HawkEDA: A Tool for Quantifying Data Integrity Violations in Event-driven Microservices

Prangshuman Das
University of Copenhagen
Copenhagen, Denmark
nlx677@alumni.ku.dk

Rodrigo Laigner
University of Copenhagen
Copenhagen, Denmark
rnl@di.ku.dk

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

ABSTRACT

A microservice architecture advocates for subdividing an application into small and independent components, each communicating via well-defined APIs or asynchronous events, to allow for higher scalability, availability, and fault isolation. However, the implementation of substantial amount of data management logic at the application-tier and the existence of functional dependencies cutting across microservices create a great barrier for developers to reason about application safety and performance trade-offs.

To fill this gap, this work presents *HawkEDA*, the first data management tool that allows practitioners to experiment their microservice applications with different real-world workloads to quantify the amount of data integrity anomalies. In our demonstration, we present a case study of a popular open-source event-driven microservice to showcase the interface through which developers specify application semantics and the flexibility of *HawkEDA*.

CCS CONCEPTS

• **Information systems** → *Database management system engines.*

KEYWORDS

microservice, event-driven architecture, data integrity

ACM Reference Format:
Prangshuman Das, Rodrigo Laigner, and Yongluan Zhou. 2021. HawkEDA: A Tool for Quantifying Data Integrity Violations in Event-driven Microservices. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3465480.3467838

1 INTRODUCTION

The emergence of cloud computing as a paradigm for large-scale deployments has made a case for practitioners to rethink how traditional applications, those following the so-called monolithic architecture, are designed and deployed. Particularly, we are witnessing the growing adoption of the *microservice architectural style* [10].

In contrast to a monolithic architecture, where application modules are coupled and communicate with each other via function calls, a microservice architecture promotes designing modules as independent applications decoupled from each other. The interaction between microservices occurs via lightweight mechanisms like HTTP-based protocols or asynchronous messages. As these modules are independent, changing, testing, and redeploying a functionality is often less complex since only the affected microservice is involved instead of the entire application. Microservices can also be polyglot, using different programming languages and underlying databases (the so-called decentralized data management principle). Besides, microservices promote fault isolation, where faults can be constrained to an individual microservice boundary (e.g., through a container-based deployment), curtailing failures from propagating to other architecture components.

1.1 Challenges and Motivation

Although microservice architectures present several benefits, such as fault isolation and scalability of individual components, designing microservices is, however, a non-trivial task.

C1. Asynchronous event streams. Even though microservices are supposed to operate independently, they often present functionality dependencies among each other [7].

Practice #1: To avoid coupling microservices through their defined APIs, practitioners are increasingly relying on asynchronous events [6], often backed up by message queues like Kafka [5], to communicate a need or trigger a computation in another microservice (e.g., as part of a workflow).

Practice #2: In addition, it is often the case where substantial data management logic is implemented at the application-level in microservices [7]. In other words, as the microservice state is oblivious to the underlying databases, the microservice becomes a stateful application.

Asynchronous events in microservice architectures pose challenges to maintaining application safety, particularly on enforcing ordering constraints on processing distinct events generated by different sources. As these events trigger updates in a microservice's private state, the event processing order plays a significant role in ensuring data integrity.

Ensuring event-based constraints at the applications-level is a complex and error-prone task. Therefore, developers tend to rely on weaker models, such as eventual consistency, that fall apart on enforcing event-based constraints. This first observation leads us to the question **#1**: *How harmful the lack of event-based constraint enforcement is to application safety?*

C2. Cross-microservice coordination. Besides, it is usually the case where a microservice needs to coordinate with another microservice to carry out a cross-microservice business transaction.

This may be in two forms: (i) Querying data from a different microservice. For example, in an online shopping application, the *order* microservice may retrieve the available discounts for a given

group of users from the *user* microservice. (ii) Locking items belonging to a different microservice. For example, the *cart* microservice may request the *stock* microservice to reserve a stock item.
Practice #3: Because of Practice #2, application-level concurrency control is a prevalent practice in microservices, creating a great barrier to ensure application integrity.

To achieve higher performance, practitioners end up implementing cross-microservice requests under a weak isolation level and this may lead to data integrity violations subject to the contention level of workloads. The widespread implementation of cross-microservice requests takes us to the question **#2**: *What are the trade-offs of encoding weak isolation requests across microservices regarding data integrity and performance?*
Motivation. It is difficult for microservice developers to experiment with different design decisions (e.g., how to co-locate data and logic across microservices), not to mention measuring their impact on performance and correctness in their applications. In microservices, this becomes more pressing since, instead of only reasoning about the chain of function calls and such implications in a single database state (i.e., a monolithic application), developers have to reason about the complex interplay of events and requests spanning multiple microservices. As a result, they encounter challenges in identifying the most appropriate application safety and performance trade-offs and resorting to suboptimal designs.

1.2 Contributions

While existing work has focused on investigating application-level concurrency control [2] and performance inefficiencies [9] in monolithic web applications, no work has focused on supporting microservice developers to reason about the trade-offs on designing microservices. To fill this gap, we present *HawkEDA* [1] [2], a data management tool to support microservice developers to experiment with and reason about application safety and performance trade-offs of event-driven microservices. To meet this goal, a microservice developer specifies its application through two building blocks:
(i) Application invariants – An invariant expresses the application correctness criteria. For instance, a developer may specify that an order containing a product item experiencing unavailability in stock should not observe a `payment_processed` event. The observation of such an event under the specified condition incur in an anomaly.
(ii) Application interfaces – Express the interfaces through which the tool can interact with and reason about the application, including the events spanning multiple microservices, their schemas, the APIs to interact with, and the message broker's (that enables the microservice's event-based communication) to connect to.

To the best of our knowledge, *HawkEDA* is the first online monitoring tool targeted for capturing data integrity anomalies in event-driven microservice architectures, allowing for prompt identification of anomalies arisen from concurrency control handled at the application-level. The demonstration of *HawkEDA* will guide users on the lifecycle of monitoring the complex arrangement of event-driven microservices. We will allow participants to modify the workload characteristics, e.g., data skewness and amount of requests, and observe various metrics such as end-to-end latency, throughput, and data integrity violations as the workload evolves.

[1]Hawks usually surprise their victims with unexpected behavior, bringing to light the weakness and vulnerabilities of the chosen victims in their environment
[2]Available at: https://github.com/LennoxAlexion/HawkEDA

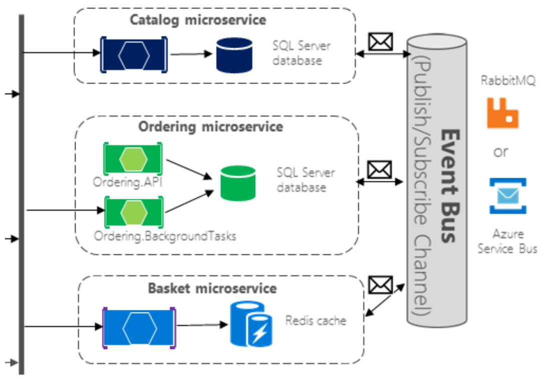

Figure 1: Microservices investigated (adapted from [1])

2 CASE STUDY

In light of the challenges introduced in § 1.1, in this section, we discuss how these are revealed in practice through a case study of the event-driven microservice application *eShopContainers* [1], a popular open-source repository that serves as a reference for microservice developers.

In this case study, we focus on the following microservices: (i) *Catalog*, responsible for managing available products; (ii) *Ordering*, responsible for the processing of orders, and; (iii) *Basket*, entailed for keeping track of user's basket items. The application relies on a message broker interconnecting the defined microservices through asynchronous events. The events may cut across several microservices and are the building blocks driving the execution of business transactions in the application. Our target microservices are shown in Figure 1, which exhibits a partial view of the *eShopContainers* architecture. We discuss the challenges next.
C1. Asynchronous event streams can create problems to applications with *eShopContainers'* characteristics. Consider the case when item price update events are submitted asynchronously by the *Catalog* microservice and, concurrently, the *Basket* microservice receives a request for order checkout. In this case, the *Basket* microservice packages all basket items for the given user, considering the last known price for each, before forwarding such request to the *Ordering* microservice. However, without enforcing an ordering constraint, it is unknown in which order both events should be processed by the *Basket* microservice, which affects the guarantee of reflecting the latest prices in a request. The possible interleaving of event streams can lead to any ordering, totally dependent on the eventual arrival of events.
C2. Although the use of asynchronous events to trigger updates in different microservices is becoming a trending practice in industry settings [6], we also witness the need for synchronous coordination, particularly in cases where competition for resources takes place. Consider the case when concurrent users are placing orders with intersecting catalog items. Under contending workloads, synchronization mechanisms become necessary to safeguard catalog items from being oversold. However, in practice, several practitioners end up eschewing synchronization mechanisms to safeguard data integrity and end up experiencing anomalies.

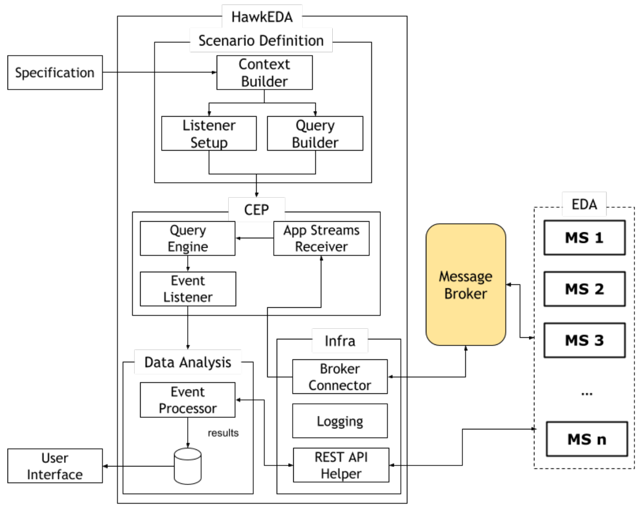

Figure 2: Architecture of HawkEDA

3 DESIGN AND IMPLEMENTATION

HawkEDA's design reflects the characteristics found in event-driven microservice architectures: (i) It should offer appropriate flexibility to adapt to and process the distinct set of event streams exhibited by microservices; (ii) It should allow users to specify what invariants are expected to be enforced by the microservices, such as event ordering constraints, and; (iii) It should allow real-time detection of data integrity violations. *HawkEDA* is implemented in Java and presents an interactive shell interface so developers can input their application's specifications. The tool presents a component-based design, exhibited in Figure 2, which we discuss next.

Scenario Definition. To support users specifying their application invariants and characteristics in the tool, a fundamental building block is a *scenario*. For instance, a user may specify that *event1* is expected to precede any occurrence of *event2* given a specified *key* present in both event schemas, thus specifying a dependency relation. Besides, a scenario also specifies whether the tool runs continuously or not, an important feature to support users testing their event-driven microservices in staging environments before deployment in "production."

Every scenario class must implement the *ScenarioInterface* to instantiate the queries operating over the application event streams, set up the required scenario environment, including but not limited to cleaning memory buffers from past executions and creating synthetic data for session-based testing. For instance, a scenario should define the characteristics of the workload such as data skewness, dataset to operate over, and scale factor of the underlying microservice databases. Such fine-tuning of parameters is allowed through the interface definition, enabling users to create an artificial load that reflects their real-world scenarios.

An example scenario definition for **C2** is shown in **Listing 1**. A `checkout_cart` event is issued by *Cart* microservice after an user's checkout request and includes the catalog items requested. A `payment_processed` event confirms that the order was correctly processed by the *Payment* microservice. The invariants define that (1) no more than one `payment_processed` event should be observed per user checkout and (2) overselling catalog items should not be observed. It is noteworthy that invariant 1 is independent and, without a dependency relation, overselling may be incorrectly detected by the tool. According to *eShopContainer*'s semantics, one can only assert whether an order was placed (and therefore its items) after the payment has been issued. Therefore, to capture such dependency relation, a precedence relation can be established as an invariant, as described by *invariant #2*, which shows that the tool can only reason about oversold catalog items after a payment is issued.

Listing 1: Scenario definition example

```
# Workload definition
(i) Load products to Catalog microservice
(ii) Load users to User microservice
# Execution definition
Define skewness of requests over data items
Define distribution of users submitting requests
Define amount of time of scenario execution
# Events definition
A. checkout_cart { checkout_id, items[{ product_id, ... }] }
B. payment_processed { payment_id, checkout_id, ... }
# Invariants definition
1. SUM (payment_processed) = {0,1} FOR EACH checkout_id
2. SUM (checkout_cart.items[i.product_id] where i ∈ items)
    <= <product amount> (ref. to (i))
    AFTER payment_processed WHERE checkout_cart.checkout_id
       = payment_processed.checkout_id
```

Complex Event Processing. The case for monitoring microservices' behavior over data streams brings the opportunity to take advantage of a well-established approach for detecting patterns from streams: complex event processing (CEP) [8].

After receiving the user's input, the *scenario definition* module sets up a contextual information of the application, including the message broker server information, queues to connect to, and necessary APIs for communicating to. Then, the module builds the CEP queries to detect the constraints specified. The application context is packaged and forwarded to the CEP component.

The queries are then instantiated through the CEP query engine, responsible for continuous processing of application-defined streams. To enable CEP queries in our tool, we use *Esper* [4], a Java-based library that allows embedding full-featured CEP capabilities in a Java application. The queries are defined via an Event Processing Language, making a sufficient abstraction for expressing events and their relations over time as found in our case.

The stream receiver module is responsible for connecting to a queue service (e.g., message broker), subscribing to specified application streams, and ingesting those streams into the CEP query engine. For last, an event listener module is responsible for receiving the output streams of the defined CEP queries and forward these to the data analysis component. More specifically, the output reflects data integrity anomalies observed in microservices behavior.

Data Analysis. The resulting streams from the CEP query engine are shipped to the data analysis component, comprised of an event processor module and a data store that accumulates the observed anomalies from experiments ran over time. The event processor module becomes necessary because some detected patterns over the streams must be combined with data retrieved from microservices' APIs to reason about the existence of anomalies. For instance, when

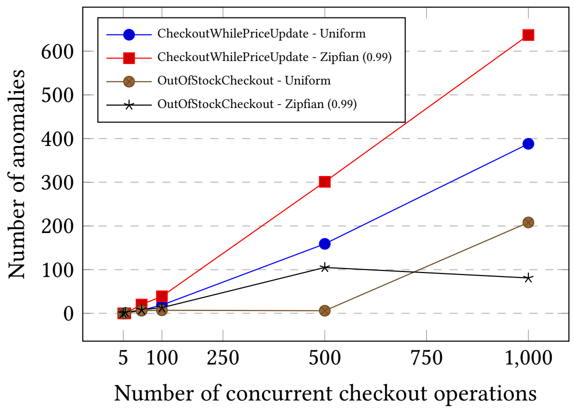

Figure 3: Number of anomalies under different workloads

the outcome of an operation is not materialized as an event, as in the case of verifying which price has been applied to a product (**C1**), it becomes necessary to query a microservice through its API to obtain the result of the operation.

Infra. To decouple the event analysis and CEP components from functionalities that concern hard-wiring *HawkEDA* with a set of monitored microservices and, at the same time, to allow a higher level of extensibility to our tool (e.g., future support for other communication mechanisms, such as RPC), we opted to encapsulate infrastructural aspects, those that provide supportive features to *HawkEDA* components, into a separate component. For instance, the *REST API Helper* module provides communication capabilities that encapsulate the HTTP-based protocol necessary to interact with the microservices.

4 DEMONSTRATION SCENARIO

In our demonstration, we would like to highlight the characteristics discussed in § 3, specifically, the ability to adapt to heterogeneous event-driven microservice applications and process event streams to identify violations in real-time. We propose two demonstration scenarios, one for each challenge introduced in § 2. For both scenarios, the audience is guided on how to use the tool to specify the application events, constraints, and the workload characteristics. A screen shows the amount of data integrity violations observed while the tool and the microservices are executing in real-time.

C1. For the unsafe interleaving of streams scenario, we invite the audience to specify an expected correct ordering of distinct events processing. Specifically, the audience will specify that every *order placement* should reflect price updates issued before the checkout request from a user. In the end, we will be able to respond to question **#1**. Depending on the workload and distribution characteristics defined, the number of anomalies observed may vary. For instance, as exhibited in Figure 3, a *Zipfian* distribution from YCSB's core workload [3] with a Zipfian constant of 0.99 generates one very popular catalog item for contention. Hence, it yields exponential growth in anomalies with an increasing number of concurrent requests. Whereas with a uniform distribution, the items are selected uniformly from a key space of 100 catalog items, and the number of anomalies seems to grow linearly.

C2. For the second scenario, we turn our attention to understand how harmful cross-microservice requests under a weak isolation level are to ensure application integrity. Particularly, the audience is invited to specify that the system should prevent items from being oversold, an interaction taking place between *Order* and *Catalog* microservices. In the end, the users will be able to respond to question **#2**. In this sense, it is noteworthy that depending on the concurrency and skewness experienced by the application, handling concurrency control at the application-level may be a suitable trade-off for many microservice applications. Figure 3 exhibits that, up to 500 concurrent requests, a uniform distribution does not face a substantial amount of anomalies.

5 STATUS AND FUTURE WORK

At the time of this writing, *HawkEDA* only supports the specification of application semantics through the implementation of a Java-based interface. The interface exposes generic methods that are used to instantiate a *scenario*. In the future, we intend to extend the tool to support the specification of an application scenario through loosely structured file formats, like JSON, to achieve higher flexibility and reduced complexity for the developers.

Besides, to make *HawkEDA* more appropriate for a broader audience, we intend to integrate the tool with a graphical user interface while providing more advanced features such as historical analysis of anomalies observed in each application version. Lastly, although different workloads can be specified through a scenario in our tool, we intend to support the specification of volatile workloads within a single scenario to cope with applications with higher fluctuation in data distributions.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement No 801199 and Independent Research Fund Denmark grant No 9041-00368B.

REFERENCES

[1] .NET Application Architecture Reference Apps. [n.d.]. *eShopOnContainers*. https://github.com/dotnet-architecture/eShopOnContainers
[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.
[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*. 143–154.
[4] EsperTech Inc. [n.d.]. *Esper*. http://www.espertech.com/esper
[5] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.
[6] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portorož, Slovenia, Aug 26-28*. 213–220.
[7] 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. arXiv:2103.00170 [cs.DB]
[8] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-Performance Complex Event Processing over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 407–418.
[9] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-World Web Applications. In *Proceedings of the ACM on Conference on Information and Knowledge Management*. 1299–1308.
[10] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (2017), 301–310.