Transactions and Serverless are Made for Each Other

QIAN LI
AND
PETER KRAFT

IF SERVERLESS PLATFORMS COULD WRAP FUNCTIONS IN DATABASE TRANSACTIONS, THEY WOULD BE A GOOD FIT FOR DATABASE-BACKED APPLICATIONS.

Serverless cloud offerings are becoming increasingly popular for stateless applications because they simplify cloud deployment. This article makes the argument that if serverless platforms could wrap functions in database transactions, they would also be a good fit for database-backed applications. There are two unique benefits of such a transactional serverless platform: time-travel debugging of past events and reliable program execution with exactly-once semantics.

Serverless cloud platforms such as AWS (Amazon Web Services) Lambda and Azure Functions are increasingly popular for building production applications as varied as website front ends, ML (machine learning) pipelines, and image-processing systems. These platforms radically simplify development by managing application deployment. Developers can deploy functions with the click of a button and the platform automatically hosts them, guarantees their availability, and scales them to handle changing loads.

Serverless platforms are primarily used for stateless

operations such as image resizing or video processing. Here, we'll argue they should also be used to deploy stateful applications, particularly *database-backed applications* whose business logic frequently queries and updates a transactional database such as Postgres or MySQL. Database-backed applications are ubiquitous in modern businesses; examples include e-commerce web services, banking systems, and online reservation systems. They run primarily on server-based platforms such as Kubernetes. Thus, they form a massive opportunity for serverless offerings, including the backends of most enterprise APIs and much of the modern Web.

To make serverless work for database-backed applications, serverless platforms would need to make one critical addition: *Allow developers to execute functions as database transactions*. Figure 1 shows an inventory reservation function implemented in a conventional serverless platform versus a transactional serverless platform. The `checkInventory` and `updateInventory` functions perform SQL queries. In a conventional serverless platform, if a function accesses the database, developers must obtain a database connection, manually begin a transaction, execute business logic and SQL queries, and then finally commit the transaction (figure 1a).

By contrast, a *transactional* serverless platform manages the database connection: If a function accesses the database, it uses a platform-provided connection that automatically wraps the function in a transaction (figure 1b). The idea of building such a platform has been explored in several research projects—by these authors[1] and others.[3,4]

FIGURE 1: **AN INVENTORY RESERVATION FUNCTION**

a. conventional serverless

```
1 # Check if an item is available, then reserve it
2 def reserveInventory(itemId, num):
3    conn = getConnection(DBurl)
4    conn.beginTransaction()
5    avail = conn.checkInventory(itemId)
6    if (avail > num):
7       conn.updateIinventory(itemId, avail - num)
8    conn.commitTransaction()
```

b. transactional serverless

```
1 # Check if an item is available, then reserve it
2 def reserveInventory(itemId, num):
3    # Connection supplied by the platform
4    avail = conn.checkInventory(itemId)
5    if (avail > num):
6       conn.updateIinventory(itemId, avail - num)
```

As this article explains, a transactional serverless platform not only is more convenient for the developer, but can also provide powerful benefits for database-backed applications beyond the capabilities of conventional serverless or server-based systems.

First, a transactional serverless platform makes programs easier to debug. Modern applications are difficult to debug because they run in distributed settings with frequent concurrent accesses to shared state, so

bugs often involve complex race conditions that are not easy to reproduce in a development environment. Reproducing errors is particularly difficult in conventional serverless platforms, because their execution environment is transient and exists only in the cloud. A transactional serverless platform, however, can simplify debugging through *time travel*.[2] Because the platform wraps functions in transactions to coordinate their state accesses, a debugger can leverage the transaction log to "travel back in time" and locally replay any past transactional function execution.

Second, a transactional serverless platform can provide reliable program execution. Writing reliable database-backed applications is difficult because they often coordinate several business-critical tasks, any of which may fail. In a server-based application, addressing this problem is difficult as developers must manually track each request's status and recover failed requests. Conventional serverless platforms make this easier by automatically restarting any task that fails, but this can be problematic if it causes an operation to execute multiple times (for example, paying twice). If functions are transactions, however, the platform can record their success or failure *in the same transaction as their business logic*, thus guaranteeing that each function executes once and only once.

PROGRAMMING A TRANSACTIONAL SERVERLESS PLATFORM

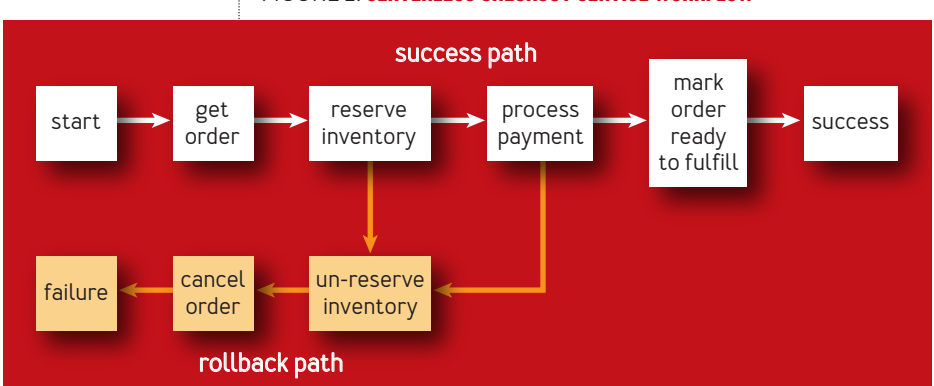
A transactional serverless platform could provide a programming model similar to today's serverless

platforms, where developers write programs as *workflows of functions*. Each function performs a single operation. Workflows, implemented as directed graphs or state machines, orchestrate many functions. Popular serverless workflow orchestrators include AWS Step Functions and Azure Durable Functions.

The distinguishing feature of a transactional serverless platform is that all functions accessing the application database are wrapped in ACID (atomic, consistent, isolated, and durable) database transactions, as shown in figure 1b. These functions must be deterministic and have no side effects outside the database. Functions not accessing the database, such as those making external API calls, work the same as they do in conventional serverless platforms.

As a running example for this article, figure 2 shows a diagram of a serverless checkout service workflow that first reserves inventory for all items in an order, then processes payment for the order, and finally marks the

FIGURE 2: **SERVERLESS CHECKOUT SERVICE WORKFLOW**



order as ready to fulfill. Each step is implemented in a separate function. All functions except "process payment" (which uses a third-party payment provider) contact the database and are wrapped in transactions. If any step fails, the workflow runs rollback functions to undo previous operations (e.g., returning reserved inventory if the payment fails).

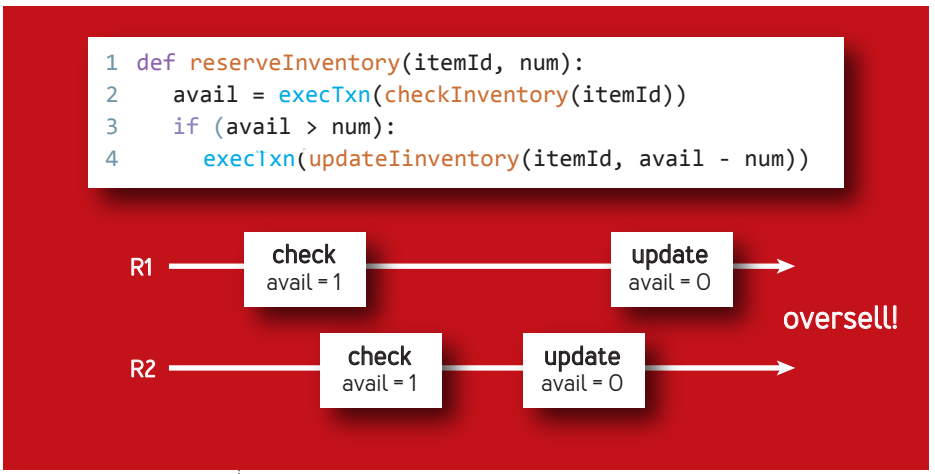
TIME-TRAVEL DEBUGGING

One powerful and unique feature enabled by a transactional serverless platform is time-travel debugging: letting developers faithfully replay production traces in a local development environment to reproduce bugs that happened in the past. Time-travel debugging is especially useful for database-backed applications because they frequently run in distributed environments where bugs manifest as race conditions that occur only under high concurrency and are nearly impossible to reproduce locally.

For example, suppose the "reserve inventory" operation in figure 2 is split into two separate transactional functions, as in figure 3, which shows a buggy implementation of the "reserve inventory" operation. This implementation contains a race condition where if two requests arrive at the same time, both can reserve the same item, potentially causing the vendor to sell more items than it has available.

Debugging issues like this is tricky because they surface only if multiple concurrent requests with specific inputs are interleaved in a specific way with a particular database state. To reproduce the bug locally, the developer must determine not only which requests caused the bug, but

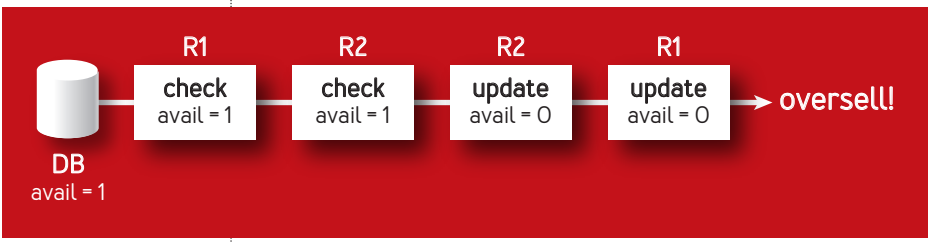
FIGURE 3: **A BUGGY IMPLEMENTATION OF THE "RESERVE INVENTORY OPERATION"**



also the order in which different operations in those requests interleaved and the exact database state that made the bug possible. In a conventional platform, tracking execution order and reconstructing database state are prohibitively expensive: Requests execute concurrently on many parallel threads on many distributed servers, potentially modifying the database thousands of times per second.

By contrast, prior research[2] has shown that a transactional serverless platform makes faithful replay practical because each function is wrapped in an isolated, atomic, and deterministic transaction. This enables a *time-travel debugger*, which can faithfully replay a production trace (including race conditions and concurrency bugs) in two steps:

FIGURE 4: **A TIME-TRAVEL DEBUGGER REPLAYING AN EXECUTION TRACE**

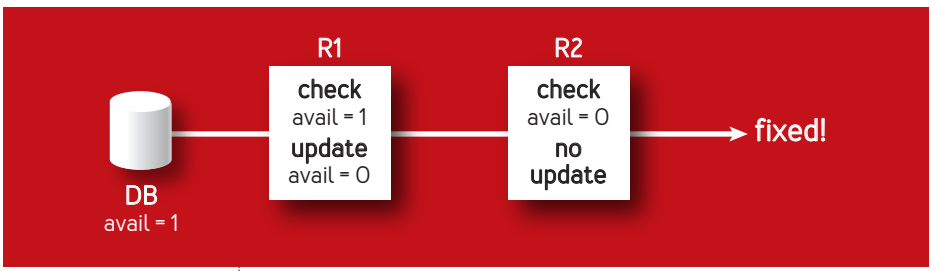


1. Using database transaction logs, it can reconstruct the state of the application database at the time of the trace's first request.
2. It can locally execute each request in the trace on the reconstructed database, executing their transactional functions in the order they originally executed in the application database's transaction log.

A time-travel debugger improves developers' lives by reproducing complex concurrency bugs in a controlled local environment. For example, if the debugger is run on a trace containing the bug described in figure 3, it executes both check transactions on a database containing only one item, then executes both update transactions, thus overselling the item and reproducing the bug. This process is shown in figure 4.

A time-travel debugger can provide another powerful feature called *retroaction:* the execution of modified code over past events. For a given trace, the debugger performs retroaction similarly to faithful replay but uses the updated implementation of each function instead of the original one. Retroaction is especially useful for regression testing: running a new code version over old production

FIGURE 5: **A TIME-TRAVEL DEBUGGER TESTING A FIX TO THE RESERVE INVENTORY BUG**



traces to verify it handles them correctly. For example, let's say the bug in figure 3 was fixed by combining the check and update functions into a single transactional function. A time-travel debugger can retroactively test this fix by re-executing the original trace but running the combined function in place of the original checks and updates. As shown in figure 5, this validates that the fix eliminates the bug.

RELIABLE PROGRAM EXECUTION

Another key benefit of a transactional serverless platform is *reliable program execution*. Many database-backed applications must coordinate multiple business-critical tasks, any of which may fail. For example, the checkout workflow in figure 2 performs three tasks for each order: (1) reserving its inventory; (2) processing its payment; and (3) marking it as ready to fulfill. To execute reliably, such applications must not only handle failures in any of those tasks, but also recover from interruptions such as server crashes. Specifically, they must have two properties:

- *Programs run to completion.* If a program begins executing, it must continue, recovering through any interruptions until it reaches a terminal success or failure state. For example, if the checkout service is interrupted after processing a payment, it must recover and either mark the order as fulfilled (if the payment succeeded) or cancel the order and return reserved inventory (if the payment failed).
- *Operations execute exactly once.* While executing a program, each of its operations must execute once and only once. For example, if you are recovering the checkout service after it is interrupted, you cannot naively resend the payment request; otherwise, the customer may pay twice. You must instead determine the status of the original payment request (whether it was sent at all, and if so, whether it succeeded or failed) and recover accordingly.

Manually obtaining these properties in a traditional server-based application is difficult. One approach is to write the application as a state machine that checkpoints its state to persistent storage after every operation. If the program is interrupted, resume execution from the last checkpointed state. To ensure exactly-once execution, make all operations idempotent so they can be safely re-executed during recovery. While such an approach works, it is tedious and error-prone, and requires careful program design.

Existing serverless platforms simplify writing programs that run to completion but do not provide exactly-once execution. This follows naturally from the serverless programming model. If a program is written as a workflow

of functions, the workflow orchestrator can record the workflow's state after every function execution, then resume from the last recorded state if workflow execution is interrupted. Thus, serverless function orchestrators such as AWS Step Functions and Azure Durable Functions run workflows to completion, restarting each function until it succeeds or reaches a predefined failure state.

Durable workflow engines such as Temporal provide similar guarantees for server-based programs, provided they are written as workflows of operations. Because orchestrators treat functions as black boxes, however, they cannot provide exactly-once semantics, but instead restart each function until it succeeds. If a function crashes after completion but before its success is recorded, it is re-executed, potentially corrupting data.

As prior work has shown,[1,4] a transactional serverless platform can guarantee not only that programs run to completion, but also that transactional operations execute exactly once. Because the platform wraps functions in transactions, it can record the success or failure of a transactional function *in the same transaction as the function*. Therefore, if a function completes, its success or failure is always recorded in the database, while if a function fails, all its actions are rolled back by the database. Thus, the platform knows never to re-execute a function with a recorded result but can always safely re-execute without a recorded result.

CONCLUSION

Database-backed applications are an exciting new frontier for serverless computation. By tightly

integrating application execution and data management, a transactional serverless platform enables many new features not possible in either existing serverless platforms or server-based deployments. This article has explained how such a platform could benefit application debuggability and reliability.

Its additional benefits include:

- *Observability*, as the platform can track the full history (provenance) of each data item through all functions that have modified it.
- *Security*, as the platform can monitor all operations on data in realtime.
- *Performance*, as the platform can collocate transactional functions with the application database.

We look forward to future work in this space.

References

1. Kraft, P., Li, Q., Kaffes, K., Skiadopoulos, A., Kumar, D., Cho, D., Li, J., Redmond, R., Weckwerth, N., Xia, B., Bailis, P., Cafarella, M., Graefe, G., Kepner, J., Kozyrakis, C., Stonebraker, M., Suresh, L., Yu, X., Zaharia, M. 2023. Apiary: a DBMS-integrated transactional function-as-a-service framework. arXiv:2208.13068; https://arxiv.org/abs/2208.13068.
2. Li, Q., Kraft, P., Cafarella, M., Demiralp, C., Graefe, G., Kozyrakis, C., Stonebraker, M., Suresh, L., Yu, X., Zaharia, M. 2023. R^3: record-replay-retroaction for database-backed applications. *Proceedings of the VLDB Endowment* 16(11), 3085–3097; https://dl.acm.org/doi/10.14778/3611479.3611510.
3. Wu, C., Sreekanti, V., Hellerstein, J. M. 2020.

Transactional causal consistency for serverless computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 83–97; https://dl.acm.org/doi/10.1145/3318464.3389710.

4. Zhang, H., Cardoza, A., Chen, P. B., Angel, S., Liu, V. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th Usenix Symposium on Operating Systems Design and Implementation*, 1187–1204; https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

Qian Li *is a co-founder at DBOS, Inc. She received her PhD in computer science from Stanford University, advised by Christos Kozyrakis. Her research interests include serverless, databases, and cloud resource management.*

Peter Kraft *is a co-founder at DBOS, Inc. He obtained his PhD in computer science from Stanford University, where he worked with Matei Zaharia and Peter Bailis. His interests are in databases, cloud infrastructure, and distributed systems.*