundamentally exposes a low-level machine-level abstraction (i.e., raw VMs or containers). Recent work has also proposed to extend this paradigm to multiple clouds [38]. Compared to SSMS, such orchestration systems better supports compatbility with earlier VM-based cloud software, but has limited ability to optimize applications as low-level blackboxes.

***Serverless Frameworks.*** Much of the prior work on disaggregated cloud applications focuses on the paradigm of "serverless", particularly the Function-as-a-Service paradigm (FaaS) [2, 5]. Despite the promises of simplified and flexible cloud programming [24], FaaS is considered flawed and cannot support efficient data processing, state management, or complex coordination [19]. Researchers have proposed various solutions to this by either orchestrating fault-tolerant workflows across FaaS instances [3, 10, 12], or improving stateful programming support [17, 23, 37, 42]. In contrast, SSMS is designed with statefulness and fault-tolerance as a first-class concern, and also incorporates provisioned resources.

***Streaming and Actor Frameworks.*** SSMS is closest to actor systems such as Ray [32], Orleans [14] in its programming model and abstraction. However, SSMS is based on more classical distributed systems modeling of I/O automata [30]. Most actor frameworks also do not provide strong fault-tolerance guarantees, whereas SSMS provides transparent and resilient state management using the CReSt primitive and DARQ system. In this respect, SSMS is similar to previous proposals of actor-oriented database systems [15], but we engineer SSMS as an integrated system rather than a separate database backend. SSMS also share many characteristics with stream processing systems such as Trill [16] and Kafka Streams [41]. Most notably, many modern stream processing systems also provide strong exactly-once guarantees and take advantage of the guarantee to dynamically optimize for execution [21]. The main difference is

that SSMS targets a more general cloud workload beyond streaming, and also explicitly optimizes for cost in addition to performance.

***Cloud Operating Systems.*** Some recent work proposes radical re-engieering of the current cloud stack. Several proposals exist for building a new operating systems layer over mutliple machines to hide distributed complexity in the data center [35, 39]. The most radical of these approaches argue that cloud applications can be built on top of a high-performance distributed SQL database [36]. SSMS, in contrast, is an application-facing system that operates above the usual OS layer. SSMS is closest to recent proposals from Google to write applications as logical monoliths but physically distribute them with an automated runtime layer [18]; unlike this proposal, SSMS provides resilience as part of the guarantee, which enables many of our optimization techniques.

## 6 CONCLUSION

We proposed the Serverless State Management System (SSMS), a cloud abstraction layer that combines a logical application model, strong fault-tolerant primitives, and transparent runtime optmizations. We sketched a prototype design of one such SSMS that is able to provide transparently resilient state management for an actor-like interface, and propose a variety of automatic optimizations possible under this architecture. We believe SSMS can serve as the "narrow waist" between user applications and cloud infrastructure and unlock new potentials for the cloud.

## REFERENCES


[1] Amazon Elastic Container Service. https://aws.amazon.com/ecs/, 2023.

[2] AWS Lambda. https://aws.amazon.com/pm/lambda, 2023.

[3] AWS Step Functions. https://aws.amazon.com/pm/step-functions/, 2023.

[4] Azure Disks. https://azure.microsoft.com/en-us/products/storage/disks/, 2023.

[5] Azure Functions. https://azure.microsoft.com/en-us/products/functions, 2023.

[6] Overview of Azure Page Blobs. https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-pageblob-overview, 2023.

[7] Production-Grade Container Orchestration. https://kubernetes.io/, 2023.

[8] Scaling up the Prime Video audio/video monitoring serbice and reducing costs by 90%. https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, 2023.

[9] Sizes for Virtual Machines in Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes, 2023.

[10] Temporal. https://temporal.io/, 2023.

[11] Transactional Outbox Pattern. https://microservices.io/patterns/data/transactional-outbox.html, 2023.

[12] What are Durable Functions? https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, 2023.

[13] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Conference on Innovative Data Systems Research*, 2005.

[14] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[15] P. Bernstein, M. Dashti, T. Kiefer, and D. Maier. Indexing in an actor-oriented database. In *Conference on Innovative Database Research (CIDR)*, January 2017.

[16] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, dec 2014.

[17] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano. New directions in cloud programming. *ArXiv*, abs/2101.01159, 2021.

[18] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, and A. Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 110–117, New York, NY, USA, 2023. Association for Computing Machinery.

[19] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *ArXiv*, abs/1812.03651, 2018.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, Mar. 2011. USENIX Association.

[21] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4), mar 2014.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 261–276, New York, NY, USA, 2009. Association for Computing Machinery.

[23] Z. Jia and E. Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[24] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[25] T. Kraska, T. Li, S. Madden, M. Markakis, A. Ngom, Z. Wu, and G. X. Yu. Check out the big brain on brad: Simplifying cloud data processing with learned automated data meshes. *Proc. VLDB Endow.*, 2023.

[26] C. S. Kulkarni, B. Chandramouli, and R. Stutsman. Achieving high throughput and elasticity in a larger-than-memory store. *Proc. VLDB Endow.*, 14:1427–1440, 2020.

[27] T. Li, B. Chandramouli, S. Burckhardt, and S. Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[28] T. Li, B. Chandramouli, J. M. Faleiro, S. Madden, and D. Kossmann. Asynchronous prefix recoverability for fast distributed stores. *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[29] T. Li, B. Chandramouli, and S. Madden. Performant almost-latch-free data structures using epoch protection. *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022.

[30] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 137–151, New York, NY, USA, 1987. Association for Computing Machinery.

[31] L. Ma, W. Zhang, J. Jiao, W. Wang, M. Butrovich, W. S. Lim, P. Menon, and A. Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1248–1261, New York, NY, USA, 2021. Association for Computing Machinery.

[32] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, Oct. 2018. USENIX Association.

[33] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *Conference on Innovative Data Systems Research*, 2017.

[34] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[35] M. Schwarzkopf, M. P. Grosvenor, and S. Hand. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[36] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia. Dbos: A dbms-oriented operating system. *Proc. VLDB Endow.*, 15(1):21–30, sep 2021.

[37] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[38] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[39] A. Szekely. $\sigma$ os: Elastic realms for multi-tenant cloud computing, September 2022. Available at https://dspace.mit.edu/handle/1721.1/147373.

[40] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[41] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2602–2613, New York, NY, USA, 2021. Association for Computing Machinery.

[42] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, Nov. 2020.