DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps

TIANYU LI*, MIT CSAIL, USA
BADRISH CHANDRAMOULI, Microsoft Research, USA
SEBASTIAN BURCKHARDT, Microsoft Research, USA
SAMUEL MADDEN, MIT CSAIL, USA

Providing strong fault-tolerant guarantees for the modern cloud is difficult, as application developers must coordinate between independent stateful services and ephemeral compute and handle various failure-induced anomalies. We propose *Composable Resilient Steps* (CReSt), a new abstraction for resilient cloud applications. CReSt uses fault-tolerant *steps* as its core building block, which allows participants receive, process, and send messages as a single uninterruptible atomic unit. Composability and reliability are orthogonally achieved by reusable CReSt *implementations*, for example, leveraging reliable message queues. Thus, CReSt application builders focus solely on translating application logic into steps, and infrastructure builders focus on efficient CReSt implementations. We propose one such implementation, called DARQ (for *Deduplicated Asynchronously Recoverable Queues*). At its core, DARQ is a storage service that encapsulates CReSt participant state and enforces CReSt semantics; developers attach ephemeral compute nodes to DARQ instances to implement stateful distributed components. Services built with DARQ are resilient by construction, and CReSt-compatible services naturally compose without loss of resilience. For performance, we propose a novel speculative execution scheme to execute CReSt steps without waiting for message persistence in DARQ, effectively eliding cloud persistence overheads; our scheme maintains CReSt's fault-tolerance guarantees and automatically restores consistent system state upon failure. We showcase the generality of CReSt and DARQ using two applications: cloud streaming and workflow processing. Experiments show that DARQ is able to achieve extremely low latency and high throughput across these use cases, often beating state-of-the-art customized solutions.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Information systems** → *Cloud based storage.*

ACM Reference Format:
Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2023. DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps. *Proc. ACM Manag. Data* 1, 2, Article 117 (June 2023), 27 pages. https://doi.org/10.1145/3589262

1 INTRODUCTION

Modern cloud developers build applications by composing various distributed components such as vendor services (e.g., Amazon S3 [13], Azure CosmosDB [14]), custom-built microservices (e.g., with

*Work started during internship at Microsoft Research.

Authors' addresses: Tianyu Li, litianyu@csail.mit.edu, MIT CSAIL, Cambridge, Massachusetts, USA, 02139; Badrish Chandramouli, badrishc@microsoft.com, Microsoft Research, Redmond, Washington, USA, 98052; Sebastian Burckhardt, sburckha@microsoft.com, Microsoft Research, Redmond, Washington, USA, 98052; Samuel Madden, madden@csail.mit.edu, MIT CSAIL, Cambridge, Massachusetts, USA, 02139.

2836-6573/2023/6-ART117
https://doi.org/10.1145/3589262

Kubernetes [20] or Azure Service Fabric [17]), and stateless compute (e.g., serverless FaaS [12, 16]). The key challenge when building stateful applications in this highly distributed environment is fault-tolerance. Despite the wealth of work in highly-available and reliable individual components, developers must still painstakingly ensure they work together correctly upon failure [47].

Consider a simple example where service A must pass the result of a user request to another service B, which processes the result and updates its state (e.g. for aggregation in analytics). Service A may fail after sending the result to B, but before it persistently records that it has done so, leading to duplicate results sent on recovery. Similarly, B may fail and lose the results after A delivers the result, leading to lost data. To mask this complexity, developers want end-to-end *resilience*, defined as behavioral indinstinguishability of the application from failure-free scenarios, except perhaps with performance degradation due to failure handling.

Resilience is essential for applications with strong correctness requirements (e.g., involving monetary transactions or irreversible physical effects); in other cases, it is a powerful abstraction that hides distributed system complexity from the average user, simplifying user applications and increasing user productivity. Such resilient systems are prolific in the modern cloud: for example, batch processing systems such as Spark [70] transparently retry failed nodes to mask failures; Netherite [30] and Ambrosia [39] leverage DBMS-style logging across heterogeneous machines, Kafka Streams [68] implements consistent stream processing using persistent logs and cross-node transactional writes, and Azure Durable Functions [15, 30] guarantee effectively-once execution of composed serverless functions through a carefully implemented stateful messaging layer. These solutions share many characteristics, such as dependency tracking, asynchronous checkpoints, carefully defined and enforced atomicity scopes, message retries/deduplication, etc. But perhaps the greatest commonality is that these solutions are highly complex, specialized, and difficult to implement. To build a new resilient system, infrastructure developers must still design their own mechanisms and handcraft their implementations, which is often time-consuming and error-prone. Ideally, developers can build heterogeneous distributed components on a general abstraction and compose them into resilient systems with reusable building blocks.

In this work, we present *Composable Resilient Steps* (CReSt), a new abstraction that separates composable application logic from resilient distributed system implementation. In CReSt, components operate in *steps*: a component consumes some (≥ 0) messages, updates its local state, and optionally sends a series of messages as a result. Because almost any distributed system can be specified with message-passing primitives [37], it is easy to translate existing systems into a sequence of steps. CReSt effectively *decouples* application logic (specified as CReSt steps) from mechanisms for resilience and composition (implementation of CReSt steps). For developers, CReSt steps are all-or-nothing and uninterruptible by failures, similar to database transactions, making them a powerful abstraction – as shown later, applications written with steps are resilient by construction and seamlessly compose with other CReSt applications while preserving resilience. For infrastructure builders, CReSt is a narrow interface that allows for efficient, reusable implementations that benefit many applications.

To demonstrate the effectiveness of CReSt, we build DARQ, a highly optimized CReSt implementation for the modern disaggregated cloud. DARQ is a storage service that encapsulates the state of CReSt components and emulates a failure-free messasging layer between them; developers attach ephemeral compute to DARQ to receive messages and perform CReSt steps. When using DARQ, developers focus on application logic as steps, and DARQ transparently ensures atomicity and durability of steps via (group) commit, and resilient composition via message retry and deduplication. Individual services built with DARQ are therefore resilient, and resiliently compose with other CReSt-compatible services through a narrow message-passing interface. Under the hood, DARQ consists of a layer of CReSt logic on top of a performant log, making it lightweight to deploy

and scale. Correctness of DARQ is grounded in our formal modeling of CReSt components as I/O automata [53] in a fail-restart environment where faulty components are quickly replaced with previous snapshots of their state from reliable cloud storage. For performance, We introduce a novel scheme of *speculative step execution*, where DARQ messages are processed before persistence. Similar in spirit to early lock release [64], speculative execution allows for subsequent steps to occur without waiting for persistence of previous ones, effectively *parallelizing* slow cloud persistence, leading to dramatically reduced end-to-end latency. DARQ internally tracks step dependencies and automatically restores the system to a consistent state upon failure, transparent to CReSt users.

Note that DARQ is merely one implementation of CReSt, and other implementation are possible. For example, one can use distributed transactions [29, 45, 57] across senders, step processors, and receivers to orchestrate atomic steps, or adopt widely-used cloud programming patterns such as transactional outboxes [6] and Sagas [5]. In contrast, DARQ is specifically engineered to work with storage-compute separation in the modern cloud and compose decoupled services, free of any requirement of cross-service atomicity such as traditional distributed transactions with two-phase commit. In exchange, DARQ incurs additional persistence round-trips (mitigated by speculative execution) in favor of distributed coordination across service boundaries. To show that CReSt can express diverse cloud applications, and that DARQ offers competitive performance to alternatives, we build an example streaming application and workflow application using DARQ. Our evaluation of these systems shows that DARQ can help build resilient cloud applications, often outperforming handcrafted solutions in terms of throughput and latency, while being simpler and more reusable.

To summarize, we make the following contributions:

- (Sec. 2) We propose CReSt, a general abstraction for resilient cloud applications based on composable fault-tolerant steps.
- (Sec.3) We design and build DARQ, an efficient CReSt implementation as a service for the modern cloud, based on group commit and reliable message delivery.
- (Sec.4) We propose a novel speculative step execution scheme with DARQ to bypass slow cloud storage and reduce end-to-end latency in composed cloud applications.
- (Sec.6) We prototype and evaluate a stream-processing system and a cloud workflow system using DARQ, showcasing that DARQ-based applications are performant, general, and easy to build compared to the current state-of-the-art.

Finally, we cover related work in Sec. 7 and conclude in Sec. 8.

2 COMPOSABLE RESILIENT STEPS

We now introduce the Composable Resilient Steps (CReSt) model in detail. With CReSt, developers design applications based on a simple mental model of stateful participants and message channels, as shown in Figure 1. Participants perform atomic, all-or-nothing *steps* that receive some (≥ 0) messages, process messages and update their local state, and finally send some (≥ 0) messages. In more concrete terms, each CReSt participant is a state machine with some local state (accessible as `participant.State`). CReSt programs invoke `participant.Receive()` to acquire messages intended for the participant, and evolve the system exclusively through `participant.Step(consumed, newState, outgoing)`, where `consumed` is the list of messages consumed, `newState` is the updated component state, and `outgoing` is the resulting list of messages to send to other participants. Once a step completes, its effects become fault-tolerant – the messages it consumes will not be delivered or consumed again, any local participant state becomes recoverable, and any outgoing messages will be delivered asynchronously and exactly-once to other CReSt participants. This *implicit resilience* dramatically simplifies the application logic, as it

hides complex low-level mechanisms (e.g., checkpointing, recovery, retries, and deduplication) for developers to focus on the high-level application logic.

Steps often have application-level meanings, corresponding to processing of a request, completion of task, etc. For example, a simple RPC service that handles client requests to atomically compare-and-swap (CAS) a named value can be thought of as performing steps that consume the client request message, atomically execute the operation, and reply with the result. In this case, each CAS participant has local state that is a dictionary of named values; the service logic consists of a (sequential) processing loop that calls `participant.Receive()`, deserializes client requests, performs the compare-and-swap operation, and invokes `participant.Step()` to persist the effects and expose responses. If clients and the CAS server both participate in CReSt, the resulting end-to-end application is also resilient, because failures cannot interrupt steps, only delay execution until retry.

2.1 Example Applications with CReSt

Before formally introducing the CReSt model, we highlight the generality of CReSt using several real-world applications that can be easily specified using CReSt steps.

Cloud Workflows: Cloud workflows are programs that chain together multiple services in a distributed environment. Systems such as Temporal [21], Azure Durable Functions [15], or Amazon Step Functions [1] implement workflows in various ways as an important cloud programming abstraction. In CReSt, one can think of workflows as a coordinator state machine (which may be virtual) sending a series of messages to component services, and keeping workflow state locally (i.e., the coordinator performs `Step(prevResponses, programState, nextRequest)`, and services perform `Step(request, state, response)`). Workflows specified with CReSt are resilient when executed on an environment that correctly implements CReSt achieving the same guarantee as systems such as Azure Durable Functions. Such modeling is also similar in spirit to the *serverless message-passing model* used by Netherite [30] and other similar work [65, 72], to implement resilient workflows.

Stream Processing: Streaming systems consist of various stream processors performing computation on an underlying stream of data organized into partitions and topics. In CReSt, a stream processor can be modeled as continuously performing steps to consume batches of messages from upstream, processing them locally, and sending results downstream. For example, a stateless filter processor may receive message m and emit `Step(`$[m]$`, null, `$[m]$`)` if the predicate evaluates to true on m, or emit `Step(`$[m]$`, null, [])` if false, marking m as consumed without sending it downstream. A stateful operator may not immediately output a message, but may continuously take steps to update its local state (e.g., a count operator receiving message m may emit `Step(`$[m]$`, count + 1, [])`, and only emit `Step([], count, [`$asMessage(count)$`])` periodically). Several steps may also be coalesced into a single step for batched I/O. Stream processing specified with CReSt steps is resilient by default, providing "exactly-once" semantics [62, 68], in the sense that each data item in the input data stream is reflected exactly-once in the result. Application developers do not need to implement retransmissions, checkpointing, or other fault tolerance mechanisms, which are handled by CReSt implementations.

Distributed Commit Protocol: Traditional distributed databases implement ACID transactions using the 2-phase commit protocol, which is tightly integrated with the write-ahead log for fault-tolerance [57]. This protocol can be re-specified in CReSt as a sequence of steps – using a successful 2-phase commit sequence as an example, the transaction coordinator first uses a step to note down the start of commit of t and sends prepare messages to workers (i.e., `Step(request, coordinatorState, PREPAREs)`), and workers take steps to check if the commit is allowed and respond with the result (i.e., `Step(PREPARE, workerState, OK)`). When all workers respond,

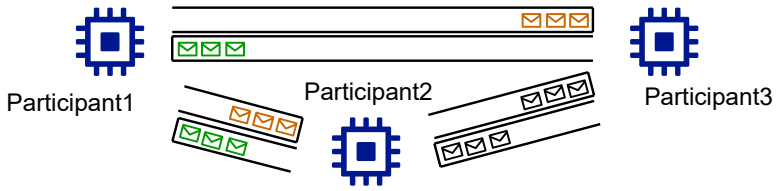


Fig. 1. An example of a CReSt system with 3 participants. Each participant is a state machine performing steps that atomically update the state and send/receive messages to/from channels.

the coordinator steps all OK messages, locally marks the transaction as committed, and sends commit requests (i.e., `Step(OKs, coordinatorState, COMMITs)`), followed by workers resiliently applying the commit (i.e., `Step(COMMIT, participantState, ACK)`). Note here that the protocol designer does not need to reason about component failures or message loss, as CReSt transparently guarantees resilience.

2.2 Fail-Restart Model

Given a CReSt application, with each component and the steps it performs, our goal is to *deploy* it to the failure-prone cloud while preserving application semantics. Intuitively, our goal is *behavioral indistuinguishability* – users should not be able to differentiate between executions of the CReSt application in a failure-free environment and one that automatically recovers from failures. This is difficult; consider again the CAS service – the service may fail without sending the reply, or the CAS service may send the result, but crash and lose the updated state. To make matters worse, resilient steps in our example involve more than just the CAS service; a service that is resilient by itself may still fail to guarantee resilient steps, as clients may crash and lose replies.

We address this in two steps. **First,** we introduce the *fail-restart model* (Section 2.2). Unlike the fault-oblivious CReSt model, it closely models cloud deployment environment by subjecting all components to possible failure, but allowing them to (quickly) restart and recover their state from durable storage. **Next,** we present a *transformation* (Section 2.3) that automatically converts a CReSt application to be fail-restart, and show that this transformation is faithful to the original semantics.

2.2.1 Modeling Cloud Failures. As is standard in distributed systems modeling, we model participants and channels using I/O automata [53]. Each participant is an automaton that performs atomic stateful actions to its local state, and interacts with other automata using asynchronous messages. A step is formally a segment of original automaton execution that starts with some (≥ 0) Receive actions, followed by arbitrary local updates, and then some (≥ 0) Send actions. Obviously, we can decompose any execution trace into a series of steps. As mentioned, we define failures in CReSt as *fail-restart*. More formally, a fail-restart automaton has two copies of its state — one volatile (e.g., in-memory) and one persistent (e.g., checkpointed on S3). A fail-restart automaton normally operates based on its volatile state, but can perform a Commit action to copy its volatile state to persistent storage or vice-versa with a Restore action. Fail-restart automata also react to special Restart inputs by erasing their local in-memory state; this corresponds to automated management tools such as Kubernetes detecting and repairing a failed node. Note that message channels are unaffected by restarts and will continue to deliver any in-flight, unreceived messages in order. In practice, the "restarted" machine is likely an entirely different one launched to replace an unresponsive machine and may even co-exist with the original machine temporarily [8]. We assume that infrastructure providers can effectively mask this with mechanisms such as leases [40] or distributed consensus [48, 59].) This external signal can arrive arbitrarily and unconditionally,

corresponding to the arbitrary nature of failures. Our goal, then, is to translate the naively specified CReSt application (corresponding to normal I/O automata with no failures) to fail-restart automata.

2.2.2 A non-solution. At first glance, we may achieve resilience by executing steps inside the fail-restart automata and inserting Commit after each step. However, this naive transformation does not achieve resiliency. Performance implications aside, the key challenge here is *atomicity*. Consider for example an execution where a participant receives a Restart right after a Receive, before it has done Commit. Then the recovered state corresponds to a point before the Receive, but the message may have already been removed from the queue and appear lost. Clearly, it is not enough to just checkpoint and restore components individually: to recover to a consistent state, we need a more carefully designed protocol, which we will now present.

2.3 Implementing CReSt: A First Cut

We now sketch a basic fault-tolerant CReSt implementation by presenting a transformation from a failure-oblivious CReSt application to fail-restart automata. We intend for this implementation sketch to provide intuition for correctness. This scheme also serves as a starting point for DARQ, our optimized CReSt implementation, which we discuss in detail in Section 3.

As sketched in Figure 2, the transformation wraps the CReSt application and adds two persistent message queues – an in-queue and an out-queue of messages[1]. At a high level, our goal is to use these two queues and automatic invocations of Commit to simulate a failure-free execution environment for the fault-oblivious CReSt application in a fail-restart setting. To achieve this, the transformed automaton continuously pulls messages from the network into the in-queue; messages from different channels are not ordered with each other and may appear in arbitrary order on the in-queue. These messages are then supplied to the CReSt application as results to `participant.Receive()`. When the CReSt application invokes `Step`, consumed messages are removed from the in-queue, and the outgoing messages are added to the out-queue. Note that message consumption need not be in strict FIFO order – suppose `participant.Receive()` yields messages m_1 and m_2, the participant may consume them in arbitrary order, or simultaneously as an atomic unit, by explicitly supplying either m_1, m_2, or $\{m_1, m_2\}$ as the `consumed` argument to `Step`. The transformed automaton periodically performs a Commit that persists both the queues and the state of the original automaton without interleaving Commit with any `Step`, which ensures that no partial step is committed.

On Restart, however, even if the recovered transformed automaton state contains no partial steps, there may be partial leftover effects on the network. Specifically, a Restart may occur after message receipt but before messages are persistent on input message queues, or after sending but before the output message queue can mark it as sent. We borrow from the classical TCP algorithm [60] by retrying and deduplicating messages. The transformed automaton continuously retries out-queue messages until acknowledged, which happens after the destination automaton persists the received message. Acknowledge messages are then pruned from the system. To avoid processing the same message more than once due to retries, in-queues deduplicate incoming messages using sequence numbers. Each outgoing message includes a unique identifier that consists of a participant id and a sequence number assigned as they enter the out-queue. Each in-queue maintains a deduplication table that maps each participant id to the largest seen sequence number from that participant, neglecting messages with lower sequence numbers. This deduplication state is persisted during Commit along with the rest of the automaton state; we similarly enforce that

[1] "Queues" are really misnomers, as they are interleavings of multiple underlying message channels, and therefore only partially ordered. Additionally, as we explain later, messages on the queue can be buffered and then consumed out-of-order. We have decided to refer to them as queues in accordance with tradition [29].

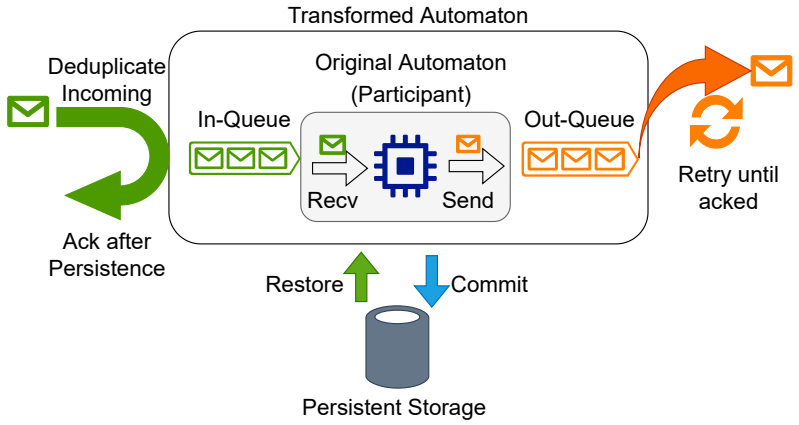


Fig. 2. CReSt Transformation & First-Cut Implementation – The original automaton performs pure CReSt steps. Our translated automaton uses locally stored messaging queues, checkpoints, and retry-based deduplication to achieve fault-tolerance guarantees.

Commit does not occur between receiving a message and updating the deduplication state, so they appear atomic upon recovery.

Combined, the two message queues effectively emulate a reliable, fault-tolerant, exactly-once messaging layer for the CReSt application in a fail-restart environment. This ensures that the transformed automaton faithfully executes steps from the original CReSt participant despite failures. Note that our construction also guarantees resilience for non-deterministic participants, because the message queues "determinize" a step by capturing its effects persistently before exposing them [37]. If a non-deterministic step experiences failure, either its effects are recovered and the step is not re-executed, or none of its effects are recovered or exposed and re-execution is not distinguishable from first invocation.

3 DARQ

The previous section described how a CReSt application can be faithfully translated into fail-restart automata with persistent message queues, retries, and deduplication. Implementing this efficiently in the cloud, however, presents major challenges. Most significantly, infrastructure builders must contend with:

- **Stateless Compute:** Modern cloud architectures emphasize flexibility and on-demand scale-out. Most often this involves separating compute and storage and using autoscaling stateless compute services such as serverless FaaS [44]. The fail-restart model must be adjusted to this setting.
- **Incremental State:** Our earlier construction saves the entire state of a participant and all queued messages in each Commit. This performs very poorly when queues are deep or participant state is large.
- **Blackbox Services:** Almost all modern cloud systems rely on cloud provider services such as Amazon S3 or Azure CosmosDB. Such services typically expose an RPC-based API, and cannot be easily modified by users. For CReSt systems to be correct, developers must extend CReSt semantics to existing services.
- **Latency:** As specified, CReSt requires frequent and synchronous persistence of participant state for fault-tolerance, leading to high latency in processing. To make matters worse, in complex applications where processing requires many message-passing rounds, the latency overhead is accumulative and can quickly become unacceptably high.

To overcome these challenges, we introduce DARQ (Deduplicated, Asynchronously Recoverable Queue), an efficient implementation of CReSt that addresses each of the challenges above.

3.1 Processors and Self-Messages

We first give a high-level overview of the DARQ APIs and guarantees, with a running example of the CAS microservice from before. As shown in Figure 3, at the core of DARQ is a storage service that encapsulates the state of one CReSt participant plus the persistent message queues introduced before. External clients or other participants communicate with the encapsulated participant by sending messages through the DARQ producer API. For example, an external client may submit a request to invoke CAS(x, 42, 0) by adding it as a message to the relevant DARQ, as shown on the left in Figure 3. To evolve participant state, developers attach ephemeral compute nodes, called *processors*, to DARQs. Each processor is loaded with the "business logic" of the CReSt participant that performs fault-oblivious steps, and DARQ pushes any messages received to it for application handling. Processors store participant state locally in volatile memory for direct access, but must persist such state and expose effects of processing through the DARQ step API. As CReSt specifies, a DARQ step request includes identifiers for the messages consumed, updated state, and any outgoing messages (e.g., the example CAS service step in Figure 3 consumes the request, update value of x to 42, and send result to client with success).

DARQ encodes participant state through *self-messages* – processors send their future (amnesic) selves enough information to restore state. In the simplest case, as a part of each step, the attached processor copies the entire updated state as a message (or a succinct reference to externally persisted state). The message is then consumed in the next step, which generates a new self-message with updated state. For large or incremental state, self-messages can be used much like a write-ahead log – each step generates a delta record of the state and does not consume the previous self-messages. Then, at recovery time, all the unconsumed self-messages will be replayed to the processor in step order. For example, the step shown in Figure 3 can be encoded as a delta record setting x to 42; upon recovery, these deltas are applied to reconstruct the full state. For efficiency of replay, users may periodically perform steps to consume previous self-messages and emit a new self-message that coalesces previous deltas, similar to checkpointing in traditional DBMSs [56] (more on this in Section 3.2).

To summarize, DARQ is a concrete implementation of the CReSt construction presented earlier, structured as a cloud service working with ephemeral compute nodes called processors. Users write applications in a fault-oblivious manner and DARQ transparently ensures fault-tolerance by automatically enforcing CReSt semantics, persisting state, and recovering from failures. We have designed DARQ to expose an API similar to streaming/messaging services such as Kafka [7] or EventHubs [25], but with support for encoding state through self-messages. Note here that despite the superficial similarities, DARQ fundamentally subscribes to the CReSt model rather than streaming or simple message-passing.

3.2 DARQ Implementation

We now cover our implementation of DARQ in detail:

Storage Backend. Recall that each DARQ instance logically stores three pieces of state (in/out message queues and participant local state) that must be updated atomically, as shown in Figure 4. As shown in Figure 4, DARQ multiplexes them all onto an underlying physical log of messages (recall that state is encoded as self-messages), distinguished from each other with a message type field (shown as different colors in Figure 4). Each message is uniquely identified using a log sequence number (LSN). Steps are atomic in DARQ as they are simply log appends: consumption of previous messages is represented with a special completion message with consumed LSNs, and appended

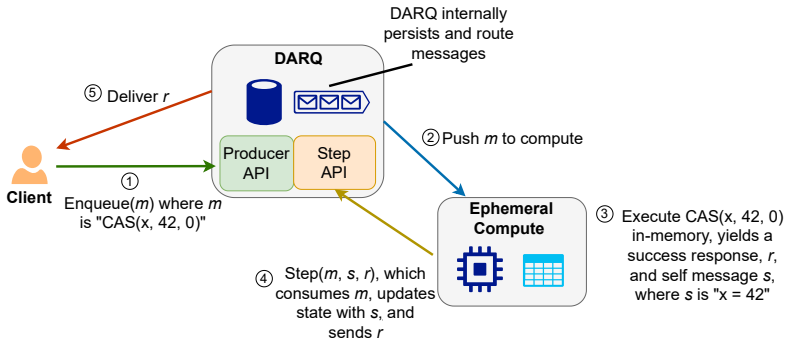


Fig. 3. DARQ Overview – Developers attach ephemeral compute to each DARQ instance and DARQ delivers messages by pushing to the compute; the compute node then interact with the rest of the system via steps in DARQ. State is maintained as volatile data structures in the compute node, and persisted as self-messages.

along with out-messages, similar to database commit records. Figure 4 shows one such step. Here, the DARQ service exists in a separate failure domain as the compute node. DARQ streams in-messages to the attached processor in LSN order, and the processor performs a step locally based on the in-message at 102 and the latest state at 101. After local processing, the compute node has evolved its participant state and computed a series of out-messages. These messages are then appended back to DARQ atomically with a completion record marking 101 and 102 as consumed. For our implementation, we use FasterLog [19], a popular open-source high-performance log. The DARQ log is flushed to reliable cloud storage for fault-tolerance by default, but may also rely on standard replication techniques instead [48, 59].

Failure Recovery. When a processor restarts, it loses all local volatile state and must reconstruct its state using self messages in DARQ. DARQ enters replay mode when a processor disconnects and scans the log sequentially in 2 passes: first, to identify completion records and remove consumed message from replay, and second, to replay unconsumed message to the recovered processor. Additionally, DARQ reorders self-messages ahead of any in-messages during replay, because self-messages during normal stepping are always appended to the end of the log, potentially behind unconsumed in-messages. Note that because DARQ normally does not need to stream self-messages to processors (as they are merely for recovery purposes), processors can tell that they are in replay upon entering a self-message, and that replay is finished upon encountering the first in-message. In real deployments, DARQ must additionally guard against imperfect failure detection, which may lead to two processors both attaching themselves to DARQ, not necessarily aware of each other's existence. For this, we introduce *incarnation numbers* for processors. Each processor, when first attached to DARQ, is assigned a unique, monotonically increasing number stored as part of the DARQ instance state. DARQ will only accept requests tagged with the currently recognized incarnation number, notifying processors with smaller incarnation numbers so they can gracefully terminate. Obviously, the value of the largest recognized incarnation number must be persistent for correctness, and DARQ uses FasterLog's internal epoch framework [51] to persist the value as metadata along with each log flush and ensure that the persisted number is consistent with the log content.

Enforcing Invariants. The DARQ service enforces invariants around steps to guard against programming errors or other anomalies and preserve CReSt stepping semantics. Every message is allowed to be consumed at-most-once; steps that attempt to consume any message already stepped

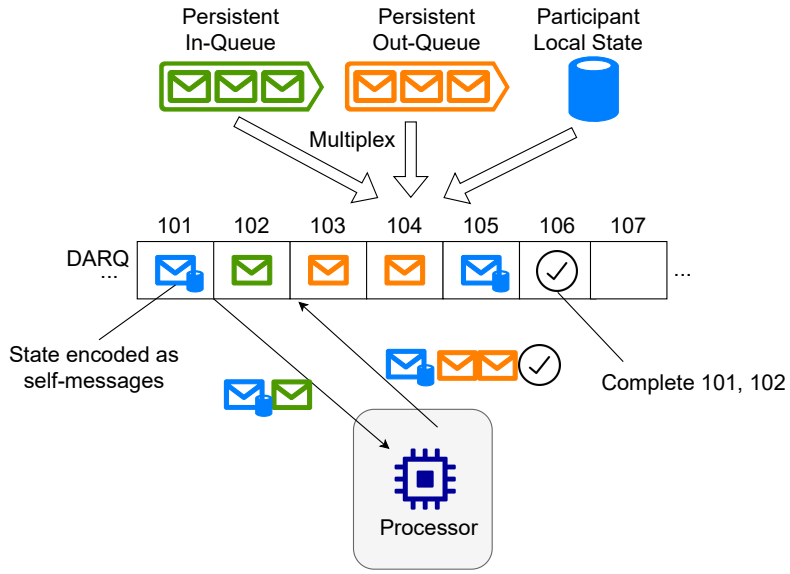


Fig. 4. DARQ Implementation – Underneath the hood, DARQ multiplexes data structures specified in the CReSt model onto a single log for storage efficiency and atomicity.

are rejected in its entirety, similar to an aborted transaction. DARQ enforces that any steppable (i.e., in or self) message is stepped at most once. To achieve this, DARQ maintains an in-memory hash table of steppable messages; when messages are added either through the producer API or as part of a step, their LSNs are added to the table. We adopt an OCC-like [46] protocol to serially validate steps by checking that all the consumed LSNs are in the steppable message table, and remove them. Steps that fail this validation process are rejected entirely. We reconstruct the steppable message table upon DARQ failure by scanning the log. Additionally, as presented in Section 2.3, DARQ needs to deduplicate incoming messages. The producer API takes extra deduplication information consisting of a globally unique producer ID and a sequence number (i.e., the aforementioned TCP-inspired deduplication scheme). When sending from a DARQ, DARQ automatically populates deduplication information. Otherwise, developers may supply application-specific values or omit them to bypass deduplication. DARQ maintains a deduplication table mapping from producer ID to the largest seen sequence number that is similarly persisted and kept in-sync with log with [51].

Background Maintenance. In the background, DARQ delivers output messages to their intended recipients, automatically retrying until the recipient acknowledges, as Section 2.3 prescribes. In addition, DARQ also needs to perform tasks such as garbage collection or checkpoints to prevent unbounded growth of self-messages. We perform these tasks by running a special process that scans the log, co-located with DARQ servers. Together with the table of unstepped messages, we can compute a prefix of the log that has been *completed* – a message is completed if it has been consumed by a step as an in/self-message, or if it has been acknowledged by the recipient as an out-message. DARQ periodically truncates the largest completed prefix for garbage collection.

Checkpoints. Lastly, long-running participants may see self-messages growing unboundedly in DARQ, preventing garbage collection and slowing down recovery. DARQ supports DBMS-style checkpointing [56] to truncate self-messages and prevent unrestricted growth. As stated before, logically, a checkpoint is a special internal step that marks all previous self-messages complete, and produces a single self-message that encapsulates the current state of the participant. Concretely

in implementation, the completion record for a checkpoint has a special record type to implicitly complete self-messages to some log offset, and users supply a custom function to snapshot the current participant state.

3.3 Integrating with Blackbox Services

As noted previously, components using DARQ to compose with each other enjoy fault-tolerance and appear to process messages exactly-once. However, it is inevitable that DARQ systems need to communicate with external, non-DARQ cloud services. Unfortunately, this usually means that DARQ regresses to at-least-once guarantees because DARQ fundamentally relies on retries to achieve exactly-once processing, and external services are generally oblivious to DARQ's deduplication schemes.

To address this, DARQ (and by extension, general CReSt) systems must compose with existing cloud services through CReSt-compatible wrappers. The general idea is for external services to behave like CReSt components by implementing deduplication and atomic steps without DARQ. To illustrate, we sketch a way for services to achieve this using the example of an ACID DBMS. Consider the well-known transactional outbox pattern [6], where the database must atomically update application state and send events/messages. The standard solution calls for a separate "outbox" co-located within the database; any request atomically writes the result to the outbox with any storage effects, most likely with a transaction. A separate, stateless message broker component then polls and sends these messages. To deduplicate incoming requests, developers can either rely on request idempotency, or implement our DARQ deduplication scheme with a table and transactionally update it with the request. Other services may similarly be modified to be CReSt-compatible and therefore resiliently compose with DARQ. Note that such solutions generally introduce overhead in performance, and complexity, but applications may already do so in practice (e.g., many applications already rely on transactional outboxes). Alternatively, services may be engineered to causally log their state and use an approach similar to [62] to deduplicate correctly after failure. We leave a more detailed study of integration strategies for future work.

Users can implement such composition in DARQ using a special type of message called *activity*. Semantically, an activity is a self-message that encapsulates a piece of work to be done on some external service. The processor is then free to execute some custom logic, most likely an RPC to the external service in question, using the LSN of the self-message as a unique deduplication token. Then, when the external service completes the request, the activity message can be marked as complete and later removed from the system. Activities that execute on CReSt compliant services guarantee resilience. For other services, DARQ regresses to the typical at-least-once guarantee common in the cloud today by retrying activities. Note that most read-only requests can bypass this mechanism, as they can be transparently retried if failed. Overall, activity support in DARQ is similar to existing cloud workflow systems such as Durable Functions or Temporal, and is largely captured by CReSt, using participant state to track individual activity progress.

4 SPECULATIVE EXECUTION

Even though DARQ correctly implements CReSt, the problem of performance remains — in the previously presented scheme, each time a message flows through DARQ, it must be persisted, which incurs a substantial latency overhead (up to 10ms if using cloud storage). Worse still, such overhead is cumulative: the more messages sent, and the more disaggregated an application is, the bigger the overhead. We propose a solution using *distributed speculative execution*, in which messages are transmitted and processed before persistence. Obviously, when DARQ fails, some processed messages or state may no longer be available, and current progress must be rolled back

and replayed from earlier sources [37]. In this section, we briefly introduce how we implement speculative execution and discuss the trade-offs of our solution.

4.1 Special-Case: Intra-Participant Speculation

Recall that in our earlier formulation, DARQ is a separate entity from the processor that attaches to it and provides compute capabilities. Consequently, one might assume that DARQ must persist messages before exposing them to the attached processor, as the processor may survive DARQ failure and continue operating on lost state. That said, processors interact with the rest of the DARQ system exclusively through message-passing that DARQ intermediates. Because DARQ is a log underneath the hood, any result of a step must be ordered behind the input to the step (i.e., consumed messages), and only recovered if the input is recovered because the log guarantees prefix recovery by definition. As a result, if DARQ sends messages to the attached processor before persistence and fails, processing of lost messages is transparent to other participants as the effects will have been lost as well. The processor, however, will have local state corresponding to rolled back steps and must recover to a point that corresponds to the recovered DARQ state, which can be achieved simply by erasing the processor state and treating the case as one of processor failure. To summarize, intra-participant speculation allows DARQ to safely expose uncommitted messages to attached processors without special handling while maintaining, on condition that DARQ recovery prompts a processor restart. Because we assume that processors fail more often than DARQ, we believe this cost to be acceptable for most deployments and therefore enable intra-participant speculation by default.

4.2 General Speculation

With general speculation, DARQ nodes send and consume messages to/from each other before they are persistent. Doing so removes persistence from the execution critical path, and can result in significant speed-ups for deep message-passing chain and/or slow storage, both of which are common in modern cloud environments [44]. However, general speculation leads to increased complexity in failure recovery – compared to the special case from before, a DARQ failure may result in losing some speculative steps but retaining (partial) effects of them in other surviving DARQ nodes. To make matters worse, these leftover effects might have been persisted or even exposed to external observers.

To solve this problem, we design and implement a rollback-based recovery scheme based on a modified version of the Distributed Prefix Recovery algorithm [50]. On a high-level, a consistent message-passing system state requires that if a participant's state reflects receiving a message, then the state of the sender reflects sending that message [33]. This produces a natural notion of *dependency* across speculative CReSt steps – a step s depends on step t if s speculatively consumed a message m that was produced by t. Our solution explicitly tracks such dependency by appending tracking information to each message. Each DARQ instance then locally accumulates dependency information; a distinguished coordinator node periodically joins information across DARQ nodes to obtain a global dependency graph. Upon failure, some speculatively executed steps are lost, and to restore the system to a consistent state, one must explicitly roll back any steps that depend on lost steps. In our solution, the coordinator node is responsible for detecting failures (through heartbeats, etc.), computing the rollback set from the dependency graph, and orchestrating such a rollback with all affected DARQ nodes. Importantly, if a step, and all of its dependencies are persistent, it will never be rolled back, and is therefore considered *stable*. The dependency graph also allows the coordinator to compute the set of stable steps, and external observers may choose to only observe effects from stable steps if they do not wish to be exposed to the complexity of rollbacks. For such observers, the speed-up of speculation stems from parallelism of persistence –

a speculatively executed message may be persisted in parallel with its dependencies, instead of waiting on them first.

To summarize, general speculation significantly reduces the persistence overhead of DARQ; in return, doing sorequires additional resources for orchestrating dependency tracking and failure recovery; effects of failures are amplified by general speculative execution, as the loss of a DARQ node may cause all of its dependents to also restart. That said, as we show in Section 6, general speculation tends to lead to significant speed-ups in scenarios where failures are infrequent. The key challenges in our implementation involve low-overhead dependency tracking, efficient computation of stable and rollback sets, coordinator selection and maintenance, and various concurrency conditions in distributed, asynchronous systems. The details of our solution is beyond the scope of this paper, and we leave this discussion for future work.

5 APPLICATIONS AND EXPERIENCE

In this section, we demonstrate the utility of DARQ by sketching how we build various resilient cloud applications using DARQ, and provide a brief experience report.

5.1 Case Study: Stream Processing

As mentioned in Section 2, each stream operator (e.g., map, aggregate) can be thought of as a processor performing CReSt steps, receiving data items as message from upstream and sending results downstream. We now sketch the implementation of several representative types of stream operators in DARQ with a focus on practical performance considerations.

Stateless Operators: Stateless operators such as maps and filters are typically early in a stream processing pipeline and must support high ingestion rate. A naïve implementation using DARQ would perform a step for every input message, which fills the DARQ log with completion records and limits throughput under high input rate. An efficient implementation can offset this with *batching*, waiting until several messages have been processed to submit a step that consumes them as a unit, generating only one completion record in addition to outgoing messages [32]. As expected, there are trade-offs developers should consider: larger batch sizes lead to higher throughput but increases processing delay as processing of previous messages is not exposed until later messages arrive.

Stateful Windows: Window functions group stream events based on (typically) time and output some aggregate for each window. Given a windowing scheme, a DARQ processor maintains a list of currently open windows, and continuously incorporates events into the partial aggregate for each window. Assume for the sake of illustration that we are implementing a tumbling window with a short window length t and some small slack ($<< t$) to account for out-of-order events. Then, the local state of a DARQ processor consists of at most two currently open windows, their starting and ending timestamp, and the (partial) aggregate. In our implementation, DARQ performs a step only when a window closes, at which point DARQ consumes all processed messages, persists local state, and emits the aggregate. This is similar to batching in stateless operators, except that window lengths serve as natural batch boundaries. Larger window sizes may warrant intermediate steps that consume previous messages but yield only a self-message to reduce replay time on recovery.

Aggregates: Lastly, consider stream operators that continuously accumulate local state from an input stream (without necessarily outputting a stream, similar to tables in KafkaStreams [9]), often with larger local state than window operators. For example, one such operator may group input events by some key and count the occurence of each. As described in Section 3, processors can persist large state by using DARQ as a write-ahead log, sending self-messages of incremental state deltas. Similar to before, our aggregate operator implementation performs batched steps, either based on the number of events or time elapsed. A batched step consumes all processed messages

and yields a small delta record as self-message. On failure and recovery, DARQ streams such self-messages to the processor so it can replay and restore its state. Users use the checkpointing mechanism outlined in Section 3 to periodically compact delta records and reduce replay time on recovery.

Example Application and Discussion. Consider a search trend alert system in the cloud – each data item in the input stream models a search request, including the string search term, a user id, the user's IP address, and the timestamp at which the search was issued. We are provided with a list of terms of interest, and want to monitor and receive alerts if certain terms see an unexpected surge in search frequency (e.g., a surge in searches for fever remedy in a region may be of interest to relevant public health officials). In stream processing terms, this task can be split into three processors:

- *Preprocessing.* This processor filters the input data based on search terms, and maps each term to a coarse-grained region code using the IP information.
- *Windowed Aggregate.* This processor counts the number of each relevant search term for each region using a tumbling window based on event time.
- *Anomaly Detection.* This processor maintains local state about past window outputs and runs an anomaly detection function to decide if an event should be emitted.

We implement this application in both DARQ as described earlier and KafkaStreams [23, 68], which is widely used in the industry and offers exactly-once stream processing on top of the equally popular Kafka system [7]. In our experience, both implementations took around 10 developer-hours and a few hours of performance tuning and debugging. Note here that DARQ is more directly comparable to Kafka as the underlying storage and messaging system, rather than KafkaStreams, which is a user library. The core DARQ stream processing implementation consisted of 280 lines of C# code, whereas the KafkaStreams implementation consisted of 172 lines of Java code. Although the two projects are not directly comparable (due to differences in languages and libraries), this serves to show that building streaming application with DARQ is similarly straightforward as with a specialized library. Compared to Kafka however, DARQ-based stream processing is more composable under the CReSt model. For example, DARQ-based resilient workflows may directly invoke DARQ-based stream processing without loss of resilience, whereas workflow systems must treat Kafka as an external system and apply additional failure-handling logic. Such composability is possible because CReSt participants orchestrate resilience with each other through a minimal message-passing interface rather than through more complex mechanisms such as multi-participant transactions. In this sense, it is possible to make Kafka similarly composable by providing a CReSt wrapper around it, but as we show in Section 6, Kafka's support for its richer feature set has performance overheads compared to a minimalistic implementation of CReSt such as DARQ.

5.2 Case Study: Resilient Cloud Workflow

Cloud workflows chain multiple service invocations together into a larger program resiliently. Compared to stream processing, workflows are typically less demanding in raw throughput, but put more emphasis on flexibility and generality of the programming model. In the remainder of this subsection, we first show how to implement a simple workflow with DARQ from scratch; then, we show how existing feature-rich workflow systems such as Azure Durable Functions [15] can be modified to run on DARQ.

Simple Workflow. Consider a simple case where the workflow is encoded as a static task graph of idempotent tasks (passing results as messages along the edges). Assuming each task of the workflow executes on a CReSt participant (the host) implemented with DARQ, there are two primitives that DARQ must support: a fork operation that spawns tasks after a previous task is finished, and a join operation that starts a task after all previous tasks are finished. To fork a task in

DARQ, the participant sends completion messages to the hosts of all subsequent tasks. To join tasks, the host of the subsequent task is first loaded with dependency information (e.g., the task to start C is dependent on the completion of previous tasks A and B). Upon receiving the completion message from the fork primitive, the host internally updates the dependency set of assigned tasks, and starts the task if all dependencies are completed. To ensure fault tolerance, each task corresponds to a step – at the end of the task, the host performs a step that consumes completion message from previous tasks and emits completion message to downstream tasks. The atomicity of this step ensures that either all tasks are eventually started, or that none of them are started and the initial task is replayed. For long running tasks, developers can optionally perform intermediate steps that consumes completion messages from previous tasks and update internal host state to mark the task as started (potentially along with any intermediate state so tasks do not restart from scratch on recovery).

Azure Durable Functions. DF has a frontend that translates user code into workflows, implements higher-level abstractions such as critical sections, and schedules them dynamically with load-balancing. The underlying resilient infrastructure is represented as a series of atomic computation units under the serverless message-passing model [30, 31], which prescribes a series of stateful instances each fetching, executing, and committing work items from instance-local queues. Thus, DF can map directly to CReSt, with stateful instances as participants and work items as steps.

Note that, because instances are fine-grained, it is often advantageous to group them into coarser partitions, and treat each partition as a CReSt participant. This is how the Netherite backend for DF operates [30]. In contrast to DARQ, Netherite relies on external persistent event queues (e.g., Azure EventHubs), a persistent key-value store for participant state, and a local commit log. DARQ replaces all of these infrastructures and allows for DF to directly work on top of DARQ through a straightforward translation layer.

To validate that DARQ is suitable for DF workloads, we extract *traces* by instrumenting real DF runs and build a DARQ application to execute these traces, bypassing the complex DF frontend. These traces encapsulate each processing step DF took at each Netherite workers, including the messages sent and received, their size, any local instance state update, and the amount of time user code took to execute as part of the DF step, capturing all the dynamic scheduling and load-balancing performed by the DF frontend. We use this prototype as part of our evaluation scheme in Section 6 to compare the efficiency of DARQ against Netherite, and leave a more thorough integration with DF as future work.

6 EVALUATION

We study the performance of DARQ and DARQ-powered systems in comparison to existing solutions under various configurations. We also evaluate the performance of DARQ using a series of microbenchmarks. Specifically, we focus on answering the following research questions:

- Do DARQ-powered systems achieve resilience with comparable or improved performance compared to hand-crafted solutions?
- Does speculative execution improve application latency?
- Is DARQ scalable and economical as a cloud building block?
- Can DARQ tolerate failures and deliver acceptable recovery performance under the fail-restart model?

6.1 Workloads and Benchmark Setup

We implement DARQ as a C# service deployed on Azure, with C# producer/processor clients. As mentioned, DARQ is built on top of the FasterLog [19] library, which works with a variety of storage backends. We experiment both with disks attached to Azure virtual machines, and the more

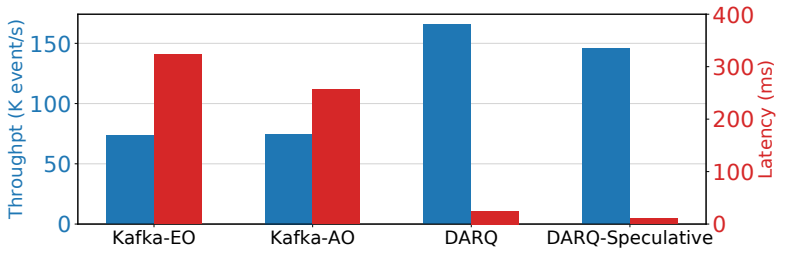


Fig. 5. Streams Latency and Throughput

economical but slower Azure blob storage [22]. Processors can either run in the same process as DARQ for performance, or be fully remote. To fully stress DARQ, We run most mirobenchmarks on two high-end L48v3 [3] machines, each with (virtualized) 48-core 3rd Gen Intel Xeon Platinum 8370C (Ice Lake) 2.8GHz processors, 384 GiB of memory and 6 × 1.92TB directly-mapped local NVMe storage. For larger benchmarks, we use a cluster of the cheaper DS14v2 [2] machines with (virtualized) 16-core Intel Xeon Platinum 8272CL 2.60GHz processors and 112 GiB of memory. All machines are colocated within the same proximity group [4] with accelerated networking enabled [18].

To understand how DARQ performance translates to real application performance, we implement the 2 prototype systems modeling stream processing and cloud workflow execution as described in Section 5, and compare them against real-world solutions currently deployed. For all experiments, we enable intra-participant speculation by default, and optionally enable general speculation when specified.

6.2 End-to-end Benchmarks

For all benchmarks in this section, we use a cluster of DS14v2 machines in conjunction with Azure cloud blobs.

Streams – End-to-end. We deploy the example stream processing application described earlier to both Kafka streams and our DARQ-based implementation. We use pre-computed, randomly-generated workloads to emit events at a configurable, fixed rate into the system for 30 seconds. We use Confluent Cloud's fully managed Kafka offering and deploy in the same Azure data center as our test machines. In Figure 5, we report the throughput and latency of the stream processing application as before, we ramp up our event issue rate until throughput no longer increases for throughput experiments, and report latencies measured when the system is at about half that load. The final consumer of stream processing results is co-located with the initial producer, sharing the same system clock for accurate latency measurement. We tune Kafka streams performance according to published best practices [10, 11], and report results from both exactly-once processing mode (EO) and the default at-least-once mode (AO). Note that all topics in Kafka are single-partition for a fair comparison with DARQ, which is supported by a single underlying log. Typical production Kafka deployments would run with more than one partition, but the same optimization can be applied to DARQ systems as well. For DARQ, we run with 5ms checkpoints to Azure blobs storage backend. We can see in Figure 5 that DARQ is competitive with Kafka stream's performance, achieving higher throughput and lower latency in general, although this may be an artifact of engineering overhead present in the more feature-complete Kafka system. Meanwhile, latency is an order-of-magnitude lower when DARQ runs in speculative mode, which presents a clear advantage over Kafka. Overall, we have shown that despite DARQ's generality, DARQ can perform on the same order-of-magnitude of performance as existing solutions in stream processing.

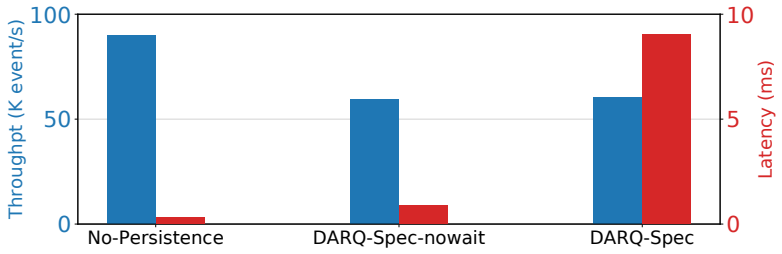
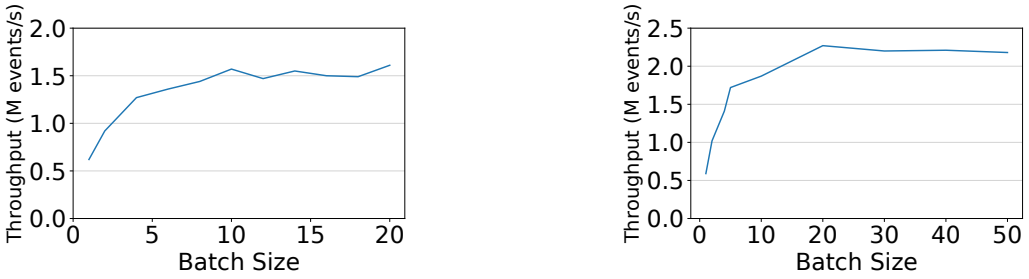


Fig. 6. Diamond-workflow Latency and Throughput



(a) Filter **(b)** Sum

Fig. 7. DARQ Stream Operator Throughput

Streams – Operators. To better understand the trade-offs in implementations of individual DARQ stream operators, we study two simplified operators: a filter operator and a sum operator. Recall from Section 5 that the key performance booster in DARQ stream operator implementation is *batch size*. For the filter processor, a batched implementation submits one step to consume input messages and output filtered messages for every n messages processed; similarly, for the sum operator, a batched implementation persists its local counter as a self-message and consume input messages for every n messages processed. To quantify this the speed-up, we pre-load 10M 8-byte integer messages onto DARQ, and measure the time it takes for DARQ to process them, varying n for each operator. We construct the workloads such that the filter operator has a selectivity of around 10%. As shown in Figure 7, batching doubles the performance of filter and almost quadruples the performance of sum. Such benefits are mostly attainable with small batches, leaving the impact of batches on replay time and latency minimal. This is because the overhead of small batches comes in the form of additional completion records, which takes up space in the log and detract from the effective I/O bandwidth used for user data. As batch size increases, the ratio of user data to completion records rapidly increases, leading to diminishing returns.

Resilient Workflows: Diamond. We first study the performance of DARQ on a simple workflow, where one task starts two other tasks, and executes an other task when both are finished – we call this the "diamond benchmark" due to the diamond shape of the task graph. Each task performs trivial work, so that the workflow stresses the orchestration system rather than compute. We schedule tasks such that each of the 4 tasks in a workflow execute on a different machine. We implement a non-resilient baseline using sockets and an in-memory task queue as the baseline, which represents the theoretical upper limit of performance in a failure-free setting. For DARQ, we run two configurations – one where DARQ runs speculatively on Azure blob storage backend, and another where DARQ returns without waiting for commit (i.e., latency for components speculatively

executing without outputing to an external uswer). We fire 500000 workflows total and measure throughput as well as end-to-end latency, shown in Figure 6. The results show that DARQ achieves about 70% of the baseline throughput, which showcases that DARQ is a lightweight solution. DARQ latency is unsurprisingly an order-of-magnitude higher than the theoretical-best baseline, but as we can see from DARQ-nowait, most of the latency stems from the slow cloud storage. With global speculative execution, DARQ-based workflows achieve comparable, sub-millisecond latency as the baseline.

Resilient Workflows: Netherite. For more complex workloads, we turn to cloud resilient workflows. For this benchmark, we use the previously described DARQ-based trace simulator to execute traces collected from Microsoft's Netherite workflow engine. In our simulator, we assign one DARQ to each Netherite worker, and pre-load the steps that worker took in the trace in-memory of the DARQ processor. Each time a processor encounters a message, it looks up the step that consumed it in the trace and checks if it has received all the messages that the step consumed; if so, the step is "executed" by sleeping for the the amount time user code took to execute in DF, and then the output messages sent. We also simulate local state update by sending it to the processor as a self-message. All messages are padded with arbitrary bytes up to the size specified by the trace. We take representative traces from the example DF applications from [30], with a "hello world" workflow that calls several tasks in sequence(*hello*), and a banking workflow that implements reliable transfer of currency between accounts using critical sections (*bank*).

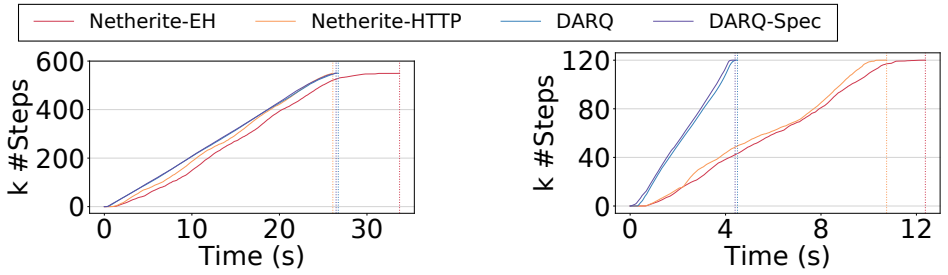
Note here that the production version of Netherite uses Azure EventHubs as the underlying messaging fabric, which is expensive. For a fair comparison, we modify Netherite to use simple http connections as the messaging fabric. We report our results in Figure 8 against the timings of the executed DF traces. In these graphs, we plot the number of completed steps over time, and plot a vertical dashed line when an engine is finished. For the hello benchmark, Netherite-http slightly outperforms DARQ, as the benchmark features no cross-partition communication, and Netherite implements lightweight single-node speculative execution. For the bank workload, which features cross-node dependency and deep invocation pipelines, DARQ significantly outperforms Netherite. In general, speculative execution does not help with throughput. We plot the latency of end-to-end workflows in the hello and bank benchmarks in Figure 9. We can see that DARQ, particularly with speculation, significantly reduces workflow latency of the bank scenario, which features cross-worker communication. For the simpler hello benchmark that runs the entire workflow on a single node in steps, Netherite's optimization around single-worker speculative execution greatly reduces its latency, outperforming non-speculative DARQ. However, with speculation turned on, DARQ is competitive with the optimized Netherite implementation. We have also run this experiment on other workloads from [30] and observe that DARQ either outperforms or is competitive with Netherite, but do not plot them here due to limited space.

To summarize, DARQ-based workflow systems have some overhead when compared to a dedicated engine like Netherite in optimized cases (e.g., hello), but offer competitive or even superior performance in cases where fault-tolerance cross-machine coordination dominates execution cost.

6.3 DARQ Service Metrics

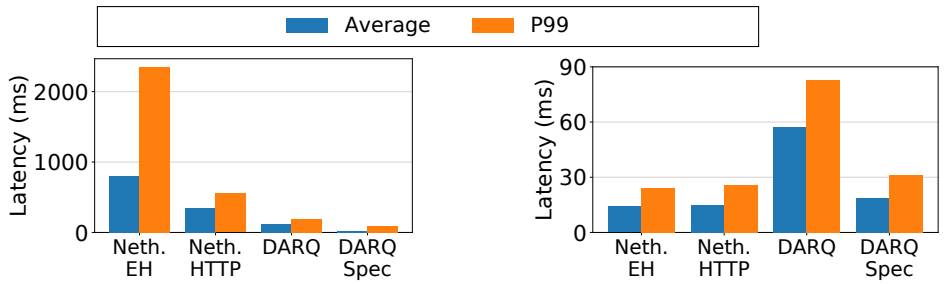
We now present a suite of benchmarks on the DARQ service itself to understand its performance, and the cost of its CReSt guarantees. All messages are randomly generated 1 KB byte arrays.

Producer Performance. We first study the performance of DARQ producers, which limits how fast DARQ can ingest work and deliver messages internally to each other. We issue 1 million enqueue requests from a single client machine that is allowed to have w requests outstanding at a given time. For throughput, we report results after incrementing w until throughput does not improve. We report our measurement results on two L48v3 machines in Table 1. As shown, DARQ can



(a) hello **(b)** bank

Fig. 8. CReSt Trace Throughput



(a) bank **(b)** hello

Fig. 9. CReSt Trace Latency

ingest around 650k operations per second, or 650 MB/s in throughput on a fast SSD. In comparison, FasterLog, the underlying log storage engine of DARQ, can achieve raw throughput of 1.59M op/s. The DARQ overhead is primarily from remote request processing and additional processing to deduplicate and validate incoming messages, and from maintaining data structures used to ensure DARQ stepping semantics. We also measure the throughput of DARQ with deduplication and validation turned off, which yields around 10% speedup on local SSD. To study the impact of storage performance on DARQ, we also run the same benchmark on Azure Blobs with a simulated storage device entirely in-memory. We can see that DARQ becomes bottlenecked by processing rather than storage performance at around 900k op/s, while on slower storage, the cost of storage dominates (around 30MB/s throughput for single blobs). For latency, we report sampled results after tuning w such that we achieve around half of the maximum throughput, to limit the impact of queueing delay on our measurements. We can see in Table 2 that DARQ achieves sub-millisecond latency for non-persistent enqueue, and less than 5 ms on average for a (batched) persistent enqueue. Note that similar to DPR operations in [50], DARQ operations are always first completed in-memory and acked, at which point speculative processors can consume the message and truncate (if any) out messages from the originating DARQ. Commit latency is the relevant metric for non-speculative DARQ processors or other producers that require confirmation of persistence. We also report latency on other storage backends, and see that slower storage (such as cloud blobs) only affects non-speculative latency, as expected.

Processor Step Performance. We now study step performance of DARQ, similarly on the performant L48v3 machines. We first measure throughput of DARQ steps by pre-loading 1 million 1KB input messages to DARQ, and processing them using a trivial processor that immediately

Storage	DARQ	DARQ No-Val	Raw FasterLog
Memory	829.74k op/s	923.56k op/s	2.75M op/s
Local SSD	641.28k op/s	676.18k op/s	1.59M op/s
Page Blob	30.17k op/s	30.52k op/s	34.54k op/s

Table 1. DARQ Producer Throughput

Storage	Median Latency	P99 Latency	Std. Dev.
SSD-Completion	0.12 ms	0.19 ms	0.89 ms
SSD-Commit	3.27ms	9.59ms	2.45 ms
Blob-Completion	0.08 ms	0.22 ms	0.13 ms
Blob-Commit	6.66 ms	18.94 ms	3.44 ms

Table 2. DARQ Producer Latency

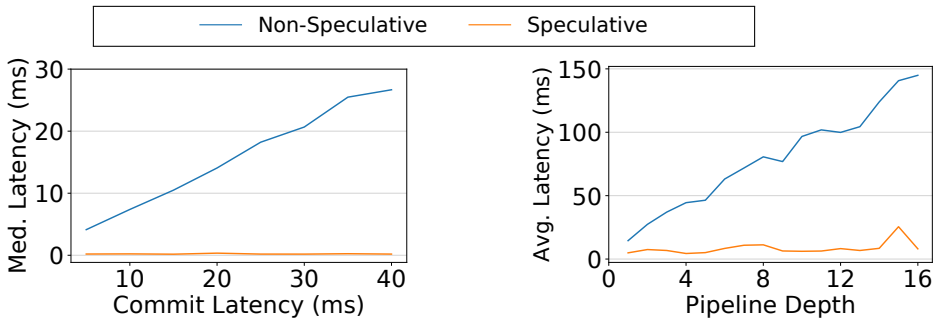
Processor	DARQ	DARQ No-Val	Raw FasterLog
Colocated-ssd	741.84k op/s	959.69k op/s	1.01M op/s
Remote-ssd	215.15k op/s	250.31k op/s	-
Colocated-blob	33.22k op/s	33.45k op/s	33.54k op/s
Remote-blob	32.32k op/s	32.83k op/s	-

Table 3. DARQ Processor Step Throughput

marks encountered messages as completed. Recall that to perform a DARQ step, the processor has to first scan the consumed messages in DARQ, compose the step locally, and then submit the step to DARQ as a local function call or RPC; DARQ needs to validate that the steps are well-formed and enforcing that it only steps previously unstepped messages. We run this microbenchmark with both a co-located processor and a remote processor, and report the results in Table 3. As seen, a co-located DARQ processor can process more than 700k operations per second, which is more than the ingestion capacity of a single DARQ. Remote DARQ processors, in comparison, see a reduced throughput of just above 200k operations per second, mostly due to remote processing overhead. We also run the same benchmarks where we disable various parts of the step and measure the resulting throughput in Table 3. Here we can see that the write-back to DARQ is the most significant contributor to overhead, and step validation only presents a small overhead. Unsurprisingly, when running with cloud storage, storage becomes the bottleneck compared to processing.

Effect of Speculation. Here, we study the effect of speculation on DARQ performance in detail. We use a microbenchmark where we enqueue 1 million messages into a DARQ and attach a co-located processor that steps by reflecting the message back to its sender (which locally hosts a "sink" DARQ), which measures the latency between sending of a message and receiving it in return. This benchmark is indicative of the cost of composition, as it corresponds to the overhead DARQ introduces between starting a step and subsequent work (i.e., when downstream DARQ processors are able to process the output messages). We run the same benchmark under two scenarios – one where we increase the delay between each commit to simulate slow storage, and one where we generalize the reflection benchmark to route messages through more DARQs before eventually returning to the sender. We show the results in Figure 10 as measured on the weaker but more numerous DS14v2 machines. As expected, speculative execution essentially "parallelizes" storage overhead, and performs much better under high storage latency or when with many storage roundtrips. The cost of speculative execution is in the complexity of engineering and in recovery performance, which we study later.

Scalability Discussion. Even though DARQ supports high throughput, it is a serial bottleneck due to its log-structured implementation. We have not found this to be an issue in our experience, as in many applications, steps are computed and such computations are more likely to become a



(a) Varying Device Latency

(b) Varying Pipeline Depth

Fig. 10. Effect of Speculation on Latency

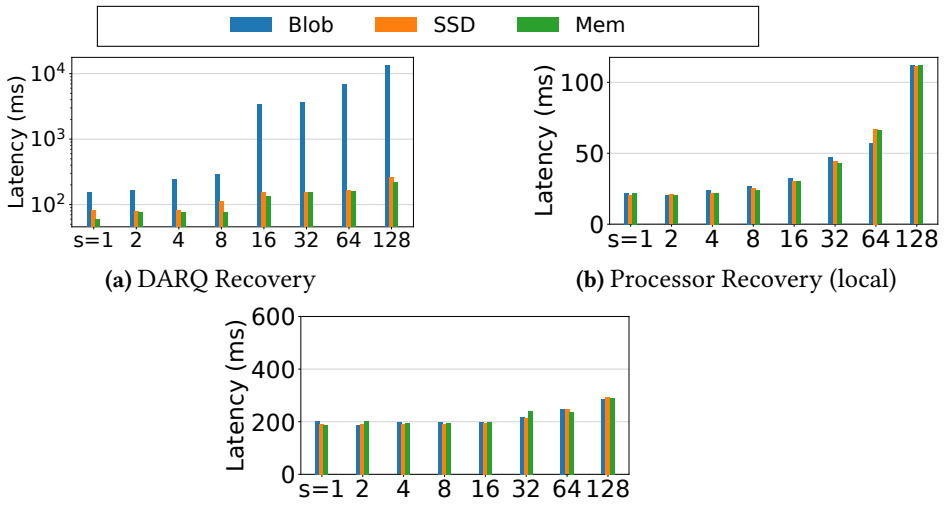
bottleneck before raw I/O performance of DARQ. In the case where applications do bottleneck on DARQ throughput, developers can resort to adding parallel devices/faster storage to boost I/O performance, or structure their applications to benefit from multi-DARQ parallelism.

6.4 Failure Recovery

Lastly, we study the performance of DARQ under failure.

Microbenchmarks. We first study the performance of DARQ recovery in controlled microbenchmarks on L48v3s. In these experiments, we randomly populate DARQ with some amount of steps, and then manually recover from the checkpointed state and measure the time it takes before the processor is ready for normal operation (the first message reaches user code for consumption). Note that this time does not include any time the user processor may take to reconstruct its local state from DARQ as that is application-dependent. We show the results in Figure 11 varying the size of the log to recover (measured by scale factor of the generation process, one scale factor corresponds to roughly 40MB of log to replay). We first study recovery when the DARQ service is down and must be reconstructed from storage. We can see that recovery speed here is largely dependent on the speed of the underlying storage — for slower Azure blob storage, it takes significantly longer to recover than fast, in-memory simulated storage. Overall, however, recovery speed is linear with the size of the log to replay, and as we will show later, the average size of the log to replay in a realistic application tends to be very small. Another scenario is when the DARQ service does not fail, but attached processors fail or are replaced. In this scenario, replay is still necessary to reconstruct processor local state (recall that processors expect self-messages upfront and only incomplete messages on replay), but it is unnecessary to replay from storage, as the tail is likely to be in memory in the DARQ service. Consequently, as shown, recovery performance is in general better than DARQ failures, and is no longer dependent on storage backend performance.

End-to-end Benchmarks. To see recovery in action, we use the stream processing benchmark from before and introduce a failure in the middle of an extended (90s) run on DS14v2s. We show results of failure from both speculative and non-speculative modes. For brevity, we only show results of failure on the second processor (windowed aggregator); failures on other processors yield similar results but at a slightly different scale due to the difference in steady-state throughput at each stage of the stream processing. We show our results when an entire DARQ node is killed in Figure 12. Here we can see that for both configurations, there is a latency spike when a failure occurs, as we wait for the DARQ service to re-establish itself and restore all previous connections, which takes around 300ms in our measurement. The service is then quickly back to normal for the non-speculative DARQ. For speculative DARQ, the system performs extra work for a DPR

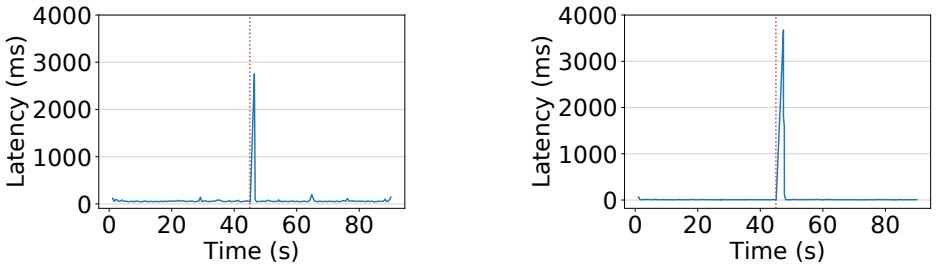


(a) DARQ Recovery

(b) Processor Recovery (local)

(c) Processor Recovery (remote)

Fig. 11. Recovery Microbench



(a) Non-Speculative

(b) Speculative

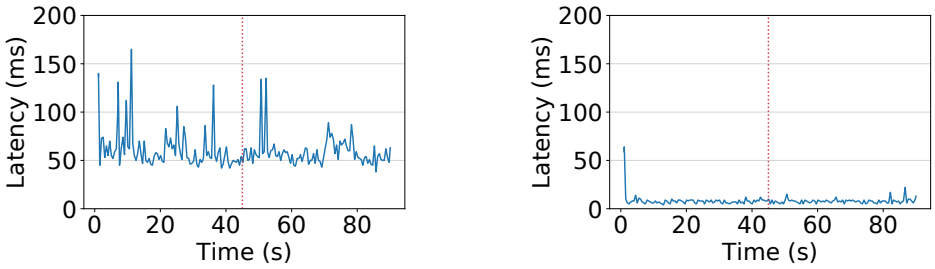
Fig. 12. Recovery from DARQ restart

rollback of other processors, which causes both the latency spike to be larger, and for recovery to take longer.

We also show DARQ performance when introducing a processor failure in Figure 13. As there is no need to restart the DARQ server, processor recovery is merely a matter of replaying the outstanding portion of the log, which only takes a few milliseconds, and is barely noticeable on the plot. This serves to show that DARQ-powered systems can tolerate compute node churn quite well without excessive overhead. To better understand the outstanding log size during our run, we sample untruncated DARQ size and show the results in Figure 14. We can see here that our aggressive GC scheme (which truncates the log at 1MB granularity) successfully limits the replay size to around 1.5 MB throughout the benchmark.

7 RELATED WORK

Resilient Systems. The concept of end-to-end application resilience in database systems dates back to [28] and [27]. Ambrosia uses DBMS-style logging to provide resilience for general-purpose message-passing distributed programming; however, the Ambrosia solution is built as sidecars [24]



(a) Non-Speculative **(b)** Speculative

Fig. 13. Recovery from processor restart

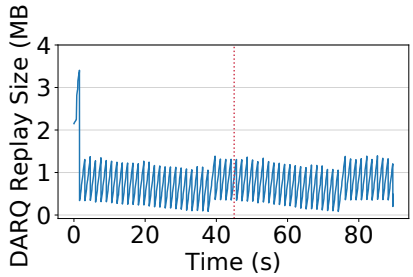


Fig. 14. DARQ replay size during benchmark

to applications running on standalone machines. CReSt (and by extension, DARQ) is inspired by Ambrosia, but instead uses an abstract I/O automata model to define resiliency. The DARQ solution focuses on the service-oriented cloud model and is a storage service, distinguishing it from the Ambrosia model. Other than Ambrosia, Netherite [30] and Temporal [21] similarly build resiliency into message-passing systems, but are more specialized towards their respective programming model, directly interfacing with application developers. Recent work [65, 72] have proposed modeling workflows as dataflow problems, which makes it possible to repurpose existing work for exactly-once stream processing for resilient workflows. In contrast, CReSt and DARQ are geared towards infrastructure builders, who will likely use DARQ to build systems like Netherite, Temporal, and stream processing. There has also been implementation of virtual resiliency at the OS/VM level [34, 36, 52] but these works are typically much more low-level and therefore rely on high-cost physical logging, whereas CReSt and DARQ only log logical steps and transitions.

Transactions, Queuing, and Streaming Systems. Another way to achieve resilience in cloud systems is through multi-node transactions [35, 41, 49]. One might argue that CReSt is a form of transaction in that it is an atomic, multi-operation primitive. We contend that while this is conceptually true, CReSt is a highly *specialized* form of transactions for the use case of cloud service composition. Most importantly, CReSt treats distribution and asynchronous communication as first-class concepts in its formulation, making it more natural for existing distributed systems to be modeled using CReSt. The inclusion of asynchrony in the CReSt model also allows for implementations of CReSt that does not require two-phase commit, as general transaction implementations do. Recent systems such as Apiary [45] and DBOS [63] attempt to build distributed systems on top of a distributed transaction processing layer, which would allow disparate applications to perform ACID transactions across them, sidestepping the problem CReSt and DARQ solve. Other

systems [61, 66, 71] similarly use transactions to orchestrate virtually resilient executions between components; these systems are generally easier to work with than CReSt or DARQ, but are less general as a result, as developers must extend their transaction mechanisms to cover any new components. DARQ itself is inspired by classical (transactional) queuing systems [29, 42, 55], and more recently, streaming systems such as Kafka [68] and Azure EventHubs [25]. Of particular note is [29], which proposed using queues that orchestrate transactions with databases to ensure recoverable and resilient client request handling. CReSt similarly relies on atomicity between queues and some other data store, but chooses to co-locate queues with the data store on a single node as the atomicity mechanism instead of transactions. DARQ's log-based implementation is also largely inspired by earlier write-ahead logging techniques [54, 56] from the database community, and recent log-based solutions for distributed systems [26, 43].

Optimistic/Rollback-Replay Recovery. To our knowledge, speculative execution in distributed systems is first proposed by [58]. DARQ's reliance on speculative execution can be viewed as a class of optimistic recovery [67]. We make heavy use of the recent DPR work [50], which in turn builds on a wealth of prior work of rollback-recovery systems [37]. Such solutions are also implemented recently in dataflow [38], streaming [62], and actor systems [69] to great effect.

8 CONCLUSION

Implementing fault-tolerance is increasingly important yet difficult and error-prone in today's highly distributed and disaggregated cloud environment. We presented CReSt, an abstraction for building resilient cloud systems that extends the classical message-passing model with resilient steps. We argue that many fault-tolerant cloud programming paradigm, such as stream processing and workflows can be mapped to CReSt steps. We also design and implement DARQ, an efficient framework that implements CReSt optimized for the cloud. Our benchmark shows that DARQ-powered applications achieve resilience, are competitive in performance, and are easier to build than from scratch. We believe CReSt and DARQ to be valuable tools in constructing fault-tolerant modern cloud applications.

REFERENCES

[1] Amazon Step Functions. https://aws.amazon.com/step-functions/, retrieved 13-Oct-2022.
[2] Dv2 and DSv2-series. https://learn.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series, retrieved 13-Oct-2022.
[3] Lsv3-series Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/lsv3-series, retrieved 13-Oct-2022.
[4] Proximity Placement Groups. https://learn.microsoft.com/en-us/azure/virtual-machines/co-location, retrieved 13-Oct-2022.
[5] Saga Pattern. https://microservices.io/patterns/data/saga.html, retrieved 13-Oct-2022.
[6] Transactional Outbox Pattern. https://microservices.io/patterns/data/transactional-outbox.html, retrieved 13-Oct-2022.
[7] Apache Kafka. https://kafka.apache.org/, retrieved 15-Jan-2023.
[8] Kubernetes Pod Lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, retrieved 18-Jan-2023.
[9] Streams and Tables in Apache Kafka: A Primer. https://www.confluent.io/blog/kafka-streams-tables-part-1-event-streaming/, retrieved 19-Jan-2023.
[10] Confluent Developer – Optimizing for Latency. https://docs.confluent.io/cloud/current/client-apps/optimizing/latency.html, retrieved 26-Sept-2022.
[11] Fine-tune Kafka performance with the Kafka optimization theorem. https://developers.redhat.com/articles/2022/05/03/fine-tune-kafka-performance-kafka-optimization-theorem#, retrieved 26-Sept-2022.
[12] Amazon Lambda. https://aws.amazon.com/lambda/, retrieved 28-Aug-2022.
[13] Amazon S3. https://aws.amazon.com/s3/, retrieved 28-Aug-2022.
[14] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/, retrieved 28-Aug-2022.
[15] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, retrieved 28-Aug-2022.

[16] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, retrieved 28-Aug-2022.
[17] Azure Service Fabric. https://azure.microsoft.com/en-us/services/service-fabric/, retrieved 28-Aug-2022.
[18] Create a Windows VM with accelerated networking using Azure PowerShell. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-powershell, retrieved 28-Aug-2022.
[19] FasterLog and the Microsoft FASTER project. https://github.com/microsoft/FASTER, retrieved 28-Aug-2022.
[20] Kubernetes. https://kubernetes.io/, retrieved 28-Aug-2022.
[21] Temporal. https://temporal.io/, retrieved 28-Aug-2022.
[22] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/, retrieved 30-Aug-2022.
[23] Kafka Streams. https://kafka.apache.org/documentation/streams/, retrieved 30-Aug-2022.
[24] Sidecar Pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar, retrieved 30-Aug-2022.
[25] Azure Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/, retrieved 31-Aug-2022.
[26] M. Balakrishnan, C. Shen, A. Jafri, S. Mapara, D. Geraghty, J. Flinn, V. Venkat, I. Nedelchev, S. Ghosh, M. Dharamshi, J. Liu, F. Gruszczynski, J. Li, R. Tibrewal, A. Zaveri, R. Nagar, A. Yossef, F. Richard, and Y. J. Song. Log-structured protocols in delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 538–552, New York, NY, USA, 2021. Association for Computing Machinery.
[27] R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery guarantees for internet applications. *ACM Trans. Internet Technol.*, 4(3):289–328, aug 2004.
[28] R. Barga and D. B. Lomet. Phoenix: Making applications robust. *SIGMOD Rec.*, 28(2):562–564, jun 1999.
[29] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. *SIGMOD Rec.*, 19(2):112–122, may 1990.
[30] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, apr 2022.
[31] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.
[32] 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.
[33] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.
[34] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 161–174, USA, 2008. USENIX Association.
[35] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, page 113–122, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
[36] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery.
[37] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002.
[38] I. Gog, M. Isard, and M. Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 373–387, New York, NY, USA, 2021. Association for Computing Machinery.
[39] J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, R. G. Peshawaria, T. Zaccai, and I. Zhang. A.m.b.r.o.s.i.a: Providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, Jan. 2020.
[40] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.
[41] J. Gray. Notes on data base operating systems. In *Advanced Course: Operating Systems*, 1978.
[42] J. Gray and J. Gray. Queues are databases. In *In Proceedings 7th High Performance Transaction Processing Workshop. Asilomar CA*, page 496. Prentice Hall, 1995.
[43] 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.
[44] 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.

[45] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, P. Bailis, M. Cafarella, G. Graefe, J. Kepner, C. Kozyrakis, M. Stonebraker, L. Suresh, X. Yu, and M. Zaharia. Apiary: A dbms-backed transactional function-as-a-service framework, 2022.

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

[47] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *Proc. VLDB Endow.*, 14(13):3348–3361, sep 2021.

[48] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[49] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. 1976.

[50] T. Li, B. Chandramouli, J. M. Faleiro, S. Madden, and D. Kossmann. *Asynchronous Prefix Recoverability for Fast Distributed Stores*, page 1090–1102. Association for Computing Machinery, New York, NY, USA, 2021.

[51] T. Li, B. Chandramouli, and S. Madden. Performant almost-latch-free data structures using epoch protection. In *Data Management on New Hardware*, DaMoN'22, New York, NY, USA, 2022. Association for Computing Machinery.

[52] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[53] 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.

[54] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014.

[55] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Guenthoer, and M. Kamath. Exotica: A project on advanced transaction management and workflow systems. *SIGOIS Bull.*, 16(1):45–50, aug 1995.

[56] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, mar 1992.

[57] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, dec 1986.

[58] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. SOSP '05, page 191–205, New York, NY, USA, 2005. Association for Computing Machinery.

[59] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[60] J. Postel. Rfc0793: Transmission control protocol. Technical report, 1981.

[61] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 501–516, USA, 2016. USENIX Association.

[62] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1637–1650, 2021.

[63] 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.

[64] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. In *Proceedings of the 5th International Conference on Database Theory*, ICDT '95, page 139–147, Berlin, Heidelberg, 1995. Springer-Verlag.

[65] J. Spenger, P. Carbone, and P. Haller. Portals: An extension of dataflow streaming for stateful serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, page 153–171, New York, NY, USA, 2022. Association for Computing Machinery.

[66] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[67] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.

[68] 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.

[69] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 338–352, New York, NY, USA, 2019. Association for Computing Machinery.

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, page 2, USA, 2012. USENIX Association.

[71] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[72] W. Zorgdrager, K. Psarakis, M. Fragkoulis, E. Visser, and A. Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. *CoRR*, abs/2112.00710, 2021.

Received October 2022; revised January 2023; accepted February 2023