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 systems, 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, as developers increasingly build *disaggregated* applications by composing managed cloud services (e.g., Amazon Aurora [1], Azure EventHubs [7]) and fine-grained disaggregated resources (e.g., AWS Lambda [3], Azure Blob Storage [8]). This new reality, broadly described as "serverless", claims to allow cloud developers to quickly assemble flexible applications that only consume resources 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, divides them into chunks and decodes them, analyzes chunks for defects, and then sends real-time notification for detected defects [10]. In a serverless architecture, developers ended up using S3 storage buckets for data sharing, and AWS Step Function [4] to orchestrate unreliable compute units. The Prime Video team reports that this architecture is both expensive and unscalable, prompting a rewrite of their application by migrating components onto elastic containers with custom orchestration logic and intermediate result storage; this reportedly lead to as much as 90% savings on infrastructure cost and better scalability. Essentially, the current disaggregation stack often fails to achieve the promise of serverless.

Interestingly, the Prime Video team reports that the high-level architecture of their solution remained the same across the rewrite – the same components and workflow now just deploy onto different backends, with custom scaling logic. This naturally leads to the question of whether one can *automate* such engineering efforts to make disaggregation work. Doing so requires novel abstractions that separate the *logical* cloud application, consisting of business and coordination logic, from the *physical* execution layer that resiliently deploys and manages the application. Such an abstraction layer allows the cloud to transparently swap out infrastructure without requiring the kind of rewrite effort the Prime Video team underwent. There have been various attempts at this: cluster management tools such as Kubernetes[9] make intelligent placement decisions and offer automatic crash recovery for coarse-grained execution units (i.e., containers), but provide little help for application-level tasks such as resilient orchestration of workflows; actor systems such as Microsoft Orleans [15] and Ray [35] provide strong and intuitive logical programming models, but require users to manage state (e.g., checkpointing) manually. This paper is a thought experiment about designing the right type of cloud programming abstraction from first principles rather than from existing implementations. We start by postulating the key requirements for such an abstraction. Drawing analogies from the (widely successful) abstraction of Database Management Systems (DBMSs), we propose the following cornerstones for the ideal abstraction:

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 underlying 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 users 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 help DBMS users enjoy strong guarantees without relying on application-level failure-handling logic or concurrency control. In the modern cloud, there are few cross-service guarantees, even when individual services are fault-tolerant in isolation. Consequently, users must devise complex solutions to resiliently compose them (e.g., with systems such as Temporal [12] and ExoFlow [48]). Ideally, users achieve fault-tolerance solely by building on top of resilient primitives supplied by the underlying cloud, and infrastructure providers are responsible for implementing the primitives correctly and efficiently.

Automatic Cost-based Optimizations. Cost-based query optimizers in DBMSs transform declarative user queries into highly efficient physical execution plans. Meanwhile, automatic optimization of cloud applications is still in its early days, limited to treating workloads as black-boxes and implementing only rudimentary knob-tuning or auto-scaling. The ideal cloud abstraction should employ DBMS-style optimization to intelligently navigate the the performance-cost trade-off curve, using collected statistics and workload patterns, rather than heuristics or manual intervention.

Earlier solutions address these challenges to varying degrees, but none, to our knowledge, combines all three aspects into a complete solution. We coin the term Serverless State Management System (SSMS) for a new class of cloud systems that aim to address these

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

challenges simultaneously. The obvious question, then, is whether the challenges in building such an abstraction can be solved. The rest of this paper attempts to argue that this abstraction is realistic by sketching out a preliminary design for one such system, combining recent work in actor-based programming, composable fault-tolerant primitives, and ML-enhanced automatic system optimization. Our key insight is that strong fault-tolerance is the basis for simple programming abstractions and transparent optimizations. Specifically, strong fault-tolerance allows the programming model to abstract away common distributed system mechanisms, such as timeouts, retries, and logging; the optimization layer can also rely on fault-tolerance to dynamically migrate and reschedule components without impacting application correctness. The resulting SSMS design 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. 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 several optimizations already achievable in our prototype SSMS design and provide preliminary evidence for their effectiveness.

2 OUR PROPOSAL FOR SSMS V0.1

As mentioned, we propose SSMS as the mediating runtime layer between disaggregated cloud applications and cloud infrastructure (Figure 1). 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 [33]. For most developers, this experience will be similar to the actor model, popularized by systems such as Ray or Microsoft Orleans. Compared to actors, however, SSMS automata are not constrained to sequential/in-order processing like many actor models. Hence, users have the freedom to implement more coarse-grained automata, such as entire database partitions. SSMS automata are *virtual* [15], meaning that SSMS manages the location and resources for each automata, and users interact with them only through SSMS-managed logical IDs.

Importantly, SSMS automatically and resiliently manages the state of each automaton, modeling failure and recovery under a *fail-restart* assumption. Under this model, each automaton is able to checkpoint its volatile state to external persistent storage, and is guaranteed to restart cleanly from previously checkpointed state in bounded time after failure. Systems such as Kubernetes already provide such guarantees in practice. SSMS users supply application-specific checkpointing/recovery logic, but it is up to SSMS to supply persistent storage, implement restarts, and manage checkpoints.

To correctly orchestrate complex workflows across multiple automata, SSMS relies on the Composable Resilient Steps (CReSt) model [30]. Under CReSt, each automaton receives some messages,

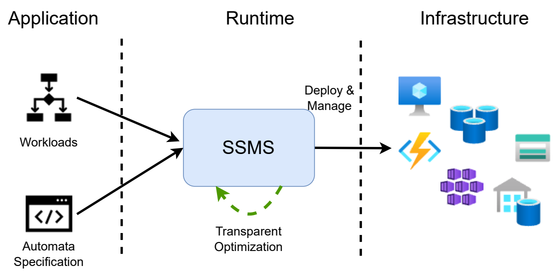


Figure 1: SSMS Overview

updates its local state, and then sends some messages as an fault-tolerant atmoic unit, similar to a database transaction. A step will either complete and have all its effects recovered after failure, or abort and have none of the effects recovered. Applications composed with CReSt are resilient by construction, meaning that external users cannot to distinguish between an execution trace with failures and one without (except through possible performance degradations due to failure handling). We require SSMS components to be written using CReSt *by default*, similar to how most DBMSs execute commands under transactions by default. SSMS takes care of implementing CReSt and exposes it as a primitive for developers, building on top of the performant DARQ system proposed in [30] to reduce overhead.

Importantly, this model allows SSMS freedom to apply a multitude of transparent optimizations to improve performance and save cost for an SSMS application. 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. Many such decisions are fine-grained per-automaton decisions, and depend on dynamically changing load, pricing, and application-level patterns (e.g., hot spots), making it difficult for humans or static workload schedulers to implement them. SSMS is uniquely positioned to implement these optimizations because, as a runtime layer, it can gather real-time statistics, and take advantage of fault-tolerance to transparently make changes to how automata are deployed.

Putting it together – our SSMS design exposes a virtual automata interface to users and transparently deploys automata to underlying cloud infrastructure. Unlike previous actor systems, 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 distributed systems anomalies from developers. SSMS takes advantage of this to implement automatic online optimizations for both performance and cost in a cloud environment.

3 SSMS V0.1 ARCHITECTURE

We now sketch how our prototype SSMS may be implemented in Figure 2, building on the DARQ system proposed by [30]. We start by introducing DARQ, and then outline how SSMS can be implemented as a management layer on top of a DARQ cluster.

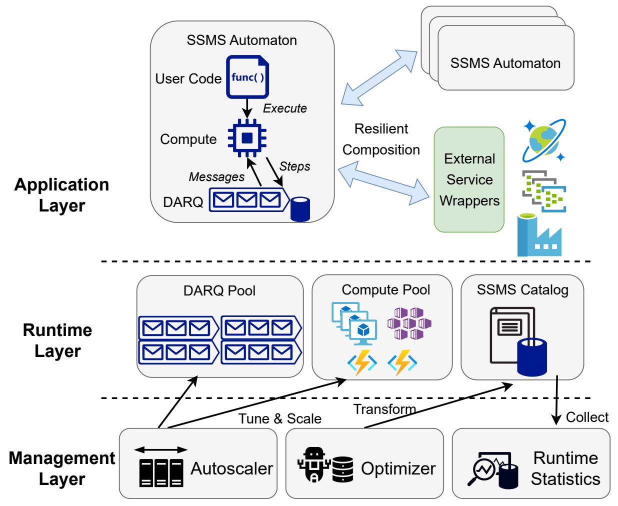


Figure 2: SSMS v0.1 Architecture

3.1 Background: DARQ

DARQ is a cloud-native storage service, and users attach ephemeral compute nodes to a DARQ instance to simulate fail-restart automata running CReSt. Automata logic is written as stateful message handlers performing steps, and DARQ takes care of reliably delivering messages, checkpointing/recovering of automata state, and enforcing CReSt. Developers persist automata state through self-messages(similar to write-ahead log entries), which encodes any state updates to the automaton and is replayed upon failure to reconstruct state. 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. Underneath the hood, each DARQ is backed by a persistent log, and DARQ uses an OCC-like protocol [29] over its log to ensure atomicity and resilience of steps and relies on a number of other DBMS-inspired techniques for performance, such as group commit [19] and early lock release [41].

3.2 Automata Implementation

By default, SSMS automata are implemented with DARQ. At any given time, SSMS maintains a pool of DARQ instances and a pool of ephemeral compute nodes. When the user requests an automaton, 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). To interact with existing cloud services (e.g., a database service), users register them as special, blackbox automata that are not managed by SSMS, but addressable to the rest of the system. Without special handling, SSMS is unable to guarantee resilience across these special automata, meaning that users may observe anomalies or duplicate messages. However, users can optionally choose to implement CReSt-compatible wrappers to address this. For example, an automaton representing an external database may provision an additional table that temporarily stashes all outgoing responses, transactionally update it with the rest of the database, and have a separate process continuously retry delivering responses until recipient acknowledgement to ensure delivery (i.e., the transactional outbox pattern). Doing so ensures CReSt semantics, and allows for the resilience guarantee of SSMS to extend to external systems.

3.3 SSMS Catalog

To manage the various DARQ instances, SSMS needs to implement a highly-available catalog service. 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) a 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 is explicitly tagged with the recipient automaton ID, 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 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 have 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, SSMS expresses each metadata operation as a *resilient workflow*, achievable with DARQs as shown in [30], ensuring that all necessary steps of an operation complete 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 SSMS's optimization to be *cost-based* and *online*, inspired by both adaptive query processing [14] and more recent work in self-tuning and self-driving databases [34, 36, 45]. On a high-level, we propose to implement 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 unified optimization target. SSMS will attempt to launch new automata in a "safe" configuration depending on the target metric (e.g., over-provision if users value performance), and then incrementally *refine* the deployment as it collects information about the behavior of deployed automata. The SSMS auto-scaler 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 auto-scaler is therefore required to detect stable parts of the workload, and launch the approrpiate amount of provisioned resources to support that workload. The primary challenge here is that the optimal provisioning depends 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 auto-scaler much less frequently than the optimizer, so the optimizer has time to converge on a good configuration before the infrastructure shifts under it. Eventually, with techniques similar to the ones proposed for the BRAD cloud data management system [27], SSMS will be able to utilize machine learning and the vast amount of runtime statistics cloud vendors collect to train better cost models and efficiently search the joint planning space of resource provisioning and placement.

4 OPTIMIZING SSMS APPLICATIONS

In this section, we present a (non-exhaustive) list of possible optimizations that are already achievable in SSMS v0.1. 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 on-demand serverless functions. Within each category, there may be further variations based on machine types, pricing (e.g., spot instances), or specialized hardware available (e.g., GPUs). The following optimizations can all be captured as placement decisions:

Scale Up/Down. To scale an automaton up or down, SSMS chooses more powerful/economical compute/storage to deploy it onto. Note also that scaling of compute and storage is separate, and it is possible to redeploy an I/O heavy automaton to faster storage while downsizing 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 slower spin-up time. In SSMS, if an automaton is inactive, its compute node may be deallocated transparently, and replaced with an on-demand FaaS only when a request arrives. Similarly, left-over state of the inactive automaton may be flushed to cold storage. If an automaton has no unconsumed (self or incoming) messages, it becomes stateless and can be unassigned from a DARQ entirely.

Co-location. Physical co-location of disaggregated resources is a key design principle for performant serverless computation [42]. 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 [30], 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 step 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 [32], and our scheme is similar to earlier systems that implement dynamic key migration [28, 31].

4.3 Transparent Replica and Stand-Bys

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 [37]. To switch to a backup, SSMS merely needs to make a local decision to re-route messages and step privilege 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 increase replicas. Alternatively, SSMS can also spin up temporary replica before a scheduled automata relocation, or in response to pre-emption on a spot instance, to reduce performance impact on operations.

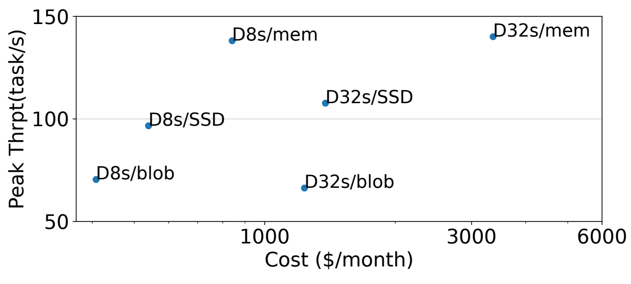
4.4 Layering Abstractions

Some important optimizations cannot be captured through placement. For example, a flexible application may wish to add or remove stateless workers dynamically depending on load; other applications may partition their states, and then dynamically migrate key ranges to respond to hot spots. With SSMS v0.1, users must still manually create automata that represent partitions or additional workers, and route requests between them. SSMS will not be able to optimize the number of workers or partitions because they are application-level logic. One way to address this issue is through multiple layers of SSMS abstractions, where each virtual automata may be implemented by a mini-system of self-managed automata underneath the hood. Consider the stateless worker case; a layered abstraction may provide a special type of automaton called a `AutoScaledStatelessSet<A>`, where `A` is a normal stateless worker. `AutoScaledStatelessSet<A>` exposes the same external interface as `A`, but internally manages a dynamic set of instances, routes messages, and load-balances between them. The rest of the system can refer to `AutoScaledStatelessSet<A>` as a single automaton, and leave the management tasks to the internal (reusable) implementation. Interestingly, such layered abstractions have a solid theoretical foundation in the form of I/O automata composition, hiding, and simulation relations [33]. We envision future SSMS implementations to ship with many such abstractions, akin to "standard libraries" of traditional programming languages.

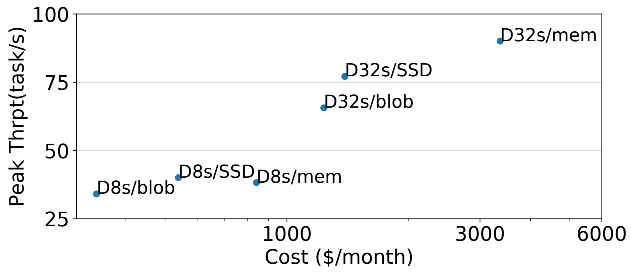
4.5 Preliminary Evaluation

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 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 [11] and three DARQ storage backends: hot-replicated (simulated with volatile memory), managed cloud SSD [5], and Azure storage blob [8]. 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 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 is 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 became cost-efficient with low utilization.



(a) c=1m



(b) c=10m

Figure 3: SSMS Cost-Performance Trade-Off Space

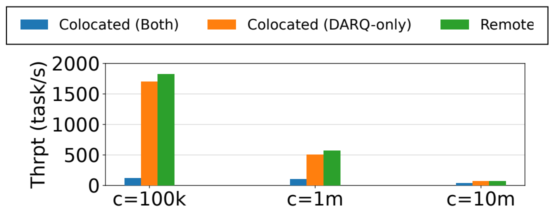


Figure 4: Benefits of Co-location in Different Scenarios

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 advantage 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. 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.

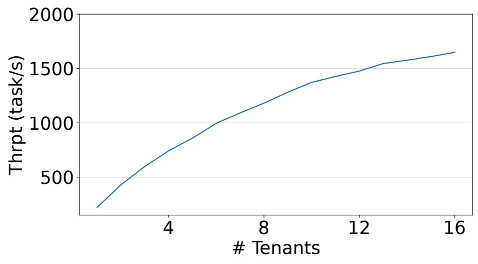


Figure 5: Benefits of Multi-Tenancy in DARQ

Multi-tenancy. Finally, we showcase the effectiveness of multitenancy in DARQs. We simulate multi-tenancy in DARQ by implementing a special message handler that partitions the log manually and runs computation with bounded parallelism (number of tenants). As shown in Figure 5, many tenants can effectively share DARQ's I/O resources up to some limit and almost linearly scale up overall throughput. As expected, though, this effect begins to flatten out as the DARQ's resources are stretched. Such inflection points vary depending on the type of jobs running and the storage backend chosen, and are again best determined by an automated agent such as SSMS rather than manual effort.

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, which burdens developers with managing VM instances and scheduling work intelligently on them. Most modern users manage VMs through higher-level orchestration systems such as Kubernetes [9], Amazon ECS [2], or Apache Mesos [22]. These systems typically employ some intelligent cluster scheduling algorithm to place workload [24], but fundamentally expose a low-level machine-level abstraction (i.e., raw VMs or containers). Recent work has also proposed to extend this paradigm to multiple clouds [43]. Compared to SSMS, such orchestration systems better support compatibility with earlier VM-based cloud software, but has limited ability to optimize applications as low-level black boxes.

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) [3, 6]. Despite the promises of simplified and flexible cloud programming [26], FaaS is considered flawed and cannot support efficient data processing, state management, or complex coordination [21]. Researchers have proposed various solutions by either orchestrating fault-tolerant workflows across FaaS instances [4, 12, 13], or improving stateful programming support [25, 42, 47]. 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 [35, 48], Orleans [15] in its programming model and abstraction. However, SSMS is based on more classical distributed systems modeling of I/O automata [33]. 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 [16], 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 [17] and Kafka Streams [46]. Most notably, many modern stream processing systems also provide strong exactly-once guarantees and take advantage of the guarantee to dynamically optimize for execution [23]. 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 [38, 44]. The most radical of these approaches argue that cloud applications can be built on top of a high-performance distributed SQL database [40]. Other systems proposals focus on tackling the challenge of managing fine-grained disaggregated resources in the modern data center, often on the hardware level [39]. 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 [20]; unlike this proposal, SSMS provides resilience as part of the guarantee, which enables many of our optimization techniques.

New Directions. Recent proposals for Sky Computing [43], Hydro [18], and Self-Optimizing Data Meshes [27] call for a rethink for how developers interact with the cloud. The key insights are heterogeneous infrastructure, strong abstractions that simplifies user programming, and intelligent self-optimization of user applications. SSMS is very much proposed in the same spirit as these initiatives, but focuses on the smaller and more concrete problem of transparent and resilient state management, which we believe is crucial for solving the bigger challenge pointed out by these other work.

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 optimizations. We sketched a prototype design of one such SSMS that can 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.

ACKNOWLEDGMENTS

We are grateful for the support of the MIT DSAIL@CSAIL member companies.

REFERENCES

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

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

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

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

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

[7] Event Hubs – Real-Time Data Ingestion. https://azure.microsoft.com/en-us/products/event-hubs, 2023.

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

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

[10] 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.

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

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

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

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

[15] 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.

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

[17] 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.

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

[19] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 1–8, New York, NY, USA, 1984. Association for Computing Machinery.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, Oct. 2018. USENIX Association.

[40] 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.

[41] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. In G. Gottlob and M. Y. Vardi, editors, *Database Theory — ICDT '95*, pages 139–147, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] S. Zhuang, S. Wang, E. Liang, Y. Cheng, and I. Stoica. ExoFlow: A universal workflow system for Exactly-Once DAGs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 269–286, Boston, MA, July 2023. USENIX Association.