Consistency and Correctness in Data-Oriented Workflow Systems

Michael Stonebraker
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
stonebraker@csail.mit.edu

Xinjing Zhou
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
xinjing@mit.edu

Peter Kraft
DBOS, Inc.
Sunnyvale, California, USA
peter.kraft@dbos.dev

Qian Li
DBOS, Inc.
Sunnyvale, California, USA
qian.li@dbos.dev

Abstract

Enterprise applications increasingly organize computation as workflows that must recover from failures and handle diverse error conditions. Although many developers can write and test a saga, few get it right when the server crashes. Durable execution is the missing piece: it guarantees exactly-once execution of workflow steps and ensures that compensations actually run, even in the face of failures. However, durability alone is not sufficient.

In this paper, we outline the semantic guarantees required for correct data-oriented workflows and argue that ACID must be extended from individual transactions to entire workflows, making them atomic, consistent, durable, and correct (AC/DC). We present a prototype database-oriented workflow system that implements durable execution and supports both physical backout and saga-style compensation with minimal programmer effort. Experiments on an e-commerce workload show a clear trade-off: transactional workflows win under low contention, while sagas deliver higher throughput and avoid aborts under contention or long-running steps. We conclude that durability is essential for making sagas correct, but that full AC/DC guarantees are needed to substantially reduce workflow complexity.

ACM Reference Format:
Michael Stonebraker, Xinjing Zhou, Peter Kraft, and Qian Li. 2025. Consistency and Correctness in Data-Oriented Workflow Systems. In *Proceedings of Conference for Innovative Data Systems Research (CIDR '26).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Every enterprise we know is moving everything they can to the cloud as aggressively as possible. They have concluded that cloud deployment is cheaper than on-prem solutions. The pricing model from the major cloud vendors shows the desirability of deploying one's applications in a "serverless" manner. Using this model, there is no continually running application server, which is charged for cloud resources whether it has client work to perform or not. Although one can imagine various architectures for achieving serverless computing, the most promising one, in our opinion, is to structure one's application as a graph of workflow "steps" (microservices) along the lines of AWS Lambda [1] and AWS Step Functions [2]. Steps share "state" either as part of the call to the "next in line" or through persistent storage (e.g. S3). This is often called a platform-as-a-service (PaaS) model [8]. With this paradigm, the platform is responsible for "orchestration", i.e. waking up a step when its inputs are available, as well as for load balancing and scale up/down. Also, there are no resources consumed when a step is waiting for input; hence an application is only charged when a step is running.

In this paper we indicate the desired semantics to be provided by the platform for data-oriented workflow applications. We assume a workflow is a computing program that is a directed graph of steps, with a single entry step and a single termination step. In the middle there can be arbitrary fanout, which allows parallel execution. A step is a function in some language (think Python, Typescript, Java,...) perhaps with updates to persistent storage (think SQL, an ORM, or file operations). The function takes a collection of arguments and returns a value or a collection of values.

In our opinion a single step should be a transaction (or a collection of transactions) and the platform (or an associated DBMS) should guarantee ACID properties for transactions. For simplicity we assume a step is a single transaction, although the generalization to multiple transactions is straightforward. Also, this discussion applies equally to on-prem platforms as well as cloud-based ones.

For example, in Figure 1, we show an example of an e-commerce checkout workflow for an online store. The logic is straightforward: each workflow step entails a DBMS transaction. Some steps are to local databases (inventory) and some are to external services (process payment). Of course, it is possible that some of the steps entail an AI agent, for example looking up any previous transactions from this buyer and deciding if s/he returned too many previous purchases and is therefore a poor customer. Workflows that entail AI (e.g. Large Language Model) steps are often called agentic AI applications.

In real life, the example workflow would usually be more complex, for example deciding from which warehouse to service the request. Also, it may entail a directed graph of workflow steps instead of a linear one. However, we will stay with the simplest version as an example. One can imagine this "forward" logic fitting on a page or two of code.

CIDR '26, Santa Cruz, CA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

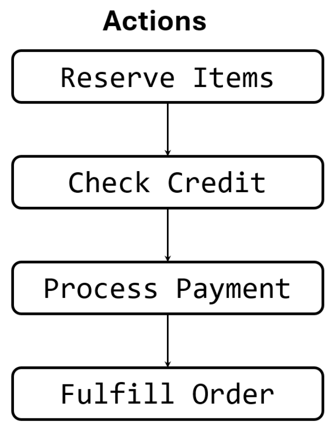

Figure 1: A simplified example of an e-commerce checkout workflow.

However, the complexity of this application comes from the error logic and from the "oops" logic. Each step in the workflow can fail. The desired item may be out of stock, the credit card may be declined, and the customer may give a bad address for fulfillment. The error logic for each step must be included in the application. For example, a bad address for fulfillment may require the whole workflow to be unwound.

Moreover, a resilient application also requires "oops" logic to handle unexpected failures. If the seller's site crashes, then they have to pick up the pieces and make sure there are no partially complete sales. The site may become overloaded and response time may become intolerable. In this case the buyer may go away, leaving an incomplete transaction. In Figure 2, we expand the checkout workflow to show both forward and error-handling logic.

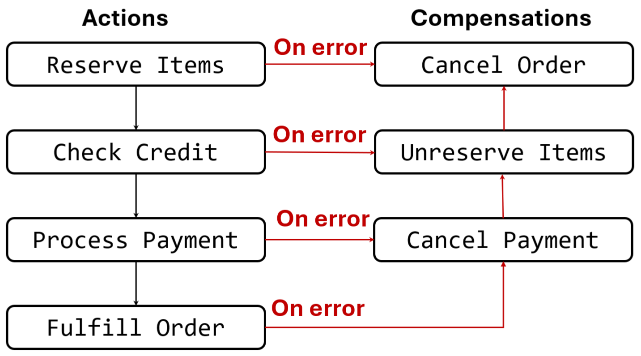

Figure 2: A simplified example of an e-commerce checkout workflow, highlighting the compensating actions that execute on error.

In our experience, the forward logic is 10% of the application while the error and oops logic are 90%. The rest of this paper deals with reducing the amount of this code by using platform services. We first consider semantic guarantees for the forward direction of workflow execution (the 10%), followed by considerations for the reverse (error) direction (the 90%).

2 FORWARD DIRECTION: DURABLE COMPUTING

With these preliminaries, a durable computing platform guarantees two things. First, it ensures that any updates to persistent state for a completed step as well as the parameters to 'the next in line' are durably saved in storage. Second, in case of a failure (machine failure, deadlock, etc.), the software managing the workflow will guarantee to resume from its last known state. Hence, it does not need to restart workflow instances from the beginning, thereby minimizing the amount of work that needs to be redone. Also, it guarantees once-and-only-once (OAOO) execution, as no completed step will be repeated. It also guarantees that workflows are completed in the absence of application failures. In simple terms, one needs to checkpoint relevant information from each step, so that the workflow can be appropriately resumed in case of a failure. There are several commercial products that advertise durability [3, 10, 12].

Notice that a durable computing platform extends the D in ACID, from a single transaction to a workflow, when platform interruptions occur, either through hardware or platform errors.

A popular current market for durability is agentic AI, where a collection of (usually long-running) steps is performed, directed by commands from a large language model (LLM). If something bad happens, the workflow can be recovered up from its last completed step, saving the time required to repeat computations. At present, agentic AI applications are largely aggregating information from multiple sources and returning some sort of predictive result. As such, they usually do not update persistent storage. As long as this is true, durability is the only required service. However, we expect agentic AI applications to expand into update-oriented ones. For example, one can imagine most of the steps in our example e-commerce application would benefit from leveraging AI. In read-write applications, whether agentic or not, durability must be augmented with other services, to which we now turn.

3 DURABLE COMPUTING IS NOT ENOUGH

In the previous section we discussed the forward direction of an application, where it makes progress from beginning to end. If there is an interruption, the platform resumes the workflow from its last completed step. However, most workflows must deal with errors where the workflow cannot be completed as planned. In the web sales example above, we noted some of the errors which can occur.

Each possible error requires some resolution. In some cases, a message to the user can resolve the situation, for example asking the orderer for a different credit card. The workflow can then proceed. In other cases, there may be no resolution, for example bad credit or overuse of returns. In this case, the workflow must be aborted. An aborted workflow will usually leave the associated database(s) in an inconsistent state. For example, there may be reserved inventory that must be returned to the available state.

Durable computing deals only with the forward direction; other facilities are needed to deal with errors. The rest of this paper deals with these additional facilities.

4 ERROR HANDLING MECHANISMS

Much of the time an aborted workflow leaves persistent data in an inconsistent state. Obviously, this is a bad outcome. Of course, one solution is to do nothing, and let a human figure out how to pick up the pieces. A much better solution is for the platform to support mechanisms for handling errors and guaranteeing atomicity.

4.1 Atomicity

Atomicity in a transaction system guarantees that the transaction either finishes or looks like it never happened (the A in ACID). In a workflow world, as we will see later, there may be alternate paths to completion when errors occur. As such, every workflow instance must complete, perhaps on some other forward path or look like it never occurred. Because this is a very similar notion of atomicity to the one in transaction systems, we will use the same word for this all-or-nothing behavior.

In our opinion, every workflow platform should support atomicity in addition to durability. To support atomicity, multiple additional facilities are required. We start with the several actions that make sense, and a directive to indicate which one(s) to use.

4.2 Alternate Steps and Directives

If the first warehouse in our example task is out of stock, then one may wish to check a different warehouse. In effect, the system needs to take an alternate step if the first step fails. Of course, it is possible to have this logic inside a given step. However, we posit this is a bad idea. First, this makes a step possibly take a great deal more time than it would otherwise, which may have throughput and latency issues discussed below. In addition, if one wants to subsequently change the logic of backup warehouses, then this will require code maintenance inside a step. In my opinion a much better solution is to define the alternate action as a step, and then have a directive associated with each step, for example, *"On failure, perform Step X"*.

We assume that each step is well tested and there are no program bugs. Hence, errors are bad outcomes of actions. They may result from deadlocks or violation of integrity constraints, or updates not accomplished (for example, a record to be updated was not found). As a result of an error, we believe there are four reasonable directives:

- Retry – in case the error is transient
- Backout
- Perform an alternate (replacement) action — for example, try another warehouse if stock is depleted in the primary one. In this case, there may need to be input from a human, for instance to say which secondary warehouse to try. Of course, there may be a sequence of alternate actions to try until one succeeds and the sequence ends.
- Manual resolution. The error may be sufficiently complex that automatic resolution is not possible. A human will have to figure out how to complete or backout the workflow.
- Do nothing. In our opinion, this is never a reasonable thing to do, because it leaves the workflow incomplete and likely in an inconsistent state.

A directive can be a sequence of remedies; for example, *Retry X times with exponential backoff else perform an alternate step.*

The directive for this alternative step must indicate what to do in case it fails. After a sequence of such steps, one step must succeed or call for a backout. Otherwise, the workflow instance never completes, and atomicity fails. We now turn to the mechanism for backing out an incomplete workflow.

4.3 Backing Out Workflows

There are three possible mechanisms to back out an incomplete workflow.

4.3.1 *Physical Backout.*

One fail-safe strategy is to run the entire workflow as a transaction with savepoints for each step. The downside of this approach is, of course, that locks will be held for the duration of the workflow.

Alternatively, if one can guarantee that no other workflow or transaction touches the items touched by this workflow, then physical backout from a log will work correctly without long-term locks. For example, If every workflow instance has a primary key (employee name, ID, etc.) and every update has a predicate with this ID, and the application is willing to guarantee that two workflows with the same primary key are never running concurrently, then physical backout will always succeed. There may well be other special cases.

In summary, either workflow-duration locks or a user guarantee of non-conflict is required.

4.3.2 *Compensation.*

Unlike a transaction system which can back out an aborted transaction by physically undoing its updates, aborting a workflow often requires a more complex operation. For example, if inventory must be made available again, one cannot simply physically adjust the value back to its pre-workflow state. If another workflow instance has made a reservation for the same item in the meantime, then returning to the pre-workflow value will cause data loss for intervening reservations.

Instead, a compensation [4] action must be performed. For example, the compensation for decrementing an item is incrementing it. For some actions, the compensation can be automatically determined, as in the example above. For others, such as giving an employee a 10% raise with an intervening action of a $5 raise, the compensation is more complex. As a result, there are compensations that can be automatically applied and others that require a user-defined compensation.

A user must assert that workflows with user-defined compensations are consistent. A consistent workflow is one which will return the state of persistent storage to its pre-workflow state when run in single-user mode.

4.3.3 *Manual Backout.*

In some cases the complexity of the backout required is not amenable to an automatic solution. In this case the system must defer to a human to pick up the pieces.

5 CORRECTNESS UNDER CONCURRENCY

A consistent workflow just guarantees a good outcome in single user mode. It says nothing about side effects caused by a backout. In general, backing out an action will occur at a noticeably different time than the action originally took place. In the meantime, the set of records qualifying for the action may have been changed by the workflow of another user. For example, a step may give an update to all employees in the toy department. Additional steps of this workflow may occur before one fails, requiring all steps to be undone. In this intervening time, another workflow instance may have added an employee to the toy department. We want the undo to be performed only on the set originally updated. This is an example of the well-studied phantom problem in transaction systems.

There are many examples of phantoms. For example, suppose one decrements an object X from a value U to a value V and an intervening step from another workflow conditionally updates X but only if it has a value greater than U. Backing out this workflow step requires cascading the backout.

In spite of this complexity, a platform should guarantee the correct outcome of parallel workflows in case of errors. A correct outcome is one that is the same as that produced by the following sequence of actions:

(1) Restore all persistent state to what it was at the time the errant workflow started
(2) Run the steps of other concurrent workflows that have finished since the errant workflow stopped.

This sequence of steps guarantees correctly backing out the errant workflow.

To perform this remediation, one needs to checkpoint all state and be able to back up time to the start of the errant workflow. Such time travel has been studied in the case of debugging [6] and can easily be extended to deal with error recovery. If one simply performs compensation, then a correct outcome is not guaranteed.

If an update is a decrement with an integrity constraint of non-negativity, then the backout is an increment and it will always succeed. For an increment, the compensation is a decrement and it will always succeed. This guarantees, for example, that electronic funds transfer can always be backed out with no cascade. We believe that workflows with only increments and decrements subject to the integrity constraints above can always be backed out.

6 PRACTICAL CONSIDERATIONS

Several practical issues arise when implementing workflow systems with the semantics described above. We discuss the most important ones in this section.

6.1 External Services

Many workflows contain calls to external services, such as Paypal or Venmo, as in our example above. These services typically accept requests and provide responses. However, if there is a failure either in the service or in the communication system between the workflow platform and the service, then the workflow platform will sometimes be uncertain whether the external request has been completed or not. When in doubt, the platform has little option besides resubmitting the request. In this case, the external service must guarantee idempotency to ensure that the request is only processed once. If it does not, then the platform (or the application) must try defensive programming.

First the platform can request the values of any items being read or updated, in "read for update mode", which will set locks in the service. These are "prior values" for the update to be processed. Then the platform can send its request. If there is a return, then the platform can record the prior and post-operation values in its log. If there is no return, then the platform can query the service again for the items in question. If they are the same as previously, then the platform can issue the request again. If they are equal to the post-operation values, then the transaction has been completed. If some other value is returned, then the platform has no idea whether the request happened or not.

Hopefully, the service can be queried whether the request was completed or not, and "can do the right thing". Otherwise, human intervention is required. Obviously, this is a mess, so most platforms assume idempotency.

If a call to an external service must be backed out, then a user-defined compensation must be run. Actions to platform databases will generally also require compensations – in the case where an intervening action from another user has occurred.

At some point in the future, services might support distributed transactions (XA) [13]. If so, the platform can implement XA and deal correctly with external transactions. Unfortunately, an external service must hold locks while distributed commit processing is occurring, resulting in a negative impact on performance. Therefore, we do not expect external services to support XA anytime soon.

6.2 Workflows with User Stalls

If one is going to backout everything until the start of an errant workflow, then there should not be any user stall in a workflow step. Otherwise, the required backout could be minutes in length. We can detect this case at compile time and break such a workflow into three pieces: the part before the stall, the step with the stall, and the rest of the workflow. Then, run the stall as its own workflow. Hence, one workflow becomes three, and we will never have to back out a workflow across a stall. This may only have a small effect if the other steps are short compared to the stall.

6.3 Steps that Cannot Be Undone

There are steps in a workflow, such as dispensing money from a cash drawer, that cannot be undone. A workflow with an irreversible step must be designed so the step in question is the last step. Also, if the cash drawer jams, then the workflow cannot be finished or undone and manual intervention is required.

Also, if fulfillment succeeds but the customer returns the merchandise, then "returns" must be a separate workflow. Otherwise, an irreversible step would not be the last action in a workflow. In other words, irreversible steps must dictate the design of multiple workflows, each having an irreversible last step.

6.4 Workflow Design Guidelines

Obviously, one should design workflow applications carefully. As noted above, one should move steps with user interventions to be separate workflows. In addition, two workflow steps that are likely to conflict with other concurrently running steps, should be combined in a single step to minimize interference. In addition, one should make workflows as simple as possible to minimize backout complexity. Of course, read-only workflows can be as complex as one wants.

7 IMPLEMENTATION IN DBOS

We now discuss the implementation of our model in the DBOS system. The academic prototype has been extensively discussed in the literature [5, 7, 11]. Here we briefly describe the commercial version of DBOS, and then we describe the DBOS implementation of durability. Finally, we describe our prototype implementation of atomicity and consistency, which is available in prototype form at https://github.com/DBOS-project/dbos-transact-py.

Commercial DBOS has focused on implementing durable execution as a lightweight library that can be integrated into existing codebases and deployed in a wide range of environments (e.g., laptops, EC2 instances, and serverless functions). It can be run with many SQL DBMSs, but herein we assume the default DBMS, which is PostgreSQL. The academic prototype supported Apiary with metadata declarative extensions for Java. In the commercial system, we now support Typescript, Java, Go, and Python. The commercial system retains a database-oriented approach to workflow management, storing all workflow state, execution history, and recovery information in relational tables. The main ones are:

(1) A workflow status table that records the execution state of each workflow instance including the current step, completion status and result values.
(2) An operation log that stores inputs, outputs and timestamps for all completed steps. This enables recovery and replay debugging.

Each workflow step is decorated with metadata indicating if it is:

(1) not transactional
(2) a transactional step using physical backout
(3) a transactional step using compensation

In the next four subsections, we describe the implementation of durability, the implementation of physical backout, the implementation of compensation, and finally a sketch of the not-yet-implemented consistency guarantees. We expect most workflows will either perform physical backout or use compensation. It is plausible that there are special cases where a mix of backout strategies will work.

7.1 DBOS's Durable Computing

DBOS implements workflow durability through a database-oriented logging approach that leverages PostgreSQL's transactional guarantees. For a transactional step, DBOS piggybacks workflow step logging with the user's database operations within the same database transaction. Before executing the step, DBOS checks whether the step has already been recorded in the operation log and returns immediately if an entry exists. The step then executes its application logic (e.g., updating inventory), and upon completion, DBOS records the output in the operation log. Critically, both the workflow log entry and the application database modifications commit atomically in a single Postgres transaction.

This atomic commit simplifies OAOO semantics: if the transaction commits, both the application changes and the log entry are durable; if it aborts, neither persists. During recovery, DBOS can determine whether a transaction step completed by checking the operation log—if an entry exists with output, the step completed; otherwise, it must be re-executed.

If a failure occurs (machine crash, process restart, network partition), the DBOS runtime queries the database to find all incomplete workflows. For each incomplete workflow, it examines the operation log to determine the last successfully completed step. The workflow is then resumed from that point, with all previous steps' results retrieved from the log. This guarantees once-and-only-once execution, as completed steps are never re-executed.

This database-oriented approach provides several advantages. First, the log is immediately durable and does not require separate checkpointing. Second, piggybacking transaction logging with application operations eliminates the need for complex two-phase commit protocols between workflow and application logging. Third, the log can be queried using SQL, enabling powerful debugging and auditing capabilities. Fourth, multiple DBOS instances can share the same database, providing automatic failover without complex consensus protocols.

7.2 Physical Backout Via Workflow-level Transactions

In DBOS's standard durable computing model, each transaction step executes in its own independent database transaction that commits immediately upon completion. This per-step transaction model provides excellent concurrency since locks are released as soon as each step finishes. However, when a workflow fails and must be backed out, each completed step's changes have already been committed to the database. Physical backout becomes impossible because the transaction logs might have been truncated, and other concurrent workflows may have already read or modified the same data, leading to the phantom and cascading rollback problems discussed in Section 5.

To enable physical backout, we extended DBOS to support workflows with physical backout steps using PostgreSQL's savepoint mechanism. For such workflows, our implementation wraps the entire workflow execution in a single database transaction spanning all steps. Before each step executes, the system creates a savepoint—a named marker within the transaction. If a step fails, the system rolls back to the previous savepoint, undoing all changes made by the failed step while preserving the work of earlier steps. Critically, because all steps execute within the same transaction, the workflow's changes remain invisible to other concurrent workflows until the entire workflow commits. If the workflow must be backed out entirely, a single transaction rollback undoes all steps atomically.

However, physical backout workflows have a significant limitation: they hold database locks for the entire workflow duration. If workflows are long-running or if multiple workflows access the same data, lock contention can severely impact throughput.

7.3 Backout via Compensation

For compensation workflows, we extended DBOS to implement saga-style backout. Each step can be decorated with a compensation function that logically undoes the step's effects. For example, a `reserve_inventory` step might have a `release_inventory` compensation that increments the inventory count.

When a saga workflow fails, the system automatically invokes the compensation functions for all completed steps in reverse order. The compensation functions are themselves workflow steps, logged in the operation log and subject to the same durability guarantees. If a compensation fails, the system retries it according to the configured retry policy.

The system records each step's completion in the operation log before proceeding to the next step. This ensures that if a failure occurs during compensation, the system knows exactly which compensations have been executed and which remain.

The saga approach provides better concurrency than transactional workflows because steps do not hold locks between steps. However, it provides weaker consistency guarantees because the modifications by intermediate steps of a workflow are exposed to other concurrent workflow. As discussed in Section 5, compensations executed concurrently with other workflows may produce outcomes different from serializable execution.

7.4 Workflow correctness

The algorithm described in Section 5 can be used to ensure workflow correctness. We previously built an experimental time-travel debugger on top of PostgreSQL [9], which can be extended to support workflow recovery by leveraging PostgreSQL's point-in-time recovery capability and re-executing and committing transactional steps based on recorded snapshot information [7]. Once recovery completes, the erroneous workflow instance can be discarded and the system can resume from the correct state. However, this mechanism applies only to transactional steps and does not address external service calls.

8 EXPERIMENTAL EVALUATION

We implemented a benchmark to evaluate the performance tradeoffs between saga-based compensation and transactional workflows with physical backout using our extended DBOS implementation. The benchmark simulates an e-commerce checkout workflow similar to the example in Figure 1, with steps for reserving inventory, checking credit, processing payment, and fulfilling orders.

8.1 Experimental Setup

The benchmark runs on PostgreSQL 14 with our extended DBOS implementation in Python. The workload consists of 1,000 bicycle models, 10,000 customers with normally distributed credit balances (mean \$5,000, standard deviation \$1,000), and 10,000 units of initial inventory per model. Each order attempts to purchase one bicycle for \$1,000 for one customer.

We evaluate four scenarios designed to stress different aspects of the workflow systems:

(1) **Uniform workload**: Orders uniformly distributed across all bicycle models, tested at 1, 10, and 50 concurrent workers to measure baseline performance and scalability.
(2) **Hot items with high conflict**: Orders following a Zipfian distribution on the bicycle models with 50 workers.
(3) **Long-running steps**: We inject 1 second artificial delay into the "processing payment" step to simulate external service calls, with 20 concurrent workers. We use similar zipfian distribution on the bicycle models to create contention.

For transactional workflows, we use PostgreSQL's SERIALIZABLE isolation level to provide the strongest correctness guarantees. Each step in saga workflow runs in a separate SERIALIZABLE transaction. We implement automatic workflow retry with exponential backoff (up to 20 retries, capped at 50ms delay) for aborts caused by serialization failures. When a transaction aborts, we abort the entire workflow by physical backout or compensantion. Each experiment measures goodput (successful orders per second without aborts), latency, and workflow abort rate.

8.2 Experimental Results

Table 1 shows the performance under uniform workload at varying concurrency levels. With a single worker, transactional workflows outperform sagas, achieving 43.7 orders/sec compared to 24.0 orders/sec. This advantage stems from commit overhead. Sagas require four separate transaction commits (one per step), while transactional workflows batch all operations into a single commit. With near-zero contention, the reduced commit overhead dominates.

Table 1: Performance comparison under uniform workload

# Workers	Approach	Goodput	P90 Lat.	Abort Rate
1	Saga	24.0	51ms	0%
1	Physical Backout	43.7	30ms	0%
10	Saga	163.4	58ms	0%
10	Physical Backout	208.6	55ms	20.6%
50	Saga	319.0	110ms	0%
50	Physical Backout	397.9	196ms	48.8%

This advantage persists at moderate concurrency. With 10 workers, transactional workflows still outperform sagas, though they already show 20.6% abort rate compared to 0% for sagas. At high concurrency with 50 workers, the dynamics shift further. While transactional workflows maintain higher raw goodput (397.9 versus 319.0 orders/sec), they show 78% higher P90 latency and a 48.8% abort rate compared to sagas. The extended lock-holding duration causes serialization conflicts that require expensive retries.

Table 2 presents results under hot item contention with varying degrees of skewness. With moderate skew (Zipfian factor 0.5), sagas achieve 196.0 orders/sec goodput with 0% abort rate, while transactional workflows drop to 43.4 orders/sec with 35.7% abort rate. Under higher skew, the gap widens further. Sagas maintain 67.1 orders/sec with 0% abort rate, while transactional workflows achieve just 24.8 orders/sec with 45.7% abort rate. The increased skewness concentrates conflicts on fewer bicycle models, amplifying the contention penalty for transactional workflows while sagas remain less affected due to their per-step commit model.

Table 2: Performance comparison under hot item contention

Skewness	Approach	Goodput	P90 Lat.	Abort Rate
Zipf θ=0.5	Saga	196.0	181ms	0%
Zipf θ=0.5	Physical Backout	43.4	1,107ms	35.7%
Zipf θ=0.9	Saga	67.1	245ms	0%
Zipf θ=0.9	Physical Backout	24.8	1,724ms	45.7%

Table 3 shows the results of workflow with long-running steps and a zipfian skewness parameter of 0.9 for the bicycle models. With 1-second delay in the payment processing step simulating external service calls, saga workflows complete at 15.9 orders/sec with P90 latency of 1.1 seconds. Transactional workflows with physical backout collapse to only 2.0 orders/sec with P90 latency of 24.6 seconds and 42.5% abort rate. Since the lock must be held while waiting for processing payment to finish, this creates severe contention, with transactions repeatedly aborting and retrying.

Table 3: Performance comparison with long-running steps

Approach	Goodput	P90 Lat.	Abort Rate
Saga	15.9	1,100ms	0%
Physical Backout	2.0	24,554ms	42.5%

Our experimental evaluation reveals a nuanced trade-off between saga-based compensation and transactional workflows. Under low contention, transactional workflows with physical backout outperform sagas by up to 1.8x because they require only one commit versus four commits for saga steps. This advantage persists at moderate concurrency levels. However, under high contention from hot items, high concurrency, or long-running steps, transactional workflows suffer from high abort rates (up to 48.8%) and latency increases, while sagas maintain consistent performance with near-zero abort rates.

9 FUTURE WORK AND CONCLUSIONS

9.1 Future Work

Several directions remain for future research. First, we plan to investigate applications of our semantic model to security and privacy issues. For example, GDPR defines personally identifiable information (PII) that must be deletable upon customer request. In a platform supporting our semantics, implementation of such a request is straightforward. One can inspect the platform log (assuming it is kept indefinitely) to find and delete all required copies. If enterprises share PII data, they can use their respective logs and unique identifiers to implement coordinated deletion, assuming all parties agree to honor such requests.

Second, we need to formally define the isolation guarantees provided by saga-style workflows. While we have described the correctness issues qualitatively, a formal model would enable reasoning about what guarantees applications can rely on. This is analogous to how ANSI SQL isolation levels formalize the guarantees of different transaction isolation modes.

Third, we wish to explore techniques for providing higher degrees of isolation for saga-style workflows without sacrificing too much performance. This might include optimistic concurrency control, escrow techniques for commutative operations, or hybrid approaches that use physical backout when conflicts are unlikely and compensation when conflicts are common.

Finally, the time-travel debugging capabilities enabled by comprehensive workflow logging deserve further exploration. DBOS's operation log already enables developers to replay workflows, inspect intermediate states, and understand failure scenarios. This could be extended to support speculative "what-if" analysis and automated bug detection.

9.2 Conclusion

In summary, it is a good first step for a platform to support durability as this supports execution when there are problems in the forward direction (the 10%). Additional guarantees are required to deal with errors (the 90%), namely:

- The user must guarantee their workflow is consistent
- The platform must guarantee atomicity in the case of any possible errors
- The platform must guarantee a correct outcome in the presence of concurrent backout.

In summary, a platform should guarantee that all consistent workflows are atomic, durable and correct, which we term AC/DC. AC/DC is an additional collection of guarantees. Hence, durability is a good first step, but it only automates a portion (typically 10%) of an application. However, AC/DC is required to substantially reduce the other 90%. In our opinion all durable computing platforms should move to supporting AC/DC.

Of course, long-running workflows will often be problematic. If they must be aborted, the performance consequences will be dire, unless they can be undone without impacting other workflows. In addition, workflows that have been completed cannot be undone. For example, in our example, if the customer returns the object he ordered, then a new workflow must deal with returns, since the example one has finished. Hence, there are assorted limitations of AC/DC which an application developer must be cognizant of.

One should think of AC/DC as the extension of ACID transactions to larger semantic units. The A and C of ACID continue to be enforced. Isolation can only be achieved by holding long-term locks, which are likely to have dire performance consequences. Hence, it should be dropped as a semantic property, leading to a more complex notion of consistency and correctness.

References

[1] Amazon Web Services. 2025. What is AWS Lambda? https://docs.aws.amazon.com/lambda/latest/dg/welcome.html Accessed July 15, 2025.
[2] Amazon Web Services. 2025. What is Step Functions? https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html Accessed July 15, 2025.
[3] DBOS Inc. 2025. DBOS. https://www.dbos.dev/ Accessed July 15, 2025.
[4] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. ACM, 249–259. doi:10.1145/38713.38742
[5] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, et al. 2023. Apiary: a DBMS-integrated transactional function-as-a-service framework. *arXiv preprint arXiv:2208.13068* 1 (2023).
[6] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. R3: Record-Replay-Retroaction for Database-Backed Applications. In *Proceedings of the VLDB Endowment*, Vol. 16. VLDB Endowment, 3085–3098.
[7] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. R3: Record-replay-retroaction for database-backed applications. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3085–3097.
[8] M. McGrath. 2012. *Understanding PaaS*. O'Reilly Media, Sebastopol, CA.
[9] Qian Li. 2025. Time Travel Queries with Postgres. https://qianli.dev/docs/qianli-pgconf-timetravel.pdf Accessed December 4, 2025.
[10] Restate. 2025. Restate. https://restate.dev/ Accessed July 15, 2025.
[11] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, et al. 2021. DBOS: A dbms-oriented operating system. (2021).
[12] Temporal Technologies Inc. 2025. Temporal: Durable Execution Solutions. https://temporal.io/ Accessed July 15, 2025.
[13] X/Open Company Ltd. 1991. *Distributed Transaction Processing: The XA Specification*. CAE Specification C193. The Open Group. https://publications.opengroup.org/standards/dist-computing/c193 X/Open eXtended Architecture standard for distributed transaction processing.