Transactions Make Debugging Easy

Qian Li[1], Peter Kraft[1], Michael Cafarella[2], Çağatay Demiralp[3], Goetz Graefe[4], Christos Kozyrakis[1],
Michael Stonebraker[2], Lalith Suresh[5], and Matei Zaharia[1]

[1]Stanford, [2]MIT, [3]Sigma Computing, [4]Google, [5]VMware

Abstract

We propose TROD, a novel transaction-oriented framework for debugging modern distributed web applications and online services. Our critical insight is that if applications store all state in databases and only access state transactionally, TROD can use lightweight always-on tracing to track the history of application state changes and data provenance, and then leverage the captured traces and transaction logs to faithfully replay or even test modified code retroactively on any past event. We demonstrate how TROD can simplify programming and debugging in production applications, list several research challenges and directions, and encourage the database and systems communities to drastically rethink the synergy between the way people develop and debug applications.

1 Introduction

In this paper, we propose TROD, a novel **Transaction-Oriented Debugging** framework for modern distributed web applications such as a travel reservation website or an e-commerce microservices application. TROD targets applications that follow three design principles:

P1. Store all application-shared state in databases.

P2. Access or update shared state only through ACID transactions.

P3. Produce deterministic outputs and state changes.

We adopt these principles because they radically simplify the problem of debugging modern distributed applications. Currently, debugging is hard because developers need to unravel the complex interactions of thousands of concurrent events [28]. Existing distributed debugging tools are limited as they rely on developers to provide sufficient logs and traces [29], which requires intensive manual logging or annotations. However, if applications follow our principles, TROD can augment database transaction logging to capture a complete record of application state accesses and changes, enabling powerful features such as faithful replay of any past event.

The TROD principles are practical because they align with current trends in application design. For example, developers increasingly deploy applications on serverless platforms such as AWS Lambda [2]. These serverless applications naturally follow TROD principles because they handle requests with stateless and deterministic functions and manage state using cloud databases. In fact, we originally developed TROD to debug applications in the DBMS-oriented operating system (DBOS) project [16], which runs programs as workflows of transactional serverless functions [14]. As part of DBOS, we found it was easy to build performant web and microservice applications using TROD principles.

TROD helps developers investigate and better understand bugs by faithfully replaying past events. Faithful replay is challenging because we need to accurately reconstruct past state for applications while considering interleaving concurrent executions. However, because TROD assumes state is centralized in databases and only accessed transactionally, it can capture a detailed history of application state changes and events with lightweight always-on transaction tracing. Developers can directly query traces to locate buggy executions. Then, during replay, TROD can re-apply logged state changes to reconstruct past application states. Since TROD can consistently replay past events, it transforms most Heisenbugs [11], bugs that happen rarely and are hard to reproduce because of complex and unpredictable interactions between concurrent events, into easily reproducible "Bohrbugs".

TROD extends faithful replay to support an even more powerful debugging feature: retroactive programming. Developers can use TROD to test their modified code on past events, for example, to test a bug fix before pushing it into production. Retroactive programming is challenging because TROD cannot simply re-apply the transaction log as in replay but must actually re-execute all concurrent events as their computations and effects might change. Retroactive programming is possible in TROD because request executions only share state through transactions. Therefore, TROD can identify relevant transactions and only enumerate possible re-execution orderings of those transactions to thoroughly test different possible effects of interleaving concurrent executions.

Our proposed implementation of TROD mainly focuses on correctness issues such as functionality or semantics bugs, with an emphasis on hard-to-debug server-side concurrency issues [24]. At present, we have much of TROD running in DBOS. Preliminary results are promising, including low overhead (<15%) always-on tracing. We expect to present a demonstration at the conference.

2 Debugging Frustrations

Debugging applications that serve many concurrent requests, such as web services or microservices, is difficult because developers need to unravel the complex interleaving of concurrent executions. We illustrate real debugging frustrations using a concurrency bug (Figure 1) reported in the popular database-backed online education platform Moodle[1]: MDL-59854 [21]. This is a time-of-check to time-of-use (TOCTOU) bug: a race condition exists in the subscribeUser handler between checking if a subscription exists (1st transaction) and inserting a new subscription into the database (2nd transaction). However, debugging this issue is tricky because it only surfaces if two requests for the same user to subscribe to the same forum are interleaved in a certain way, and an error is

---

[1]Moodle stores shared application state in a relational SQL database and uses transactions to manage state. It supports MySQL, Postgres, Microsoft SQL Server, or Oracle.

```
1  def subscribeUser(userId, forum):
2    # 1st transaction: Check subscription.
3    if (isSubscribed(userId, forum)):
4      return True
5    # 2nd transaction: Insert a subscription entry.
6    result = DB.insert("forum_sub", userId, forum)
7    return result
8
9  def fetchSubscribers(forum):
10   # Error: Duplicated values in column userId.
11   result = DB.executeQuery("SELECT userId FROM
       forum_sub WHERE forum=forum")
12   return result
```

Figure 1: Simplified code for a concurrency bug in the database-backed online education platform Moodle (MDL-59854). The first handler, subscribeUser, has two transactions and manifests a race condition causing duplicated subscriptions but returns with no error. The second handler, fetchSubscribers, fetches the list of subscribers and raises an error if it detects duplicates.

only raised on a subsequent request to fetch a list of subscribers to that forum. The developer who reported this bug commented: "*You have to be pretty fast and pretty lucky to actually reproduce this issue.*"

Locating and reproducing concurrency bugs like MDL-59854 is fundamentally challenging because these bugs are caused by specific and rare interleavings of concurrent events that are seemingly unrelated. Conventional error messages and stack traces, as commonly used in bug reports, are not sufficient because they only provide information on requests that fail but not on other requests that may be involved in the bug. For example, in MDL-59854, logs and stack traces did not explain why duplicated entries appeared in the first place, and did not reference the original interleaved execution order that caused the issue. Instead, developers had to guess which requests and functions may have inserted or updated entries to the affected database table, and struggled to reproduce the bug because they lacked debugger support to faithfully replay past events.

Based on this and other bug reports that we examined, we observe that to find and reproduce bugs, developers must often collect information from multiple application production logs as well as database logs. Debugging would be easier if developers had a full view of application execution and database interactions and could replay past events. To make this possible, we are building TROD.

3 TROD: Transaction-Oriented Debugger

3.1 TROD Overview

Our TROD proposal focuses on debugging correctness issues in distributed web applications and online services such as a travel reservation web service or e-commerce microservices application. These applications typically implement their business logic in backend servers that consist of many request handlers. When a request arrives at the backend, the application runtime invokes the corresponding request handler to serve the request. Many applications implement a microservice architecture [15], so to serve a single user request, a request handler may invoke multiple other request handlers through RPCs, forming a workflow of handler invocations. TROD assumes that applications propagate a unique ID for each request (ReqId) through RPCs, which is a common practice.
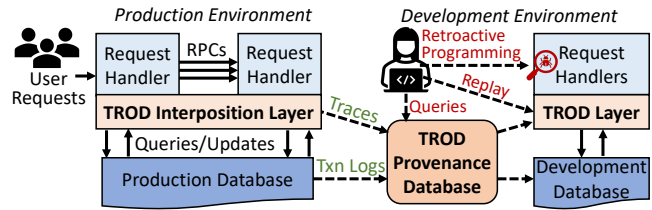

Figure 2: TROD's integration with production and development environments. TROD's tracing flows and debugging operations are marked with dashed arrows and colored text.

TROD requires applications follow three design principles (Section 1). They should store all shared state and data in a transactional database (**P1**), and request handlers should only access or update shared state through ACID transactions (**P2**). Additionally, each request handler should be deterministic (**P3**), where its output and state changes are determined only by the input and the database state at the time of its execution. Request handlers can maintain local private state for individual requests, but otherwise access or persist state across requests only through databases. TROD can work for applications using multiple data stores if their transaction logs are aligned (e.g., using a cross-data store transaction manager [7, 9]), but in the rest of the paper, we assume applications use a single DBMS for simplicity.

As shown in Figure 2, TROD has two major components:

- *Interposition Layer*: TROD adds a thin shim layer that hooks into the developer's application framework. It interposes on every handler and database query to trace application-database interactions, including transaction execution orders and information on which data items were read from or written to the database.

- *Provenance Database*: TROD utilizes an analytical database to store traced data. TROD uses it to replay past events; developers can also directly query this database for debugging.

Simplifying Assumptions. We assume external service calls are idempotent, so re-executions will not generate unexpected side effects (e.g., send an email twice). We also assume that machines are reliable and have enough hardware resources, so resource exhaustion is out of the scope of this paper. We leave proper handling of external services and hardware environments for future work.

Additionally, we assume for simplicity that the database provides strict serializability, meaning transactions are serializable and serialized in commit order [12]. However, TROD can work for lower isolation levels such as snapshot isolation and read committed by leveraging prior work on *transaction reenactment* [1], which can faithfully replay transactional histories under weak isolation levels using database audit logs and time travel capabilities.

3.2 TROD Principles

TROD design principles make debugging easier and more efficient because they exploit the synergy between application development and debugging. If request handlers are deterministic and only access shared state transactionally through a database, TROD can automatically intercept queries and trace data operations to capture provenance information. Then, during replay, TROD can efficiently restore application state by re-applying the captured sequence of changes

made to the database, without needing to replay handler executions. By contrast, if an application maintains shared state outside of the database, such as in files or in-memory data structures, obtaining provenance information requires low-level code instrumentation or intensive manual annotations and can be prohibitively expensive.

TROD principles are practical because they align with two important trends in application development. First, developers increasingly deploy applications on serverless platforms (e.g., AWS Lambda [2]). Serverless applications naturally follow TROD principles because they handle requests with stateless and deterministic functions and manage state using cloud databases. Second, due to the growing demand for strong consistency, data stores are increasingly adding transactional guarantees while providing high performance [30]. For example, many NoSQL data stores now support ACID transactions (e.g., FoundationDB, MongoDB). Therefore, it will only become easier for developers to adopt TROD principles without substantially changing their applications or data stores.

3.3 TROD in Action

To show how TROD works, we describe how a developer could use it to debug our earlier example MDL-59584 (Section 2), a bug that caused duplicated entries in Moodle's forum subscription table. A major challenge in solving this bug was figuring out which operation created the duplicate entries. In TROD, this is easy: developers directly query TROD's provenance database to track which prior requests inserted the duplicated records. TROD automatically traces essential provenance information such as request IDs, timestamps, handler names, transaction IDs, and transaction read and write sets, storing this information in structured database tables (we discuss details in Section 3.4). Here is an example query to find the requests and corresponding handlers that inserted the duplicated records:

```
SELECT Timestamp, ReqId, HandlerName
FROM Executions as E, ForumEvents as F
  ON E.TxnId = F.TxnId
WHERE F.UserId = 'U1' AND F.Forum = 'F2'
  AND F.Type = 'Insert'
ORDER BY Timestamp ASC;
```

This query returns two different request IDs with the same handler name ("subscribeUser") and adjacent timestamps, indicating a potential concurrency bug in the subscribeUser handler. Developers can give this information to TROD to replay (Section 3.5) the requests that inserted the duplicated subscription. During replay, TROD uses its captured provenance data to provide replayed transactions with the correct database state/context by interposing between handlers and the database. Developers can then use low-level debugger tools like GDB to add breakpoints in the handler code and step through the replayed request execution.

After identifying and fixing the bug, developers can test their bug-fix patch with TROD's retroactive programming feature (Section 3.6). For instance, one developer suggested [21] that isSubscribed and DB.insert should be wrapped in one transaction. Thus, they can modify subscribeUser accordingly and use TROD to re-execute request handlers to test the original two conflicting subscription requests over a past snapshot, and observe if the patch actually fixes the duplication issue.

3.4 Always-On Tracing and Declarative Debugging

To collect information needed for debugging, TROD's interposition layer automatically interposes on each request handler executed during applications' normal execution in production environments, collecting request IDs, handler names, and execution timestamps. TROD also interposes on every operation in the application database to record which request handler invokes which transactions and what data is read from or written to the database. Thus, TROD can trace the flow of data through handler executions, recording which transactions are part of which handler executions and which data items they read or write. TROD organizes this captured *provenance* data into structured tables in an easily accessible and queryable provenance database. Our prototype of this always-on tracing achieves low runtime overhead in DBOS (Section 3.7), indicating it is practical.

We illustrate TROD's provenance logs using MDL-59854. TROD records transaction executions in the Invocations table (Table 1), which contains both transaction information (e.g., transaction IDs and timestamps) and request handler metadata (e.g., the handler and function names that initiated the transaction). Records for a request share the same ReqId, so developers can easily observe the handler and transaction execution order within and across requests. Developers can query this table to inspect execution histories, such as finding all transactions involved in serving a request.

TxnId	Timestamp	HandlerName	ReqId	Metadata
TXN1	TS1	subscribeUser	R1	func:isSubscribed
TXN2	TS2	subscribeUser	R2	func:isSubscribed
TXN3	TS3	subscribeUser	R2	func:DB.insert
TXN4	TS4	subscribeUser	R1	func:DB.insert
TXN9	TS9	fetchSubscribers	R3	func:DB.executeQuery

Table 1: An illustrative transaction execution log.

For each application table, TROD tracks data provenance such as what data items are read or written by each transaction. For data writes, TROD leverages the change data capture feature provided by most of the databases. For data reads, TROD can use existing database provenance techniques to rewrite queries [1] and record read set automatically. For example, assume the provenance table name for forum subscription is ForumEvents (Table 2).

TxnId	Type	Query	UserId	Forum
TXN1	Read	Check if (U1, F2) exists	null	null
TXN2	Read	Check if (U1, F2) exists	null	null
TXN3	Insert	Insert (U1, F2)	U1	F2
TXN4	Insert	Insert (U1, F2)	U1	F2
TXN9	Read	Select UserId for F2	U1	F2
TXN9	Read	Select UserId for F2	U1	F2

Table 2: An illustrative data operations log.

Developers can use a declarative language like SQL to directly query TROD's captured data for debugging, which simplifies locating the root causes of bugs. For example, if we run the query shown in Section 3.3 over Tables 1 and 2, TROD correctly returns information identifying the buggy request executions: (TS3, R2, subscribeUser) and (TS4, R1, subscribeUser).

3.5 Bug Replay

Using TROD, developers can easily replay past requests in a development environment. Our key insight is that if handlers are deterministic and access shared state only transactionally, TROD can faithfully replay a past execution by re-executing its code normally but restoring the database before each transaction to a state equivalent to what the transaction originally saw. To make this possible, TROD adds a breakpoint before the beginning of each transaction. Within a breakpoint, TROD queries the provenance database to find what state changes the upcoming transaction depends on, and then applies those changes to the development database. Therefore, the re-executed transaction can see the same state and return the same result as the original execution. TROD then repeats the above steps at each breakpoint until all handlers and transactions finish.

TROD's replay is practical and easy to use because it can faithfully replay production events in a smaller development environment and is compatible with other debugging tools. TROD can efficiently replay request executions without needing to restore the entire production database because it can use captured provenance information to only restore those data items used in replayed transactions. During replay, TROD lets developers use familiar low-level debugging tools to inspect executions in detail. For example, developers can use GDB to add breakpoints and single-step through a replayed handler execution, or use built-in database monitoring tools to observe detailed query plans. TROD augments these debugging tools with new capabilities. For example, if a request spans multiple transactions, TROD makes it easy for developers to query which concurrent executions may have updated the database between transactions, making many concurrency bugs easier to understand.

To debug the Moodle issue (MDL-59854) with TROD, developers can replay the request R1 that caused the duplication. TROD replays the original execution order of R1, shown in Figure 3 (top). First, TROD restores the database to a snapshot before R1 execution and replays R1's first transaction isSubscribed, which finds that no subscription exists. Then, TROD injects the relevant database change made by request R2, which inserts a subscription entry (U1, F2). Next, TROD re-issues R1's second transaction to the database and in turn inserts the duplicated entry. During replay, developers can see that the database was modified by R2 between the executions of R1's two transactions, indicating the cause of the bug. Therefore, TROD provides detailed information for developers so they can better understand how duplication occurs: interleaving executions of transactions from two requests.

3.6 Retroactive Programming

TROD not only lets developers faithfully replay past executions, but also lets them *retroactively program* applications, modifying code and testing it on past events. Retroactive programming is important because it helps developers verify that a proposed bugfix patch actually fixes the bug and does not introduce new bugs.

Retroactive programming is challenging because to fix bugs, developers typically have to change application logic. For instance, developers may split a long transaction into multiple shorter ones or combine several small transactions into a big one. Therefore, TROD cannot re-apply the transaction log as in bug replay, but must actually re-execute all relevant concurrent requests. It is not obvious
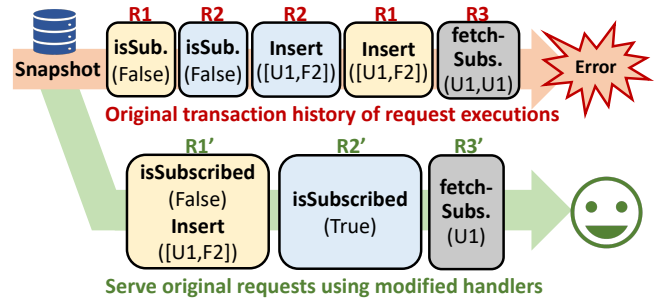

Figure 3: Top: Original transaction log. Bottom: Retroactive programming log where TROD executes the modified code to serve original user requests on a past database snapshot. We color boxes handling the same request with the same color, and put the return values in parentheses under each function name.

in which order to execute those concurrent requests, especially if their logic and transaction boundaries have been changed. Naively, there are a prohibitively large number of possible ways to interleave instructions among concurrent executions. However, since TROD requires handlers only share state through transactions, TROD can identify relevant transactions (e.g., transactions that access the same table) and enumerate possible re-execution orderings among them to thoroughly test different possible effects of interleaving concurrent executions.

To test the bugfix patch for the Moodle issue (MDL-59854), developers can use TROD to re-execute original requests R1, R2, and R3 using the modified code where the subscribeUser handler now wraps isSubscribed and DB.insert in one transaction. First, TROD restores the development database to a snapshot right before R1 execution. Then, TROD branches off from the original history to execute and trace new executions. TROD tests two possible re-execution orders for the concurrent requests (R1' first or R2' first), and we show one of them in Figure 3 (bottom). After both R1' and R2' finish, TROD executes R3'. Developers can observe that the third request which fetches subscribed users no longer raises any errors.

3.7 TROD Prototype

We are currently prototyping TROD in DBOS. As introduced in our recent paper [14], we implement always-on tracing using a high-performance in-memory buffer and find that on popular microservices benchmarks, the overall tracing overhead is <100μs per request. This causes a relative overhead of <15% when using the in-memory database VoltDB and negligible overhead when using the on-disk database Postgres. We also run declarative debugging queries over billions of events and get results in <5 seconds, which is promising for interactive debugging. We are currently implementing bug replay and retroactive programming, and expect to give a demonstration at the conference.

4 Case Studies

In this section, we examine two categories of bugs: 1) concurrency bugs, which can be debugged through TROD bug replay and retroactive programming, and 2) security bugs, which can be debugged using TROD tracing and declarative debugging.

4.1 Concurrency Bugs

To show how TROD can simplify debugging, we examine concurrency bugs discussed in prior work on non-reproducible bugs [8] and server-side request races [24]. In particular, we look at bug reports from two popular database-backed web applications: MediaWiki (MW) and Moodle (MDL). MediaWiki is an open-source platform underlying Wikipedia and related sites, while Moodle is an online education web platform; both have millions of users. Both applications use a relational SQL database, such as MySQL and Postgres, and transactions to manage application data. We manually review bug reports and search for recent bugs marked as "hard to reproduce," looking in particular for database-related bugs.

In MediaWiki, we observe that non-atomic update of multiple entries is a common cause of concurrency bugs which TROD simplifies debugging. For example, *MW-44325* [17] has a race condition where multiple concurrent edits of the same page can create duplicated site URL links, which violates the uniqueness requirement of site URL links. This bug is caused by non-atomic updates of the edited page object and the `SiteLink` table that stores page links, but developers struggled to reproduce the bug and accurately locate the root cause. The bug discussion included 33 developers, started in November 2012, closed and reopened multiple times, and was finally marked as resolved in October 2021. Developers took such a long time to fix this bug because they mostly rely on production logs to figure out ways to reproduce it; however, their logs do not provide a full view of concurrent executions and data operations, so developers have to guess the execution sequence that might lead to the error. By contrast, to debug this issue with TROD, developers can simply query TROD provenance data to inspect which requests edited the same page and created the duplicated URL links. Then, developers can use TROD to replay the buggy requests to examine the interaction of relevant concurrent executions and their corresponding changes to the page object and the `SiteLink` table.

Another similar bug was *MW-39225* [18] where the page edit handler rarely and randomly returns wrong article size changes in the presence of concurrent edits. This bug is due to non-atomic page edits so the concurrent requests can see partially updated data. Developers complained that the server logs were not helpful, and they had to guess ways to reproduce the bug, which led to a three-month-long discussion thread. Similar to MW-44325, developers can use TROD to locate and reproduce this bug through declarative debugging and bug replay.

In Moodle, we found several more database-related time-of-check to time-to-use (TOCTOU) bugs similar to MDL-59854 (Section 2), which caused duplications in other database tables and can be debugged with TROD. Sometimes, fixes to these bugs cause more bugs. For example, the patch for MDL-59854 later caused another production error, MDL-60669 [20], because it did not consider the corner case where duplications still exist in deleted courses, so restoring those courses raised errors. TROD's retroactive programming feature can prevent these new bugs by helping developers validate bug fixes before applying them in production. For example, to thoroughly test their bug fixes, developers can apply the modified code and serve past user requests directly related to this bug and other requests that may touch the same table.

4.2 Security Bugs

Prior work has shown that security bugs form a large and growing class of bugs in web applications [23]. These bugs may allow attackers to illegally modify application state or exfiltrate sensitive user data from an application, leading to severe security breaches. However, investigating security issues is hard because developers need to trace across many request handlers running on different machines. Currently, developers can only investigate these issues using system logs, but these logs may not provide sufficient information to track attacks [5]. For example, logs usually contain error messages but do not provide visibility into how applications interact with databases. By contrast, developers can easily query TROD's provenance database to check for access control violations or potential data exfiltrations.

TROD makes it easy to check web applications for violations of common access control patterns, as identified by Near and Jackson [23], such as the Authentication pattern (only allowing logged-in users to read certain objects) and the User Profiles pattern (only users themselves can update their profiles). We now demonstrate how TROD can help detect potential violations of the User Profiles pattern; TROD can detect violations of the other patterns through similar means. Specifically, we want to find all requests that illegally updated a profile (i.e., the request was not made by the profile owner). We can do this with the following SQL query:

```
SELECT Timestamp, ReqId, HandlerName
FROM Executions as E, ProfileEvents as P
  ON E.TxnId = P.TxnId
WHERE P.UserName != P.UpdatedBy AND P.Type = 'Update'
```

TROD can also support more complex forensics investigation queries such as detecting data exfiltration through workflows [5], where attackers can leverage RPCs between handlers to move stolen data laterally through workflow executions and finally exfiltrate data over a seemingly valid workflow. Since TROD traces the entire workflow of handler invocations that serve each request, developers can query TROD provenance data to track all subsequent changes made by a request that improperly accessed sensitive data, and determine if the data is exfiltrated.

5 Challenges and Research Directions

Debugging Performance and Data Issues. In this paper, we mainly focus on functionality bugs, but we hope to extend TROD to other categories such as 1) performance bugs where executions are correct but slow, and 2) data quality bugs which lead to well-formed but incorrect data, caused mainly by human errors. We can extend TROD to support more monitoring features like performance profiling to help developers troubleshoot performance bugs. For example, commercial APM (application performance monitoring) tools such as Retrace [27] and New Relic [26] allow developers to annotate their applications to obtain transaction traces for debugging slow queries. TROD can similarly augment its execution tracing to record performance metrics such as latencies of individual handlers and end-to-end executions, and store this information in a structured and queryable format. We believe these automatically generated traces would make it far easier for developers to use data-driven performance debugging tools like Seer [10], which currently require extensive and tedious manual annotations.

We also believe TROD can simplify debugging data quality issues, because it captures all application data changes and accesses. TROD can leverage prior work on data debugging. For example, we may support data quality tests over TROD's provenance database to discover erroneous edits [22], and find requests that caused data quality degradation. One challenge is to balance the granularity of provenance recording and the accuracy of data quality testing. We expect to extend TROD's bug replay to reconstruct more detailed provenance data retroactively [1] without harming performance.

Handling Multiple Data Stores. Modern web applications and microservices may use multiple data stores for heterogeneous data. For example, they can use a combination of a relational DBMS (e.g., Postgres, MySQL), a key-value store (e.g., Redis), a document store (e.g., MongoDB), and a search engine (e.g., ElasticSearch) [15]. It is challenging for these applications to use TROD because some data stores do not support transactions, and transaction logs of different stores are usually not aligned. However, recent work has proposed transaction managers that support transactions across heterogeneous, even non-transactional, data stores [7, 9]. Such transaction managers can also provide aligned transaction logs. Therefore, we believe TROD can work for applications using multiple data stores if they adopt such cross-data store transaction managers.

Guaranteeing Security and Privacy. Web applications often need to comply with regulations such as GDPR and CCPA to guarantee the security and privacy of sensitive user data. This poses a challenge for TROD because TROD may log sensitive user data such as personally identifiable information (PII). In order to effectively debug applications while respecting user privacy, TROD needs to let users completely remove any provenance data entry that potentially contains their personal information and support debugging from partial data. Therefore, we plan to research ways to maintain non-sensitive but critical metadata and partially log encrypted data for debugging without violating regulations.

6 Related Work

Provenance for Debugging. Prior work [13] has shown that provenance information can improve the reproducibility of database transactions and process debugging. Similar to TROD, VisTrails Total Recall [4] enables reproducibility for scientific workflows that interact with databases by combining workflow and data provenance. GProM [1] introduces a provenance middleware that interposes between user queries and databases; it also relies on transaction logs and database time travel to reconstruct provenance on demand to reduce overhead. GProM works for lower isolation levels, such as snapshot isolation and read-committed. Watermelon [28] generalizes why-provenance to debugging distributed systems and proposes wat-provenance (why-across-time provenance), which also requires determinism. Chen et al. [3] propose network provenance to debug computer networks at Internet scale. However, prior work focuses either on database queries or application logic, and thus cannot faithfully replay database-backed applications or support retroactive modifications. We believe that TROD could adapt and extend these ideas from prior provenance research to further optimize the tracing and representation of essential information for debugging distributed applications.

Bug Replay and Retroactive Programming. Arnold [6] and OmniTable [25] capture detailed lineage of the entire system state at the granularity of individual instructions and every memory and register state of processes, which allows them to reproduce any transient state in the past. They do not support distributed applications, but TROD can borrow ideas from these systems to handle external calls and non-deterministic events. Pensieve [29] reproduces distributed systems failures using system logs and application bytecode, but it requires developers to provide sufficient information in the error logs. TROD's retroactive programming is inspired by Retro-λ [19], an event-sourced platform that supports retroactive programming for serverless applications, but it only considers a single isolated microservice and does not support transactions.

7 Conclusion

Debugging distributed online web applications is challenging. Current practices require intensive manual logging and guesswork. In this paper, we proposed TROD, a novel transaction-oriented debugger for distributed web applications. We showed that TROD provides always-on tracing and declarative debugging, faithfully reproduces any concurrency bugs, and allows developers to test modified code retroactively on any past event. We encouraged the database and systems communities to drastically rethink the synergy between the way people develop and debug applications.

References

[1] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A Generic Provenance Middleware for Queries, Updates, and Transactions. In *TaPP 2014*.
[2] AWS. 2022. AWS Lambda. https://aws.amazon.com/lambda/.
[3] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. 2017. Data Provenance at Internet Scale: Architecture, Experiences, and the Road Ahead. In *CIDR 2017*.
[4] Fernando Chirigati and Juliana Freire. 2012. Towards Integrating Workflow and Database Provenance. In *IPAW 2012*. Springer, 11–23.
[5] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. 2022. ALASTOR: Reconstructing the Provenance of Serverless Intrusions. In *USENIX Security 22*. 2443–2460.
[6] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *OSDI 2014*. Broomfield, CO, 525–540.
[7] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *ICDE 2015*. 125–136.
[8] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *MSR 2014*. 62–71.
[9] Nuno Faria, José Pereira, Ana Nunes Alonso, and Ricardo Vilaça. 2021. Towards generic fine-grained transaction isolation in polystores. In *Poly 2021*. 29–42.
[10] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS 2019*. 19–33.
[11] Jim Gray. 1986. Why do computers stop and what can be done about it?. In *SRDS 1986*. Los Angeles, CA, USA, 3–12.
[12] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972
[13] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.
[14] Peter Kraft*, Qian Li*, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2022. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. https://doi.org/10.48550/ARXIV.2208.13068
[15] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (2021), 3348–3361.
[16] Qian Li, Peter Kraft, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis,

Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. A Progress Report on DBOS: A Database-oriented Operating System. In *CIDR 2022*.

[17] MediaWiki. 2012. Prevent creation of items having the same sitelinks (duplicates). https://phabricator.wikimedia.org/T44325.

[18] MediaWiki. 2014. Several history entries for the same content and watchlists showing wrong article size changes. https://phabricator.wikimedia.org/T39225.

[19] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-λ: An event-sourced platform for serverless applications with retroactive computing support. In *DEBS 2018*. 76–87.

[20] Moodle. 2017. Course restore can fail if it includes discussion forums. https://tracker.moodle.org/browse/MDL-60669.

[21] Moodle. 2017. Duplicate forum subscriptions due to race conditions. https://tracker.moodle.org/browse/MDL-59854.

[22] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2013. Data debugging with continuous testing. In *ESEC/FSE 2013*. 631–634.

[23] Joseph P Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *ICSE 2016*. 947–958.

[24] Zhengyi Qiu, Shudi Shao, Qi Zhao, Hassan Ali Khan, Xinning Hui, and Guoliang Jin. 2022. A Deep Study of the Effects and Fixes of Server-Side Request Races in Web Applications. In *MSR 2022*. 744–756.

[25] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the OmniTable Way. In *OSDI 2022*. Carlsbad, CA, 357–373.

[26] New Relic. 2022. Transactions in New Relic's APM. https://docs.newrelic.com/docs/apm/transactions/intro-transactions/transactions-new-relic-apm/.

[27] Stackify. 2022. Centralized Log Management. https://stackify.com/retrace-log-management/.

[28] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M Hellerstein. 2018. Debugging distributed systems with why-across-time provenance. In *SoCC 2018*. 333–346.

[29] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *SOSP 2017*. 19–33.

[30] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *SIGMOD 2021*. 2653–2666.