UNIVERSITY OF COPENHAGEN

A thesis presented to the Faculty of Science in partial fulfillment of the requirements for the degree

Master of Science in Computer Science

Enforcing data consistency in event-driven microservices through event-based constraints

Anna Lesniak
sxl492@alumni.ku.dk

Supervisors: Yongluan Zhou and Rodrigo Laigner

June 2021

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my supervisors, Yongluan Zhou and Rodrigo Laigner, for the guidance and insightful feedback. Thank you for all your support and dedication.

I would also like to thank all my friends that I met throughout the last two years at the University of Copenhagen and Dyrlægevej. Thank you for bringing a lot of fun and making the experience truly unforgettable!

At last, I would like to thank my husband, Kacper. I would not be where I am now without your continuous support, exciting ideas and, I cannot stress this enough, your sense of humour.

ABSTRACT

Microservices are an emerging architectural style for designing software applications as a set of self-contained and loosely coupled units with private data stores. Although the individual services are developed and maintained independently, in order to provide certain system functionalities, such as a correlation of distinct events, they often depend on each other. One of the inter-service communication models that are increasingly being employed are asynchronous events. Event-driven microservices generate and publish events that reflect changes in their internal state to other parties that necessitate operating over such data or react to such external changes.

Decentralizing data ownership across microservices has its implications for responding to changes. In order to preserve the flexibility and availability of microservice systems, strong consistency guarantees are usually replaced by the eventual consistency model. This introduces new challenges as application developers need to implement a significant amount of data management logic and validations at the application layer.

This leads to difficulties in reasoning about application invariants and preserving them under diversified event schedules and arbitrary interleavings. To address these challenges, we start by analyzing popular open-source microservice repositories to identify common patterns and shortcomings in application data management that lead to inconsistencies. Based on detected patterns in application encodings, we propose three categories of event-based constraints that serve as an easy-to-use guideline to developers for reasoning about data integrity.

We introduce *Stream Constraints*, a system-level stream processing solution created on top of Kafka Streams that allows developers to specify event-based constraints and enforce them on a stream of events. The evaluation of *Stream Constraints* conducted on a popular open-source event-driven microservice application shows that explicit definition of application invariants eliminates or significantly decreases the number of data integrity anomalies and introduces lower overhead compared to baseline solutions.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

1.1 MOTIVATION

Microservices are becoming a widely used application architecture style for building granular, lightweight, loosely coupled, and service-oriented systems. Although microservices are supposedly designed to operate autonomously, the components usually depend on each other to provide certain system functionalities. If the operations span across multiple microservices and require access to the information provided by different services a system must employ a communication model. In contrast to a monolithic architecture, where an application runs on a single process and its components communicate through invoking language-level methods or function calls, the microservice architecture adopts inter-process communication protocols [30].

A communication model that is increasingly being adopted, as reported by Laigner et al. [20], is asynchronous messaging. The message communication is based on an event-bus or message broker that propagates information between multiple microservices through events. The events can express a change of state or any notable action that took place. The recipients can then execute appropriate operations if the event was significant to the given service. The message-based communication is becoming widely adopted as it brings loose coupling among microservices and establishes clear boundaries between components [20]. It also ensures isolation and allows for further scalability [21]. Additionally, new events can be easily integrated into the existing architecture by simply following the used patterns. Relying on event-based communication also positively influences adaptability and flexibility due to the fact that producers and consumers are being truly separate and therefore simple to change or replace.

While microservices already face various challenges caused by errors or outage of dependent services, the asynchronous nature of communication brings another level of complexity [17]. Event-driven systems may suffer from deadlocks, race conditions, or unintended interleaving of events when supporting concurrent requests that affect the same entity or resource. The use of asynchronous events may also result in unforeseen results due to great diversification of the input considering the number of events produced, their order, or payload. The occurrence of even small defects but within multiple microservices is particularly harmful as it can escalate quickly and spread over the whole system [25].

All of these scenarios have a critical impact on data consistency maintenance within microservices. The main reason behind that is that microservices usually employ a decentralized data management model. Each service encapsulates its own private state that can be modified using well-defined APIs. Any changes to the state are then managed at the application level and it is the application developers' responsibility to reason about data correctness and integrity when designing the application behaviour. When creating message consumers practitioners must take into consideration the application-specific data invariants and rules, prepare the logic to handle interleaved events, unexpected order of events, possible duplicates or missing data [2]. There is a lack of unified support for developers when it comes to reasoning about event-based consistency requirements for their applications [20]. Therefore the same process of creating application-level data constraints is being repeated for every new microservice. The logic once created cannot be easily reused because of application specificity and technology choices. Complexity level of the process and lack of well-defined interfaces that help reason about application constraints may explain state anomalies, uncovered in [20], that are seen across microservice applications.

The arising issues cannot be solved just by using available stream processing mechanisms because their focus is put on querying the stream of data, detecting occurrences of particular patterns, and aggregating or transforming the events. Current solutions lack abstractions to enforce event-based constraints on the stream which would move the data correctness responsibility from the application level to the higher, system level. Uniform constraint interface could be then reused in all of the system components, regardless of underlying technology or type of invariants.

1.2 CONTRIBUTIONS

To address the aforementioned inefficiencies, we first analyze open-source projects that use the microservice architecture and rely on a message-based communication [29, 12, 37]. We look for shortcomings in the application logic that is unsafe under arbitrary stream processing interleaving. We identify functionalities and typical patterns in microservices that may require or benefit from explicitly defined event-based constraints.

Based on the uncovered invariants, we categorize a set of event-based constraints that represent different scenarios where distinct but correlated events can interleave and negatively affect data consistency. We design unified interfaces for the event-based constraints that can be used by application developers and applied in stream processing. The constraints allow for an explicit definition of correctness criteria and relevant dependencies between events. We use the concept of explicit causality [7] where the defined constraints track only those relations that matter from the perspective of a given application. It allows for moving some of the responsibility of the application developers from the application level to the system level. Finally, we design system-level support on top of Kafka Streams API that enforces defined constraints. Applying the constraint system in the microservice environment helps to avoid anomalies in event processing thus alleviating the tasks of developers on handling complex event scenarios.

To evaluate the solution, we select a popular open-source repository as a baseline and perform experiments to understand the effects on the data consistency and the cost of enforcing the event-based constraints.

Partial results of the thesis have been accepted in the Demo and Poster Track at the ACM International Conference on Distributed and Event-based Systems (DEBS) 2021.

1.3 THESIS OUTLINE

The thesis is structured as follows. Chapter 2 presents the background of the research and describes the main concepts such as microservice architecture and event-based systems. Related work in Chapter 3 contains a review of existing solutions in the field of events processing and enforcing data consistency. Chapter 4 is a case study that presents results of the analysis of the open-source repositories. The *Stream Constraints* system is introduced in Chapter 5. Chapter 6 describes evaluation methods and presents obtained results. The outcomes and crucial aspects of the system are further discussed in Chapter 7. Chapter 8 summarizes the study and provides recommendations for future research.

2

BACKGROUND

This Chapter presents the background of the study and discusses problems of application correctness when using message-passing in the microservice architecture.

2.1 MICROSERVICE ARCHITECTURE

Microservices have emerged as an architectural style, addressing how to build, maintain and evolve large systems architectures out of independent, self-contained, and autonomously deployed units. The microservice architecture stands in opposition to monolithic systems that have been traditionally adopted in software systems and which inevitably get larger over time, becoming more complex, risky, and expensive to evolve [21, 23]. In contrast to traditional monolith applications which consist of a single unit that combines user interface, application logic, and access to data, microservice systems are composed of a set of small independent services. By using the microservice architecture developers can focus separately on implementing the basic functionalities in an isolated environment. The components can be then separately managed, changed or replaced. Changing a service implementation has no impact on other services as the communication takes place only via well-defined interfaces [32]. Additionally, using microservices does not limit the number of used technologies, as it is the case in monoliths which usually have a main choice of technology or environment [24]. Therefore, each of the microservices can be built using technologies that are the most suitable, considering the type of operations, workload, etc.

According to [24], a concept of microservices is tightly coupled with the Domain Driven-Design, a set of patterns and good practices where a complex domain is divided into multiple bounded contexts. Each context consists of coherent domain concepts and is usually covered by a single module in the software system, a single microservice in particular [34].

One of the biggest differences between monoliths and microservices lies in communication. In a monolithic application, components are invoked via function calls. Microservices, as a multi-component application, need to employ inter-process communication based on well-defined API. The communication types can be divided into synchronous protocols, like the HTTP where a client sends a request and waits for a response from the service and thus blocks the thread, or asynchronous protocols, based on callbacks or messages, where the client or message sender does not wait for the response.

2.2 DATA MANAGEMENT IN MICROSERVICES

Stateful and data-centric microservices are ubiquitous [13]. Microservices rely on databases to store and manage their state. Although it is possible for multiple services to share a single database, it is considered a serious anti-pattern. This solution eliminates many benefits that the microservice architecture brings, such as loose-coupling and isolation. A shared database requires a lot of coordination between services and considerably decreases fault-tolerance as multiple components depend on the same storage. In order to preserve the features of microservice architecture vast majority of applications employ the database per service pattern [20]. Each microservice independently stores, retrieves, and changes information in its own data store. The ownership can be established at different levels, starting from a service owning private tables in a shared database up to a separate database server [27]. Other services cannot access the data directly but perform requests using the defined APIs. Deploying this solution allows for the adoption of different database technologies, depending on the specific needs of each service.

Although loose-coupling is one of the main principles of microservice architecture, in many cases microservices still depend on each other as they need to collaborate on business logic operations. These processes often span across multiple domains and thus involve multiple services and access to different types of data. In situations when transactions span across multiple services, data integrity and consistency are becoming critical challenges. Since the data operations that need to be coordinated are executed on different databases owned by individual services the application cannot use local ACID transactions [36]. Additionally, due to the great diversity of database technologies that are being adopted and thus the lack of interoperability, using standardized protocols of distributed transactions, such as 2PC, is not possible either [20]. Many modern technologies, for instance NoSQL databases, such as MongoDB and Cassandra, and message brokers, such as RabbitMQ and Apache Kafka, do not support the distributed commit protocols. That creates a trade-off between access to 2PC and the use of modern, well-supported technologies.

The recent review of literature and open-source repositories by Laigner et al. [20] show that many microservice applications give up on strong consistency guarantees in order to achieve higher scalability. One of the means to achieve that is the adoption of the BASE (basically available, soft state, eventually consistent) philosophy [33]. It is an alternative to the traditional ACID (atomicity, consistency, isolation, durability) consistency model. To decouple services and preserve availability but at the same time maintain weak consistency the BASE model incorporates additional event orchestration and organizes computation in an event-driven architecture.

2.3 EVENT-DRIVEN ARCHITECTURE

Along with the transition from monolith systems to loosely-coupled microservices, a flexible event-driven communication replaced the centralized business process orchestration [30]. The main focus of the event-driven architecture lies in events that are published by various microservices. The architecture consists of event producers that communicate changes in the state to other components of the system. The recipient services employ programming primitives that allow for reacting to the predefined events which are relevant from their perspective [21]. Producers are decoupled from consumers as a producer is not aware of consumers listening to its events.

The event-driven architecture can employ multiple models. The events can be sent through direct message queues where an event, after it was received, cannot be replayed. Another common model uses the publish/subscribe mechanism where recipients explicitly subscribe to a specific type of events. The events can be also stored in the form of an ordered and durable log. Existing consumers continuously advance their position and new participants can access all the history and recreate the current state from the beginning.

From the consumer perspective there are three common models of processing the received events [28]:

- Simple event processing. In this model, each of the notable events triggers an action in the consumer.

- Event stream processing. In the stream of events, both ordinary and notable events happen and a consumer operates on continuous series of data by applying various stream transformations that include aggregations (e.g., calculating mean or sum), transformations or data enrichment (e.g., joining multiple data streams).

- Complex event processing. A consumer processes a series of events and matches complex patterns over the events. The patterns often use low-level models of event causality, hierarchies, and ordering. This model is commonly used to detect and respond to anomalies, monitor systems or perform advanced queries over continuous data.

The stream processing solutions are further discussed in Chapter 3.

2.4 MICROSERVICE INTERACTION PATTERNS

The event-driven architecture offers multiple mechanisms that can be applied to coordinate data management in microservice architectures where each individual system encapsulates its own state. One of the simplest coordination patterns that involve asynchronous event communication are domain or integration events.

2.4.1 *Source of Truth*

One of the crucial steps in the process of designing microservice architecture is defining the ownership of domain entities. In a system architecture, one service should represent the source of truth for the entity it manages. Other services that may also need to operate on the same entity need to request the data from the owner for each of the computations or hold their own copy of data. The copy is being continuously updated based on information sent from the data owner and thus eventually consistent. The materialized views are usually employed across the services with close collaboration where efficiency plays an important role and a local copy can speed up the computations. In general, materialized views should never be treated as the main source for querying. Nevertheless, there are use cases when it is acceptable to view data that is not yet consistent, such as analytics or reporting services that store their own copy of the system's data [11].

2.4.2 *Domain Events*

When the domain ownership is established each individual microservice may need to employ a mechanism to inform about changes in its data. One common pattern involves the use of domain events. A domain event is an event that is being published from a microservice declared as the source of truth, as a result of a decision within the domain. The events often correspond to the creation, update, or deletion of main entities in the domain.

Existing studies [33, 35] argue that services must use transactional messaging to publish events to ensure that they are published as part of the transaction that performs the change. It needs to be assured that:

- each of the produced events represents the actual state at the moment of event generation and that means no events are produced when no change is performed,
- each change that has been committed results in a published event.

Domain events are crucial components of more complex patterns that ensure data consistency across microservices. One example is the saga pattern.

2.4.3 *Saga Pattern*

The saga pattern applies to scenarios where unidirectional communication is not enough and there is a need to implement transactions that span across services. A saga is a sequence of local transactions where each individual transaction updates the database and publishes a message or event to trigger the next local transaction in the saga chain [35]. In a case when a local transaction fails the saga executes a series of compensating transactions that undo the changes that were applied so far.

The pattern exists in two coordination models:

- choreography, each local transaction of the saga participants publishes domain events that trigger the next transaction,
- orchestration, the participants do not directly reach out to the next services but there is a common orchestrator that coordinates the transaction process.

2.4.4 *Application-level Data Management*

Although the saga pattern enables applications to maintain data consistency across multiple services without resorting to distributed transactions it requires creating a rather complex programming model and designing the compensating transactions. These may be the reasons why this pattern has not gained much popularity in modern microservices applications, as examined in [20]. Instead, developers choose to rely only on domain events followed by explicit solutions implemented in the application code.

Using the integration events to implement business tasks that span across multiple microservices results in eventual consistency between those services. Lack of strong guarantees moves the responsibility of ensuring data correctness to the application level. As a result, the database systems no longer play a central role in data integrity in microservices.

First of all, in order to ensure data correctness, application developers need to reason about what consistency invariants apply to the context of the application. The invariants are rules concerning the internal state of the application that must be enforced at all times [35]. Let us consider an example application in e-commerce domain that uses asynchronous integration events to coordinate across microservices. For instance, a Basket microservice within this system could explicitly define an invariant that a given customer can have at most one open basket. This is a simple constraint that concerns only data of the Basket service. In order to satisfy this constraint developers may need to introduce additional logic that would validate incoming messages concerning baskets creation.

However, there may be also invariants that require cross-service validations. For instance, in the mentioned Basket microservice, a basket can change its state from `open` to `assembled` only if all the items in it are available. An Items microservice sends messages to the Basket whenever the availability of an item changes. On basket state change the developers may choose not to perform any additional checks assuming that the Basket microservice holds up-to-date information about items availability. Another option would be to communicate with the Items microservice directly before the state change is being saved. In this scenario, the exact consistency invariant could be defined as a given basket in a state of `assembled` has all of its items available.

One of the major difficulties the developers face is realizing and specifying the consistency requirements for their applications in the first place. The invariants are often hidden, encoded in complex application logic and there is a lack of support for that could help in reasoning about consistency requirements in microservice architectures [20]. Additionally, the application-level consistency makes code reuse difficult, as once identified and applied constraints cannot be simply repeated due to the great diversity of application contexts. Developers need to design, implement, and validate consistency mechanisms from scratch for each individual microservice [2].

2.5 CHALLENGES OF ASYNCHRONOUS MESSAGES

The use of asynchronous message communication adds even more complexity to the already complicated process of data management in microservices.

2.5.1 *Delivery Guarantee*

Most modern message brokers guarantee message delivery [4, 1] and some of them additionally offer different options of delivery semantics:

- at-least-once, usually the default semantics that guarantees that a message will be delivered but may be duplicated under some circumstances,
- at-most-once, a semantics where data loss is possible,
- exactly-once, it guarantees that each message processing happens once and only once.

While the exactly-once semantics may be the most desirable it is not provided as a simple switch but the application has to be specifically designed to not violate the property as well, for example by using transactional consumer [4]. This trade-off makes the practitioners lean towards the default at-least-once delivery where message duplicates are possible. One of the approaches to overcome the negative effects is to use idempotent mechanisms that can be applied once or multiple times with the same result. In scenarios where idempotent operations are not possible the developers need to additionally validate and eliminate potential duplicates.

2.5.2 *Message Ordering*

On top of delivery guarantee, message systems offer the ability to ensure that messages are delivered in the order they are received [4, 1]. However, Pritchett [33] argues that at times this guarantee gives only a false sense of security. Even though client requests arrive in the system in an arbitrary order, the processing time required per request usually varies and therefore earlier requests may be finished later than the latter. This results in a situation where domain events are interleaved and no deterministic order can be derived. Application programmers must then ensure correct behavior for any possible event schedule or, if needed, reestablish message sequence.

2.5.3 *Cross-team Collaboration*

Lewis and Fowler [24] point out another important aspect of coordination mechanisms in a microservice architecture. Management of microservices in practice relies on the notion that a single team should own service over its full lifetime. As a result, any communication between services often incorporates collaboration across multiple developers teams. When using synchronous calls it is in the interest of a given development team to understand all possible responses of the external request as they need to ensure proper response handling in all scenarios. The process is very different when it comes to asynchronous communication. The owners of the producer service may be aware of possible anomalies in the produced stream of events but the produced data is usually consumed elsewhere. Therefore, these nuances may be lost if the collaboration between the teams is not close enough and the development of the subscriber will be based on assumptions that may be not always true.

3

RELATED WORK

Achieving data consistency in event-driven microservices is a difficult challenge as it requires a solution that combines components across multiple domains:

- stream processing that allows applications to analyze and detect anomalies in published data across microservices,
- application invariants that are used to explicitly define rules for data correctness in the application,
- preserving or enforcing the invariants without resorting to strong consistency protocols.

In this chapter we go through existing solutions in these fields to summarize concepts they build upon and discuss their applicability in enforcing data consistency in microservices.

3.1 STREAM PROCESSING FRAMEWORKS

The ability to handle and process unbounded streams of data is emerging as a significant need in analytics and service coordination within event-driven systems. There are multiple stream processing frameworks that respond to that common need yet offer various programming models, processing mechanisms, and data models [16].

In stream processing, each individual record is processed as it arrives. It stands in contrast to batch processing where the data has to be first accumulated and stored before the computations are applied. This can be done periodically or after reaching a given volume threshold. However, modern distributed systems often require that the data is processed in a near real-time fashion, as there are multiple factors that depend on it, for instance data consistency or the time to detect an anomaly. Most existing processing frameworks employ a model where records are processed one at a time, but there are also solutions that draw from the concept of batching and form mini-batches - groups of messages that arrive within a short time and are processed together. An example of that approach is employed by Spark Streaming [5].

The processing model of one record at a time makes it difficult to perform operations that involve reordering or sorting. Although it is possible to perform stateful, context-dependent operations, such as aggregations, the operations that change the order of messages require implementing custom buffering logic and, for instance, sorting within a batch of messages. Additionally, the stream processing frameworks impose a dataflow model defined homogeneously where each individual record is processed identically. This contrasts the heterogeneous nature of microservices that change and evolve independently.

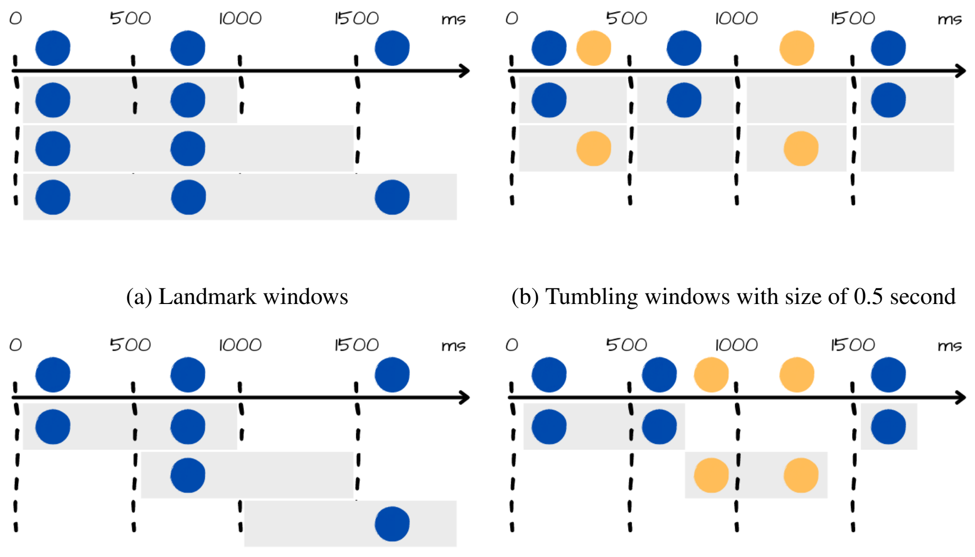
Modern stream processing frameworks offer some type of querying mechanism that allows for finding output events of interest or compute real-time analytics. There is a group of engines that use query languages that are based on SQL, a standard language for manipulating and retrieving data in databases. By transforming and expanding the SQL, the concept of CQL (continuous query language) emerged that allows for performing operations on a stream of events [6]. ksqlDB [19] is a stream processing engine that employs a lightweight SQL syntax and therefore enables users to leverage a SQL-like interface with statements like SELECT or JOIN.

Other group of frameworks does not introduce a designated language but allows the developers to create stream processors with the processing logic in one of the supported programming languages. For instance, Kafka Streams [18] is a library built on top of Apache Kafka's storage and messaging layer that provides Java and Scala APIs to define stream processors: the Domain Specific Language with built-in operations, such as map or filter, and the customizable, low-level Processor API. Apache Flink [3] combines both approaches and does require some amount of code but also supports SQL statements.

The defined queries may be executed on an unbounded stream of data and result in either [18]:

- another stream (stream-to-stream), where the output is also an unbounded stream of events but has been transformed, for instance filtered, mapped to a different type, or joined with another stream,

- table (stream-to-table), where each data record in the stream captures a state change of the table. A table is usually a result of the data records aggregation, for instance calculating mean value.



(a) Landmark windows

(b) Tumbling windows with size of 0.5 second

(c) Hopping windows with size of 1 second and advance interval of 0.5 second

(d) Session windows with gap of inactivity equal to 0.5 second

Figure 1: Examples of stream windows

The queries may be also run against bounded subsets of a stream - stream windows. The windows define a limited scope for the queries and can be specified over time (e.g., every hour or minute), number of messages, or other breakpoints. One can distinguish multiple types of windows, such as fixed (e.g., from the beginning to the end of given period), landmark, tumbling, hopping, or session windows based on the period of activity separated by a defined gap of inactivity [10], see examples in Figure 1.

The real-time processors need to deal with errors in the data stream, such as a significant latency, data loss, or out-of-order delivery. To address this the stream processing frameworks have dedicated mechanisms that handle such stream imperfections. On an example of windowing operations, it may happen that events that belong to a window arrive out-of-order because of a slow network. The stream processing frameworks need to apply an additional grace period and ensure that the events are accepted into the window. Kafka Streams and ksqlDB offer explicit grace period mechanisms but all of the mentioned frameworks take this into consideration [18, 19]. The processing models lack, however, custom abstractions that could allow for defining what stream *correctness* is, so that not only the timestamps are taken into consideration but also the types or payloads of the events in the stream.

3.2 COMPLEX EVENT PROCESSING

An increasing number of applications that require processing high-throughput streams and increasing complexity of this data lead to the expansion of stream processing models into Complex Event Processing (CEP). CEP emerged as a solution for extracting complex information from event-based systems by detecting occurrences of low-level patterns that reflect some high-level scenarios [26].

Several CEP languages have been developed that allow for defining queries that address both occurrences and non-occurrences of events and specify their order or time windows. The languages also support parameterized predicates for correlating events through their value attributes [10]. The CEP engines are responsible for combining individual events and transforming them into new composite events that can be then consumed by external applications. Wu et al. [40] design SASE, a complex query language that uses native operators to efficiently detect event sequences, see example in Figure 2.

```
EVENT    SEQ(MEDICINE-TAKEN x, MEDICINE-TAKEN y)
WHERE    [name='John'] ∧
         [medicine='Antibiotics'] ∧
         (x.amount + y.amount) > 1000
WITHIN   4 hours
```

Figure 2: CEP query that detects a sequence of events that indicates an overdose of antibiotics [40]

The most typical applications of CEP involve monitoring, business process automation, and control systems [10]. Therefore, the main goal of the execution of CEP queries is to detect the complex patterns, involving sequencing and ordering relationships, and immediately notify monitor applications when a given scenario has been observed. The CEP systems are not designed to change the detected pattern but to transform the data to a composite event and deliver it to consumer that will take necessary action. In the microservice architecture, Complex Event Processing can be used then as a mechanism for consistency anomaly detection but it lacks the ability to change or correct the identified errors. CEP does not provide designated abstractions for specification and enforcement of constraints over streams. Nevertheless, the sequential and value-based constraints of CEP, for instance the `SEQ` operator or correlating the events by their attributes, have potential to be expanded into constraints that can be actively enforced on a stream of data and not just passively detected.

3.3 EVENT CONDITION ACTION

The Complex Event Processing builds upon another execution model, the Event Condition Action (ECA) [10]. The ECA model consists of a set of rules and actions associated with them. There are three components in an ECA rule:

- event, which defines what triggers the rule; it can be a composite event, as defined in CEP,
- condition, which specifies what conditions the events need to satisfy in order to trigger an action,
- action, which defines what steps need to be taken for a given pattern.

Obweger et al. [31] propose an ECA model that allows for manipulating business entities based on defined rules. The state is encapsulated in business entity providers that provide access to managed data through easy-to-use interfaces. The interfaces are used in the rule components that, whenever activated, directly and synchronously call user-defined update functions on the business entity providers.

The ECA models could be then applied to microservice data management where for each possible erroneous event scenario a compensating action is defined. However, the ECA was developed to address the need of reacting to different scenarios of events occurring in databases. In such heterogeneous environments as microservices, each of the services would have to expose additional interfaces that could be called by the ECA application. The set of restorative interfaces will not allow for decreasing the complexity of application-level logic but change the focus from preserving the correctness of data to reacting to anomalies.

3.4 BUSINESS PROCESS MANAGEMENT

Hildebrandt et al. [15] propose a declarative, event-based process model for defining the control flow as a set of constraints or rules. The authors note that a declarative approach has a great advantage as it allows for more flexible process descriptions, in contrast to the imperative models that tend to be over-constrained. It is indeed applicable in the event-based systems because of the great variety of events and interleaving of them that makes it challenging to list all event schedules that are allowed. Therefore, any execution fulfilling the constraints of the workflow is allowed, thereby leaving maximal flexibility in the execution.

The use of the declarative model in message-based data management could be very beneficial as it allows the practitioners to focus on the most crucial constraints that have to be satisfied in order to preserve data correctness and leave other event scenarios uncontrolled.

The proposed model of DCR Graph consists of a directed graph with nodes representing the events that can happen and arrows representing relations between them. There are multiple relation types, for instance a condition relation that describes what are the preconditions for a given activity or a response relation that specifies what events must happen afterwards. The DCR Graph defines what must happen in order for the flow to be accepted. However, in case of any constraint violations, the process will not be able to reach an accepting state and the model itself does not offer any enforcing mechanisms that could alter the events without any external input.

3.5 IMPROVING DATA CORRECTNESS

Achieving data consistency in data-driven applications has been widely studied and many mechanisms have been proposed.

3.5.1 *Synchronization Mechanisms*

One of the solutions that employ synchronization mechanisms is Synapse [39], a framework for structuring heterogeneous-database applications that operate on the same data. It provides a data replication semantics that synchronizes materialized views in microservices. Synapse is based on the publish/subscribe protocol and it transparently intercepts data updates, compiles their dependencies and publishes them to the receiver. Synapse gives the developers an option to choose one of the delivery semantics:

- global, all object updates are sequentially ordered,
- causal, all updates to the same object, within the same user session and controller are serialized,
- weak, all updates to the same object are sequentially ordered, but there is no ordering guarantee regarding updates to different objects.

Nevertheless, Synapse is designed on an application level and therefore needs to provide distinct support for programming languages and database systems. A number of the most popular technologies are supported but the solution may be not applicable for every service. Additionally, the ability to specify specific constraints is limited as the practitioners have the option to choose one of the predefined delivery semantics but the Synapse imposes many underlying concepts, for instance the definition of dependency between objects.

3.5.2 *Application Invariants*

Improving data consistency through explicitly defined application invariants has been studied and resulted in systems that allow for specifying and additionally resolve the constraints. Indigo is a middleware that provides Explicit Consistency based on the defined invariants [9].

```
 1 @Invariant(”forall(P: p, T: t) :− enrolled(p, t) => player(p) and tournament(t)”)
 2 @Invariant(”forall(T: t) :− nrPlayers(t) ≤ capacity”)
 3 @Invariant(”forall(T: t) :− active(t) => nrPlayers(t) ≥ 1”)
 4 @Invariant(”forall(T: t, P: p) :− active(t) and enrolled(p, t) => participant(p, t)”)
 5 public interface ITournament {
 6     @True(”player($0)”)
 7     void addPlayer(P p);
 8
 9     @True(”tournament($0)”)
10     void addTournament(T t);
11
12     @True(”enrolled ($0 , $1”)
13     @False(”participant ($0 , $1)”)
14     @Increments(”nrPlayers ($1 , 1)”)
15     void enrollTournament(P p, T t);
16 }
```

Figure 3: Invariant specification for an example tournament application that uses Indigo [9]

Figure 3 shows an example of a tournament application that defines methods for creating players and tournaments and allows players to enroll in a tournament. Additionally, the application specifies a few invariants that must be true at all times, for instance: for each existing enrollment the involved user is present and so is the tournament (line 1).

The Indigo identifies which operations would be unsafe under concurrent execution and proposes two techniques to address them: violation-avoidance that relies on an exclusive reservation system or invariant-repair that allows operations to execute without restriction and restore invariants by applying compensation operations to the resulted state.

Balegas et al. [8] propose an extension to the Indigo system with deterministic conflict resolution policies based on static analysis. The authors devise an algorithm that detects executions that might violate an application invariant. Then modifications are proposed that will prevent those violations. If there are multiple alternative modifications for preserving invariants the programmers must select the most appropriate one.

The presented solutions give strong grounds to reason about application invariants and preserving data integrity. However, the techniques do not target event-driven applications and mainly focus on concurrent updates without considering other anomalies, such as data loss or out-of-order operations. In order to apply them to event-based microservice systems, it would be beneficial to create system-level abstractions that would allow the modifications to be extracted from the application logic and moved to the event processing layer.

4

CASE STUDY

To understand problems that arise in the complex interplay of event-driven microservices, we analyze multiple popular open-source applications [12, 29, 37]. We investigate application logic patterns that are unsafe under arbitrary event interleaving, seeking assumptions and common techniques for preserving data consistency. In this Chapter, we illustrate examples of implicit event constraints and discuss their potentially harmful impact on data consistency based on the Lakeside Mutual application [29].

4.1 LAKESIDE MUTUAL

Lakeside Mutual is an example of a microservice application created with the Domain-Driven Design [29]. The application provides several digital services for managing insurance policies, customer data, calculating risk, and could be potentially adopted by a real-world insurance company. Lakeside Mutual consists of 11 microservices, both frontend and backend applications, mostly developed with Java and JavaScript languages. The system employs the database-per-service pattern and each individual service owns a specific domain context, e.g. customers, policies, or insurance risk.

The Lakeside Mutual microservices use a few different means of communication. Most of the service-to-service communication is handled by the HTTP protocol, sometimes also with the use of WebSockets for full-duplex channels. Communication between the Risk Management client and server is based on the gRPC, an open-source framework that implements the Remote Procedure Calls API.

Three of the microservices use only asynchronous message passing as a mean of communication between each other. As presented in Figure 4, data between the Customer Self-Service Backend and the Policy Management Backend is exchanged in both directions, while the Risk Management Server acts only as a message consumer. In the original source code, the application uses the Apache ActiveMQ as a message broker and relies on the ActiveMQ queue concept. Queues are FIFO pipelines of messages created by producers and then consumed by clients, one message at a time, and removed from the queue.

For the purpose of this study, the ActiveMQ queue-based communication was replaced by Apache Kafka, a publish-subscribe messaging system. The changes were made to the minimum extent possible in order to maintain the original application behaviour. The producers, consumers, and messages payload remained unchanged but the message queues were changed into Kafka topics. Created topics with corresponding producers and consumers are presented in Table 1.

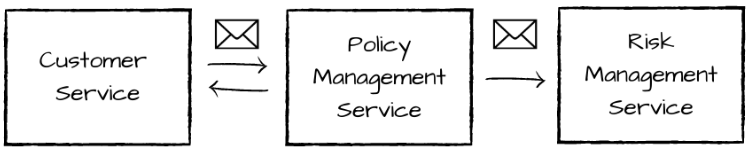


Figure 4: A subset of the Lakeside Mutual microservices that communicate via asynchronous messages

Topic	Producer	Consumer
`insurance-quote-request-events`	Customer Self-Service	Policy Management
`insurance-quote-response-events`	Policy Management	Customer Self-Service
`customer-decision-events`	Customer Self-Service	Policy Management
`policy-created-events`	Policy Management	Customer Self-Service
`insurance-quote-expired-events`	Policy Management	Customer Self-Service
`policy-events`	Policy Management	Risk Management

Table 1: Lakeside Mutual topics with corresponding producers and consumers

The following sections describe existing challenges in data management and current solutions detected in the Lakeside Mutual application architecture and code base.

4.1.1 *Policy Updates and Deletions*

The Risk Management service allows the insurance company employees to download a customer data report that contains risk assessments of existing customers. The Risk Management server subscribes to a single topic that contains two types of messages:

- `UpdatePolicyEvent`, a domain event that is sent every time a new policy is created or an existing policy is updated,
- `DeletePolicyEvent`, an event created every time a policy is deleted.

There is no separate event that reflects policy creation. The first `UpdatePolicyEvent` that is processed for a given policy is considered as a creation of the policy.

The Risk Management server uses consumed events to construct a materialized view of the policy data. When the Risk Management Client requests a risk report of current customers, the service queries its own copy of the data that is persisted in a JSON-formatted file where for every customer their existing policies are stored. The Risk Management server, when consuming the policy messages, does not perform any additional checks on saved data and simply overwrites existing policies by the consumed payload. If the policy was not present it is assigned to the corresponding customer. If the policy is present but a `UpdatePolicyEvent` was received, the policy data is overwritten by the consumed data. The deletion events result in the corresponding policy being removed from the customer.

The policy events are produced by the Policy Management service. The server provides a REST API that handles HTTP requests for retrieving and manipulating the policy data, which consists of three methods: `createPolicy`, `updatePolicy` and `deletePolicy`. While the logic of the deletion is rather straightforward, the other methods perform a sequence of customer and policy data retrievals that also involve performing calls to other microservices of the application such a the Customer Core.

The Policy Management is a microservice based on the Spring framework therefore it includes a pre-configured, embedded web server by default - the Apache Tomcat. It allows for a specific number of simultaneous requests that can be handled by the application and it is set by default to 200. It translates into the Policy Management being able to handle multiple policy creations, updates, and deletions at the same time.

The application-level code of the Policy Management service does not introduce any concurrency control mechanisms. The lack of explicit transactions causes consistency problems even in the data of this single service. However, this analysis addresses emerging consistency issues between multiple services in an event-based architecture and that is the main focus of the following observations.

The eventual delivery of the stream of policy changes creates a challenging scenario for developers and may lead to an inconsistent view. Figure 5 shows one of the possible scenarios that involve

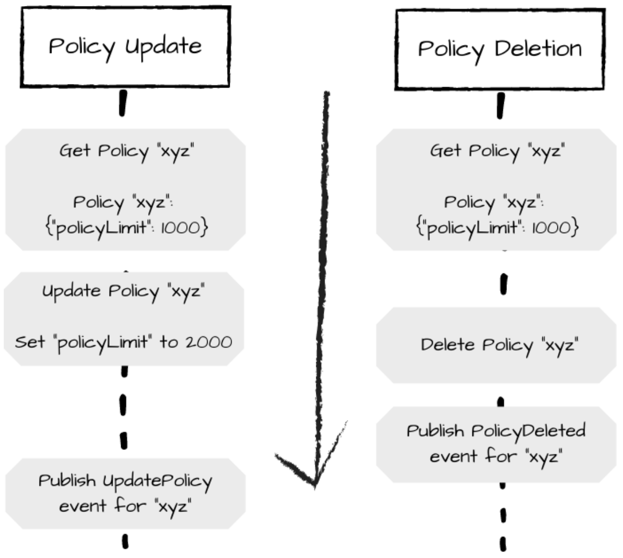

Policy Update
Get Policy "xyz"
Policy "xyz": {"policyLimit": 1000}
Update Policy "xyz"
Set "policyLimit" to 2000
Publish UpdatePolicy event for "xyz"
Policy Deletion
Get Policy "xyz"
Policy "xyz": {"policyLimit": 1000}
Delete Policy "xyz"
Publish PolicyDeleted event for "xyz"

Figure 5: An example scenario of policy update and deletion requests performed concurrently

interleavings of the events. Let's consider two simultaneous requests to the same policy. One of the requests changes the value of one of its fields while the other deletes the whole policy. After each of the changes is saved, the corresponding event is published to the `policy-events` topic. The order that the events were published determines the consumption order. If the deletion is published first and only then the update it means that the Risk Management Server will delete the policy only to recreate it with the next event.

Another shortcoming lies in the fact that the Risk Management Server does not alarm when a potentially harmful operation is being executed. An example of that would be consuming the `DeletePolicyEvent` even though the Risk Management Server does not have any information about the given policy. It could indicate that either:

- The service did not receive a creation event for the policy and therefore did not save in the materialized view. Any risk assessment reports that could be potentially generated in that period contain incorrect data.
- The policy creation event was truly never generated. That may indicate potential issues in the event publisher (Policy Management Backend) or message broker.

4.1.2 *Insurance Quote Expiration*

The Policy Management Backend service communicates also with the Customer Self-Service in the process of transforming an insurance quote into a policy. The typical scenario consists of an insurance quote request submitted to the Policy Management. The insurance company can then offer a quote and send it back to the Customer Self-Service. If the customer accepts the provided quote the policy is created.

However, all the proposed quotes have an expiration date after which they are no longer valid. The Policy Management service runs a periodical job that retrieves all the insurance quotes which expiration date has passed and for each of these sends a `InsuranceQuoteExpired` event to the Customer Self-Service to indicate that the insurance quote can be no longer accepted by the customer. However, the expiration process may be interleaved with the customer accepting the provided quote.

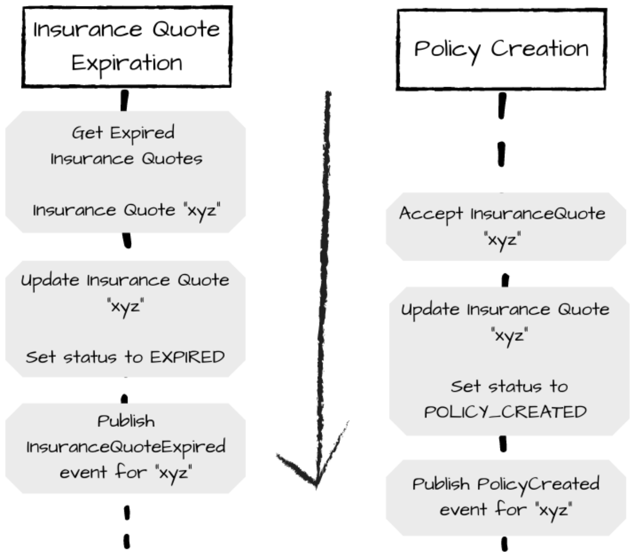

Insurance Quote Expiration
Get Expired Insurance Quotes
Insurance Quote "xyz"
Update Insurance Quote "xyz"
Set status to EXPIRED
Publish InsuranceQuoteExpired event for "xyz"
Policy Creation
Accept InsuranceQuote "xyz"
Update Insurance Quote "xyz"
Set status to POLICY_CREATED
Publish PolicyCreated event for "xyz"

Figure 6: An example scenario of two concurrent changes to a insurance quote

Before the insurance quote is persisted as expired the policy creation process is initiated. As a result both `InsuranceQuoteExpired` and then `PolicyCreated` events are send to the Customer Self-Service, see Figure 6. If the expiration event is processed first then the policy creation cannot be executed as no further status changes are accepted once the insurance quote was marked as expired. Therefore, the customer is not aware that the policy was actually created.

The correct behaviour of the system would be acknowledging that the creation process started before the expiration and thus discarding the expiration event. However, the periodic job that is responsible for expiring quotes is a separate process that does not communicate directly with the policy management coordinator. Such coordination could be introduced to the service but it would increase the complexity and not necessarily eliminate all the interleavings.

The described scenario can possibly have a critical impact on the global data consistency as the Policy Management service communicates also with the Risk Management, as described in Section 4.1.1. Therefore, if a policy was created and `PolicyCreated` event sent to the customer, it also means that the policy creation was propagated to the Risk Management server and the policy assigned to the customer affects their risk assessment.

4.1.3 *Non-transactional Message Publishing*

Using asynchronous message passing to communicate data changes requires ensuring the atomicity of the operation. The atomic action should either be completed successfully (both data change is persisted and the corresponding event was generated and published) or did nothing at all. Pritchett [33] argues that the message generation must be transactionally committed along with the data persistence,

```
Begin transaction
    insuranceQuoteRequestRepository.save(alteredInsuranceQuoteRequest);
    messageProducer.sendEvent(policyCreatedEvent);
End transaction
```

Figure 7: Pseudocode of changing an insurance quote that ensures consistency by using a transaction protocol

see Figure 7. However, this solution assumes that the message persistence is on the same resource as the database which is rarely seen in practice [20].

Practitioners seem to choose the system availability over the consistency and example of that can be also found in Lakeside Mutual. The Policy Management Backend when performing changes to the policy data does not use any transactional guarantees. What is more, it employs different sequential order of operations. The presented example of insurance quote expiration shows two different approaches to saving the data and generating the events.

```
1 // insurance quotes marked as expired
2 insuranceQuoteRequestRepository.saveAll(expiredQuoteRequests);
3 expiredQuoteRequests.forEach(expiredQuoteRequest -> {
4         InsuranceQuoteExpiredEvent event =
5             new InsuranceQuoteExpiredEvent(date, expiredQuoteRequest.getId());
6         customerSelfServiceMessageProducer.sendInsuranceQuoteExpiredEvent(event);
7 });
```

Figure 8: Order of operations in marking an insurance quote as expired

```
insuranceQuoteRequest.finalizeQuote(policyId, date);
PolicyCreatedEvent policyCreatedEvent =
    new PolicyCreatedEvent(date, insuranceQuoteRequest.getId(), policyId);
customerSelfServiceMessageProducer.sendPolicyCreatedEvent(policyCreatedEvent);
insuranceQuoteRequestRepository.save(insuranceQuoteRequest);
```

Figure 9: Order of operations in creating a policy based on its insurance quote

The insurance expiration periodic job retrieves all insurance quotes that should be expired, changes their state and performs a batch insert to the underlying database, see line 2 in Figure 8. Only after this step, the event generation is executed. The events are created sequentially, one by one. This scenario contains multiple points of failure as the execution could be interrupted between the insert and events publishing (lines 2 and 3) as well as during the event loop. That could result in some of the events being published while the rest is not.

Finalizing an insurance quote employs the opposite order as the corresponding event is being published before the status change is persisted in the database. Interruption in this scenario would result in a customer being able to see the created policy while the Policy Management service, which is the owner of the policy data, does not have any information about it.

```
public void sendEvent(Event event) {
    try {
        messageBroker.send(messageQueue, event);
        logger.info("Successfully sent an event.");
    } catch(Exception exception) {
        logger.error("Failed to send an event.", exception);
    }
}
```

Figure 10: Outline of message publication methods in Policy Management producers

The Policy Management message producers do not introduce any additional failure handling when the message publication has failed, for instance, due to the message broker not being available. All of the producer methods have a structure similar to the one presented in Figure 10. The publication method does not return any result and any potential exceptions are simply logged. We have not found any additional mechanisms in the Lakeside Mutual that would react to these errors and therefore these exceptions would have to be carefully tracked in the system monitoring to remediate the arisen consistency conflicts.

4.2 ESHOP ON CONTAINERS

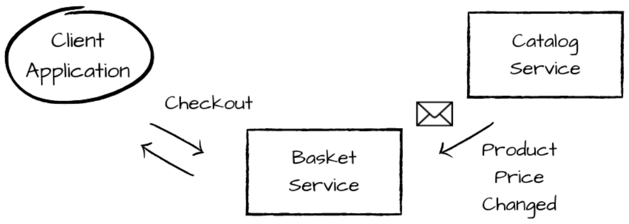


Figure 11: Communication with the Basket microservice in the eShopOnContainters application

Negative effects of lack of clear application invariants can be also observed in another open-source repository of a microservice application, eShopOnContainters. eShopOnContainters is an e-commerce system with a few well-defined domains such as orders, baskets, and products catalog. Functionalities of each individual domain are covered by distinct microservices which communicate with each other through an event bus.

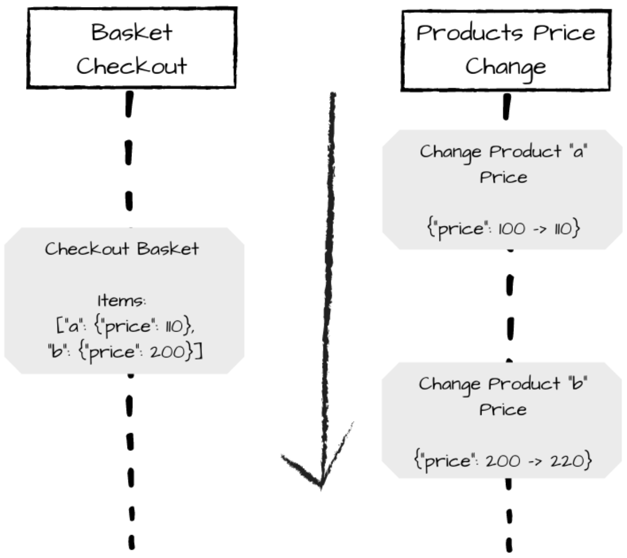


Figure 12: An example of interleaving between basket checkout and product price changes

Microservices in the eShopOnContainters application use asynchronous communication for the propagation of data updates. An example of such communication is presented in Figure 11. The Catalog microservice is responsible for maintaining and manipulating product data. It allows for creating and retrieving products but also for making changes to the existing data, for instance changing products' prices. Each price update is followed by an integration event sent to the event bus that reflects the change. One of the recipients of those messages is the Basket service that coordinates users' shopping baskets. For each received event that informs about the product's price change, all baskets that contain that product are updated with the new price. Additionally, the Basket service provides an API that allows users to finalize their orders through a checkout method.

However, the service does not employ any coordination patterns to synchronize the checkout requests with received asynchronous messages. This may result in unpredicted interleavings of changes and thus data anomalies. Figure 12 visualizes one of such scenarios. Considering the e-commerce platform has been operating for some time the administrators decide to increase the price of all the existing products by 10%. The changes are executed by the Catalog service that for each individual product update publishes a corresponding event. As the events are processed sequentially in the Basket service, it may occur that a basket is checked out while some of the price changes are not yet applied. Therefore, the prices of items in the basket are difficult to determine as they can have both the old and the new values.

With the usage of explicit application invariants, the order of the actions could be explicitly set by application developers. Without such constraints, a basket may be finalized when some of the items it contains have the new prices and the other the old ones, even though that all of the price changes happened before the checkout.

5 STREAM CONSTRAINTS

The examples of Lakeside Mutual and eShopOnContainers show how complex interaction patterns can be found in microservices and what the typical assumptions are. Although the microservices already implement a substantial amount of data management logic at the application level, the existing techniques do not take into consideration all possible event schedules and interleavings of events. This results in erroneous behaviours that have to be compensated manually [20] or may lead to uncovered inconsistencies that affect the end-users and general data integrity.

To encourage developers to reason about data correctness in the microservice applications and allow them to explicitly define the relevant dependencies of events we develop *Stream Constraints*, a system-level support that allows developers to define and enforce event-based constraints.

In this Chapter, we present an overview of the *Stream Constraints*. We describe the categorization of supported constraints, present the design of the enforcement system, and explain implementation details.

5.1 STREAM CONSTRAINTS OVERVIEW

Stream Constraints is a stream processing solution that transforms a stream of events in order to make it compliant with the defined invariants.

5.1.1 *Constraints Enforcement in Messaging Pipeline*

The stream modifications can be applied at three different points of a message pipeline:

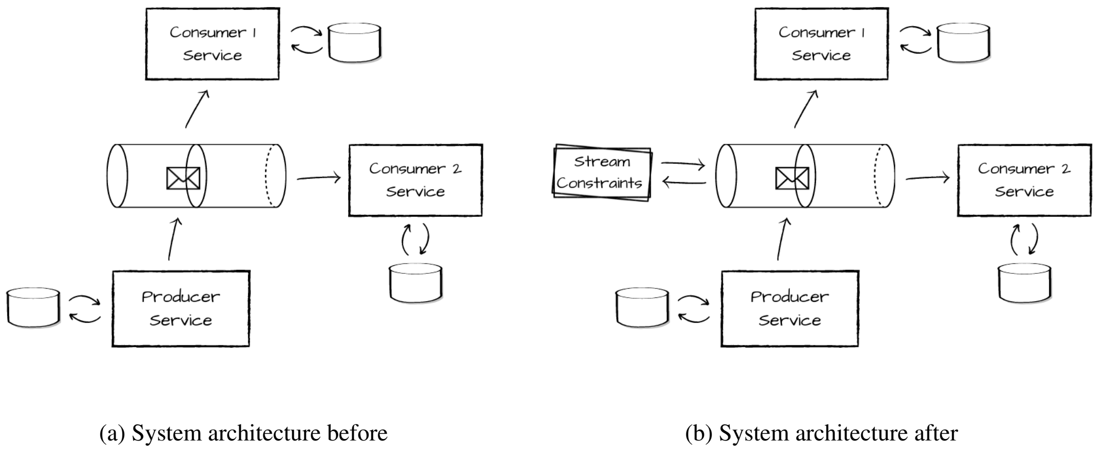
- At the producer. The constraint enforcement takes place directly at the producer, before writing events to a message broker or communication channel.
- Between the producer and the consumer. The modifications are directly connected to neither producer nor consumer but the constraint component is a middleware that operates on a message broker or channel.
- At the consumer. The modifications are applied on the stream consumed from the communication channel directly before calling methods of the consumer.

Applying the modifications at the producer appears to be the least efficient approach due to a number of reasons:

- Enforcing the constraints directly at the source is a very pessimistic approach. It would require complex coordination logic and concurrency control that could negatively impact the availability of the system. As seen in the Lakeside Mutual example, the events are usually produced in a process of manipulating the business entities. These are often initiated by a user request or external request and the time to respond is of great importance.
- Data produced by a single publisher may be consumed by multiple services and each of them has individual application constraints. Enforcing all of them at once at the producer could

significantly increase latency and be particularly challenging in a situation where services have mutually exclusive invariants.

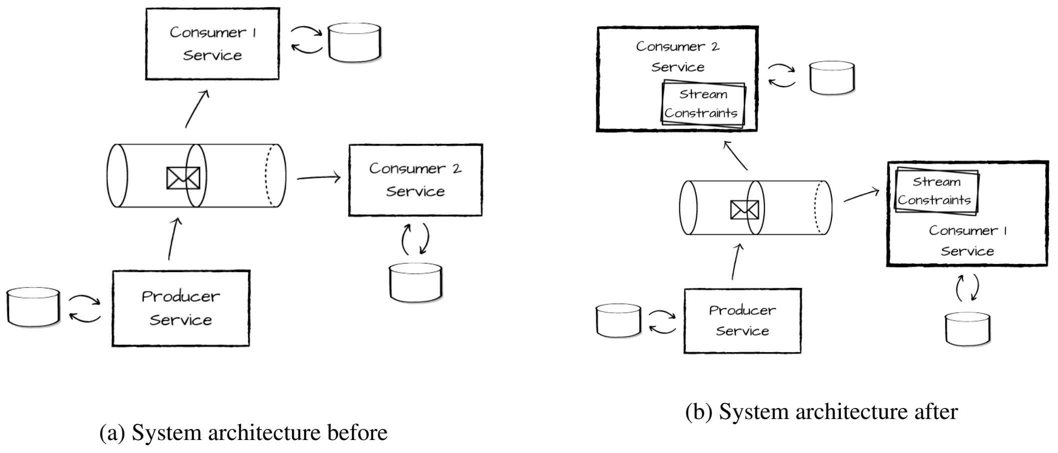
- Another challenge that is very difficult to overcome is the fact that the events may be generated by multiple producers. The modifications could be enforced separately at every one of them but the coordination between parties most likely would still be needed.



(a) System architecture before (b) System architecture after

Figure 13: Application of *Streams Constraints* in isolation

The *Stream Constraints* supports the second and the third approach: enforcing the constraints between the consumer and producer and at the consumer. Both of them have their applications in the microservice architecture. The second scenario is particularly useful when multiple consumers share the same invariants. The constraints can be then enforced only once and the altered stream is consumed by all of the services without the need to repeat the process for each individual consumer, see Figure 13. This approach can be also applied if the consumer technology is not directly compatible with the *Stream Constraints*, see an example of this in Lakeside Mutual in Chapter 6. The third solution brings data manipulation to the consumer as this is the main component that the invariants are defined for. In this approach, the altered stream is not being saved back to the message broker but the data records are passed directly to the recipient interfaces, see Figure 14.



(a) System architecture before (b) System architecture after

Figure 14: Application of *Streams Constraints* directly at a consumer service

5.2 EVENT-BASED CONSTRAINTS

Based on the analysis of the open-source repositories, we categorized the constraints to reflect the most common use cases. The identified categories of constraints act as a guideline for developers implementing event-driven microservices to reason about potential threats to data integrity.

5.2.1 *Causal Constraints*

A large part of the events produced in microservice architectures concern changes in the state of data. This category builds upon the *happens-before* relation [22, 7, 2] where some of the events precede others. An example could be an e-commerce application like the eShopOnContainers presented in Section 4.2, that allows for creating a shopping order, then it changes its status to paid, shipped and a final, delivered state. Produced events, therefore, correspond to a specific point in the *happens-before* chain.

The causal constraints specify what are the necessary prerequisites that need to be processed first. On the example of Lakeside Mutual, any update to a given policy can be processed only after the policy was created in a materialized view so after the creation event was generated and processed.

5.2.2 *Terminal Constraints*

When referring to the chain of events, in many cases there is one terminal state that should remove the possibility of further changes to the entity. If an entity has been deleted and a corresponding event has been processed the consumer should not receive any further events with updates to that entity. As detected in the Lakeside Mutual, a policy update that follows the policy deletion will result in the data being simply overwritten and no malicious behaviour will be revealed.

5.2.3 *Window Constraints*

The concept of a window in stream processing has proven to be an effective instrument to perform stream computations as it allows the processors to correlate events that happen within a specific time period. Similarly, in the window constraints events are specified through the lens of their timestamps. The example of insurance quote expiration, described in Section 4.1.2, has shown that data inconsistencies can be caused by interleavings of events that are not related but were generated in concurrent actions.

Window constraints apply to events that occur within the window with a predefined duration. These constraints are rather action-oriented than invariant-oriented and rely on detecting a pair (or group) of events within the window and enforcing one of three possible actions:

- swap the order so that the latter event is published before the former,
- drop the former event and publish the latter,
- drop the latter event but publish the former.

5.3 STREAM CONSTRAINTS API

The *Stream Constraints* was implemented in Scala language, version 2.13 and on top of Kafka Streams 2.7 [18].

5.3.1 *Kafka Streams*

Kafka Streams is a client library for building streams applications and components that aim to process unbounded, continuous data. This framework technology choice was motivated by the fact that Kafka Streams is a dedicated solution for processing and analyzing data stored in Kafka. Apache Kafka is an open-source data platform built around the message broker functionality. It has been widely adopted in industry and is the messaging choice of the biggest data-intensive applications, such as Twitter, Uber, or Netflix [4]. We believe that employing the leading stream processing framework will decrease the learning curve for microservice practitioners who want to take advantage of explicit constraint specification and at the same time, we exploit the guarantees provided by the Kafka platform, such as fault-tolerance and efficient data management.

```
case class OrderEvent(key: Int, action: String) {}
---------------------------------------------------------------------------
val builder = new StreamsBuilder
builder
    .stream("orders")
    .filter((_key, value) => value.action.equals("CREATED"))
    .to("new-orders")
```

Figure 15: Example of Kafka Streams topology definition that operates on *OrderEvent* messages and consists of filtering transformation

The core logic of each Kafka Streams application is a processor topology. The topology is a graph of stream processors: from source processors that directly consume messages, through stream operators to sink nodes. Kafka Streams provides a high-level DSL (Domain Specific Language) that supports the most common stream transformers such as mapping, aggregation, filtering or performing window computations. Figure 15 shows a (simplified) example of a very basic topology that consumes the *orders* topic which contains events regarding creation of the orders and their updates. The transformation includes filtering those events of type `CREATED` and publishes them to a new topic, *new-orders*.

However, although Kafka Streams does provide various stateless and stateful transformations, there is no such abstraction in Kafka Streams that would allow developers to specify constraints or invariants that have to be maintained over a stream.

Fortunately, aside from the DSL, Kafka Streams offers Processor API, a low-level interface that can be employed to define custom logic and connection between arbitrary stream processors. A stream processor is a node in the topology that represents a single processing step. It processes one received record at a time and propagates the computed result to downstream operators. *Stream Constraints* utilizes the Processor API and builds an interface on top of it that enables constraint enforcement.

5.3.2 *Stream Constraints Topology*

Kafka Streams DSL employs a declarative, functional programming style for defining a streams topology. To comply with that approach and enable our abstraction to be seamlessly integrated with the existing functions we introduce an additional *constrain* method to the topology builder:

```
def constrain(constraint: Constraint[K, V, L]): ConstrainedKStream[K, V, L]
```

It accepts a definition of constraints that need to be enforced or maintained, described in the following sections, and operates on the *ConstrainedKStream*. It is a wrapper type for the *KStream*, a Kafka Streams abstraction of a record stream, where each record represents a self-contained piece of data in the continuous stream [18].

Figure 16 illustrates how the new abstraction can be integrated with existing Kafka Streams DSL operators. The first two scenarios show topologies that only enforce the defined constraints and either

```
val builder = new CStreamsBuilder

builder
      .stream("orders")
      .constrain(constraints)
      .to("altered-orders")
-------------------------------------------------------------------------------
builder
      .stream("orders")
      .constrain(constraints)
      .foreach((_key, value) => processEvent(value))
-------------------------------------------------------------------------------
builder
      .stream("orders")
      .map((_key, value) => new NewOrder(value.key))
      .constrain(constraints)
      .to("altered-new-orders")
```

Figure 16: Examples of a use of the *constrain* method in Kafka Streams topologies

publish the altered stream to a different topic in the Kafka broker or directly call further event handling methods. The third example shows the possibility to integrate the *constrain* method into more complex transformations that use existing DSL processors, for example the *map* functionality.

5.3.3 *Constraint Specification*

The *constrain* method takes one parameter, the constraint. The constraints are defined through a triplet of types: `[K, V, L]`. In general, the Kafka platform operates on events (also called records or messages) in a form of key-value pairs. Therefore, `K` represents the type of key of the event. In the most typical use cases, it is either a primitive data type or String, but it can also have its own inner structure. `V` corresponds to the type of the value of the message. The values often employ a specific data format (e.g., JSON or Protobuf) and follow a predefined schema that describes what fields and types are expected to be present in the payload.

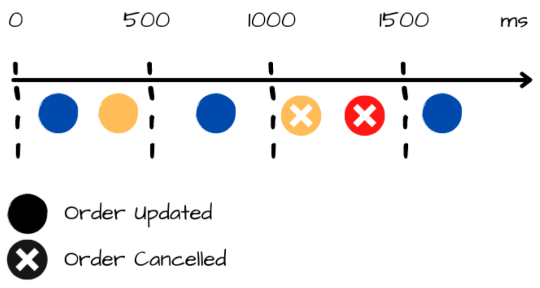


Figure 17: Visualization of different link values in a stream of events

Finally, `L` represents the type of a link that correlates events under constraints. The linking function extracts a link value from the events that are being processed. We decided to use the term *link* to avoid confusion with the message key but the link can be also perceived as a foreign or referencing key. On the example of e-commerce platform eShopOnContainers described in Section 4.2, the linking function extracts a unique identifier of a shopping order from the processed events. Different colors of events in Figure 17 represent different values of these order identifiers.

The example in Figure 18 shows a general constraint definition for messages with String as a key and OrderEvent objects as values. The link function uses the `key` field from the message payload so that all the messages with the same value in that field will be correlated.

```
new ConstraintBuilder[String, OrderEvent, Integer]
    // ...
    .link((_key, value) => value.key)(Serdes.Integer)
    .build(Serdes.String, orderEventSerde)
```

Figure 18: Constraint definition for `[String, OrderEvent, Integer]`

Additionally, Kafka Streams requires developers to specify the SerDes (Serializer/Deserializer) objects which describe how to materialize the processed data, when necessary. It is possible to use one of the predefined SerDes interfaces for primitive and basic types or provide a custom implementation as the `orderEventSerde`.

Each of the defined constraints must employ one of the three described types: prerequisite (causal), terminal or window-based. The *Stream Constraints* gives the possibility to define a single constraint or multiple of them as long as they satisfy the minimum correctness requirements, see Section 5.3.5. The interface is designed on the basis of lambda expressions where it is the developer's responsibility to define what events should be included in the constraints. The developer provides an anonymous function that is then applied to the processed records to determine whether the record is subjected to a given constraint.

```
1 val constraints = new ConstraintBuilder[String, OrderEvent, Integer]
2     .prerequisite(((_key, value) => value.action == "CREATED", "order-created"),
3         ((_key, value) => value.action == "UPDATED", "order-updated"))
4     .prerequisite(((_, value) => value.action == "CREATED", "order-created"),
5         ((_key, value) => value.action == "DELETED", "order-deleted"))
6     .terminal(((_key, value) => value.action == "DELETED", "order-deleted"))
7     .redirect("orders-dropped")
8     .link((_key, value) => value.key)(Serdes.Integer)
9     .build(Serdes.String, orderEventSerde)
```

Figure 19: Full constraint definition for `[String, OrderEvent, Integer]` with two prerequisites invariants and one terminal constraint

Figure 19 shows an example of a composite that consists of two prerequisite and one terminal constraints. All of them are considered equivalent, connected with logical *and* and the order is inconsequential. Apart from the lambda expression, it is required to provide the String identifier that is later on used in the constraints visualization, for instance *order-created* that is equivalent to:

```
(_key, value) => value.action == "CREATED"
```

It is crucial that the identifiers are correctly assigned so that the same functions have the same identifiers, and vice-versa, as there is no efficient way to determine equality between two anonymous functions.

Prerequisite constraints take two event specifications where the first event from the tuple is considered a required precondition for processing the second. The terminal constraints accept a single specification in the same form. Additionally, the constraints API offers the possibility to define additional redirect Kafka topic (line 7 in Figure 19) for all the messages that were discarded in the process of enforcing the constraints, for instance all the messages that were received after the terminal event for a given entity. If the redirect topic is specified the data records will not be silently dropped but published to the provided topic.

The definition of windows constraints is more complex as it depends on a higher number of parameters. The first two parameters (*before* and *after*) of the constraint builder follow the same convention as in the two other constraint types. Additionally, the developers are asked to specify the window duration and finally the action that will be applied to the events if they are detected within the specified period of time. The window constraints draw from a concept of session windows, described

```
val windowConstraint = new WindowConstraintBuilder[String, OrderEvent]
        .before((_key, value) => value.action == "CANCELLED", "order-cancelled")
        .after((_key, value) => value.action == "UPDATED", "order-updated")
        .window(Duration.ofSeconds(1))
        .swap
```

Figure 20: Example of window constraint definition for `[String, OrderEvent]` that swaps the event order within the given time window

in Chapter 3, where events that fall within the inactivity gap are correlated. This category of constraint heavily relies on the knowledge and experience of practitioners who need to reason about the window size. There is a clear trade-off between applying a large window that will detect anomalies even when they are spread out over time and using a small window that may miss some inconsistencies but will not affect the message latency to any great extent.

The prerequisite and terminal constraints are constructed in a from of application invariants so that they should remain true at all times. Window constraints, in turn, take the shape of *event condition action* [10] where the events are specified in the *before* and *after* clauses, the condition concerns their timestamps and the action is expressed as one of: *swap*, *dropBefore*, *dropAfter*.

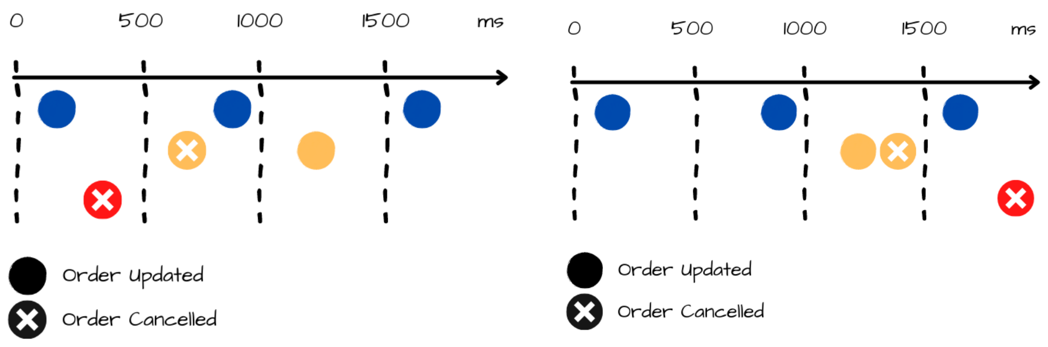


Figure 21: Application of the window constraint from Figure 20 to a stream of order events

The visualization of an events stream in Figure 21 shows the result of enforcing the window constraint shown in Figure 20. Different colors represent different values of *link* and therefore events of the same color concern the same entity. The window action is applied only to the yellow order as the given order of events (order cancellation before the update) was detected within the time window of 1 second. The events were modified in accordance with the specified action, so their order was swapped. The red entity event shows how the constraint affects a single cancellation event. The red event cannot be published immediately because of the possibility that it is followed by a late update. The cancellation is published only when the stream time advances by at least the window period and at the same time, no update events have been received for this order entity. The effect of possible duplicates of those event types is further discussed in Section 5.4.1. The blue shopping order updates are not affected by the constraint and there are no changes to them.

5.3.4 *Graph Representation*

To facilitate the process of defining complex constraints with multiple types and dependencies between each other, the *Stream Constraints* system compiles the definitions provided by the developers and presents a visual graph representation of the constraints. Graphs are built using the DOT language and Graph Visualization Software [14].

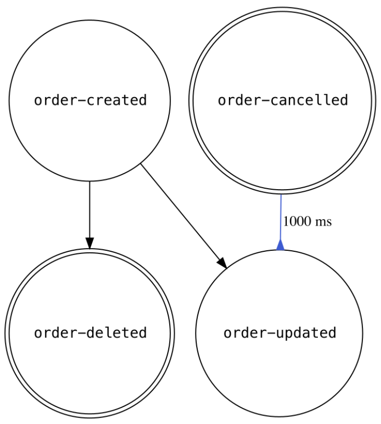


Figure 22: Graph representation of the order constraints generated by *Stream Constraints*

Nodes in the graph correspond to types of events that are extracted by using anonymous functions and use the provided identifiers. The terminal nodes are marked with a double circle. The nodes are connected by directed edges of one of two colors:

- black edges correspond to causal constraints where the head of an edge is a prerequisite for the tail of the edge, for instance *order-created* and *order-deleted* in Figure 22,
- blue edges concern window constraints and apart from the head and tail they also specify window duration (as a label, expressed in milliseconds) and the action. The shape of the arrow determines the type of action, as presented in Figure 23.

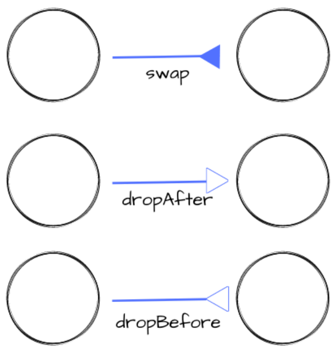


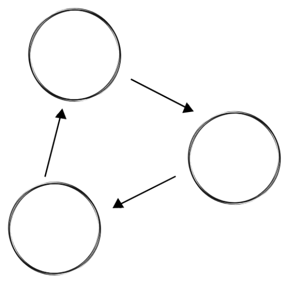
Figure 23: Window constraint edges in a graph visualization that correspond to different types of actions

5.3.5 *Constraint Correctness Validation*

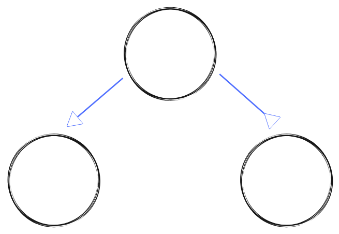
By not employing a strict constraint model *Stream Constraints* gives developers liberty and flexibility in defining constraints that cover a wide range of use cases. The solution is independent of the underlying event structure or format of the data. However, this may result in the constraints not being semantically correct, for instance, due to a mistake in the definition. To address that, the system performs basic validation of the constraints' correctness.

The verification is executed during the startup phase after the constraint graph has been built. If the specified constraints do not comply with the requirements, an error message is returned to the developer. Four major points are tested:

- there must be no cycles in the prerequisite dependencies as their presence would prevent any data records to be ever published, see example in Figure 24,



(a) Cycle in prerequisite constraints



(b) Mutually exclusive actions in two window constraints with the same head

Figure 24: Examples of constraints violations

- terminal nodes must not be prerequisites for any next events,
- there must be no mutually exclusive actions in window constraint that have common *before* event, for instance two actions: *dropBefore* and *dropAfter* that cannot be both applied if all events happen in the same time window, see example in Figure 24,
- there must be no mutually exclusive actions in window constraint that have common *after* event.

5.4 ENFORCEMENT IMPLEMENTATION

The *Stream Constraints* is designed on top of the Processor API of Kafka Streams that allows to define and connect custom processors and to interact with state stores [18]. The logic responsible for enforcing event-based constraints is implemented in two stateful transformers: Window Constraint Transformer that applies the window-based actions and State Constraint Transformer that enforces both prerequisite and terminal constraints. The processors are connected sequentially and both of them have additional output to the redirect topic (if one has been provided).

The transformers maintain the processing state using state stores. By default, Kafka Streams uses RocksDB as the underlying storage technology and it is also used in *Stream Constraints*. RocksDB is an embedded persistent database for key-value data [38]. The state is stored on the disk and continuously backed up to a Kafka broker in the form of a changelog topic to guarantee fault tolerance.

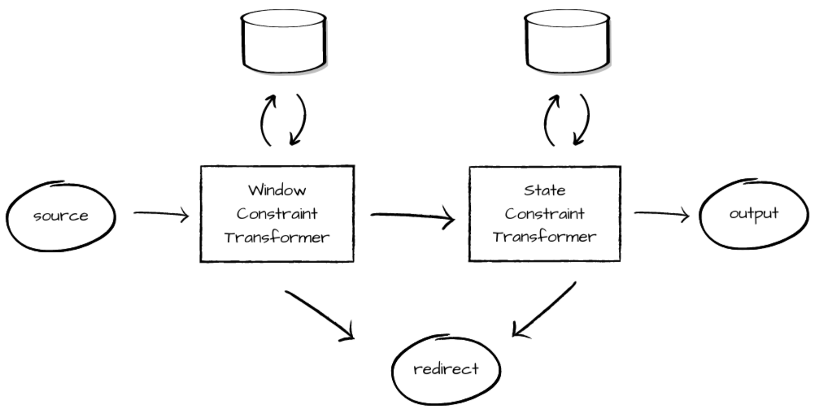


Figure 25: Overview of stream processors responsible for enforcing event-based constraints

Kafka Streams processors process one received record at a time. A single record may be either saved in the connected store for further processing, published to a redirect topic or flushed to a downstream processor, the sink in particular. Figure 25 presents an overview of this architecture.

Each of the transformers, as the very first step of computation, verifies if the processed record is liable to any of the defined constraints. If not, the record is passed further and published to the sink. If the record is subject to any of the constraints, and therefore cannot be published immediately, it is handled by a corresponding processor.

5.4.1 *Window Constraint Transformer*

Enforcing window constraints requires the use of a store where the *before* events can be temporarily saved in anticipation of potential *after* events. The transformer employs a *WindowStore* for that purpose, with link values as keys and timestamped key-value pairs as buffered values. However, as shows the red entity in Figure 21, it is possible that the *before* event is not followed by the *after* and therefore should be published. To achieve that the transformer schedules a punctuation function that periodically publishes outstanding records. For each of the defined window constraints, the punctuator retrieves records from buffer stores for which the window period has passed and no *after* events have been received.

Kafka Streams distinguishes two notions of time: the wall-clock time and the stream time [18]. The wall-clock time reflects the absolute value of time specified by the machine that stream processing is executed on. The stream time is essentially the maximum timestamp the processor has observed in the stream so far. The punctuation mechanism can operate on each of them and the Window Constraint Transformer employs a punctuator that operates on the stream time. In practice, this means that the punctuator will not be unnecessarily called when there are no new records but also that the stream time advancement has to be observed before the periodical check. For each of the window constraints defined, the interval between punctuations is equal to half of the window duration. It results in the *before* events being buffered for a duration no longer than one and a half of the window duration. The punctuation method could be scheduled more frequently, in particular after each record received, but that would significantly increase a time to the processing time as the punctuation method has a relatively high cost resulting from querying the buffer stores. The notion of time and order in *Stream Constraints* is further discussed in Chapter 7.

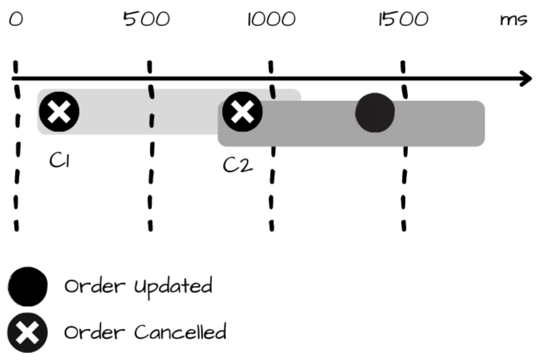


Figure 26: Visualization of windows in a scenario where duplicates of *before* events occur

One of the important aspects that need to be considered in the design of window constraints is the possibility of event duplicates. It may occur that multiple *before* or *after* events are received within the same window. The term *duplicate* is used for those events but they are not necessarily the exact copies of each other. Each of the constraints defines *before* and *after* events but it is not clearly stated whether only the *first* events processed should be considered or *all* events that satisfy the conditions. *Stream Constraints* adopted a following approach. Duplicates of the *before* events are considered in the constraint action but they do not extend the size of the window. Figure 26 shows an example of this scenario where only the event *C2* is liable to the window constraint as the time difference between *C1* and *after* event is greater than the specified window duration. Decision on the duplicates of *after* events is left to the developers as there is a trade-off behind them. *Stream Constraints* supports both eager and lazy options. In the eager approach, the constraint is enforced on the first occurrence of *after* event, and after that the constraint is considered completed, see example in Figure 27. This approach results in reducing the latency and may be applied to a system where duplicates are not expected. However, when duplicates are possible and should be considered, the developers may choose the lazy approach and wait for the full-window duration before applying the action, see Figure 28.

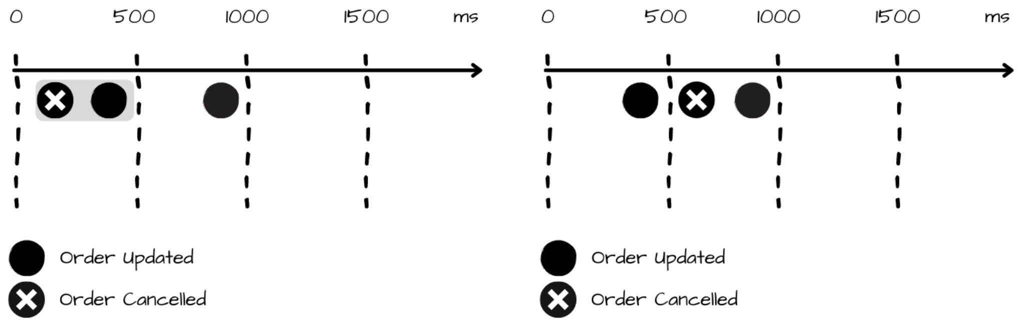


Figure 27: Enforcing window constraint with the eager approach

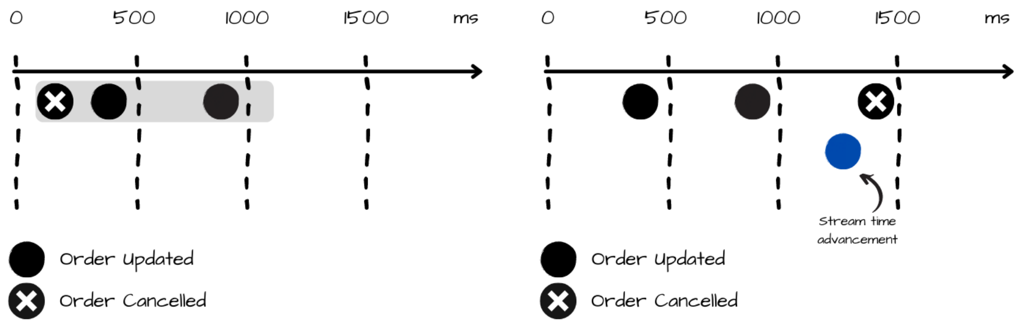


Figure 28: Enforcing window constraint with the lazy approach

The *Stream Constraints* supports combining multiple window constraints, also in the form of a dependency chain, see Figure 29. In these scenarios, all the nodes that have further dependencies, such as the *order-updated*, cannot simply employ the eager approach as they may be also a subject of the next constraints. In that situation, the action of the first detected constraint is saved for later and the stream is checked for the next constraints from the chain. In the worst case, when all constraints enforce the *swap* action and all of them have to be applied, the first event may be delayed by the sum of windows in the underlying chain.

5.4.2 *State Constraint Transformer*

The State Constraints Transformer is responsible for enforcing prerequisite and terminal constraints which are related to changes in the state of a given entity. This processor performs stateful computa-

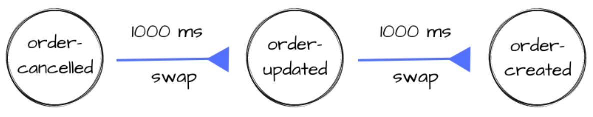


Figure 29: Chain of window constraints with a common node in *order-updated*

tions and therefore interacts with state stores. All of the state stores use a *KeyValueStore* interface and are divided into three types:

- Graph store. The store contains information about the current state of entities that have been observed so far. Keys contain link values and values represent the current state of a given entity. The values use a graph template built from the state constraints definition and apply additional information when processing data records. There are additional labels on each of the graph nodes, e.g. whether a given set of records have been processed and published or whether data records were buffered due to missing prerequisites. Figure 30 shows example records from a graph store.
- Terminated store. If there is a terminal constraint defined, the terminated store tracks all the link values for which a terminal event has been processed.
- Buffer store. Buffer stores save all the events that cannot be published because of missing prerequisites.

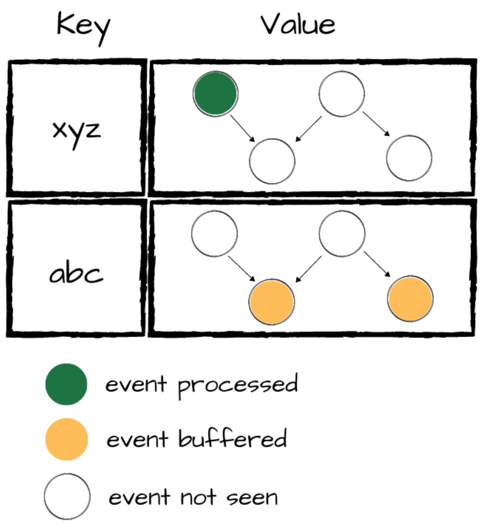


Figure 30: Example records in the graph state store of the State Constraints Transformer

The enforcement process consists of a series of interactions with the stores to verify the state invariants. If the entity from a record has not been seen yet, then a new row is inserted into the graph store. If the record can be directly published, a note is made on the corresponding node that this record has been seen. Otherwise, the record is saved in buffers and an adequate annotation is put in its graph that it could not be processed due to missing prerequisites. When the State Constraints Transformer receives a record that has been marked as a prerequisite for some other events, it checks whether this occurrence unlocks previously buffered data. The exact steps are presented in Algorithm 1.

Algorithm 1: State Constraint Transformer

```
    Input: (key, value)
    Constraint Definition: constraint
    Stores: graphStore, terminatedStore, bufferStores
 1  link ← constraint.link((key, value));
 2  if constraint is not applicable to (key, value) then
 3      forward((key, value));
 4  if terminatedStore contains link then
 5      forward((key, value), redirect = true);
 6  graph ← graphStore.get(link) || constraint.graph;
 7  node ← graph.find((key, value));
 8  prerequisites ← node.predecessors;
 9  if prerequisites are empty OR $\forall_{event \in prerequisites}$ event.seen == true then
10      forward((key, value));
11      node.seen = true;
12      successors ← node.successors;
13      forall node ∈ successors do
14          if node.buffered == true AND $\forall_{event \in node.prerequisites}$ event.seen == true then
15              buffered ← buffers.get(node.name).get(link);
16              forward(buffered);
17              node.seen = true;
18              buffers.get(node.name).delete(link);
19  else
20      buffers.get(node.name).put(link, (key, value));
21      node.buffered = true;
```

The *KeyValueStores* by default do not have a retention period and will retain data forever until explicitly deleted. A record can be possibly deleted from the graph store when the entity receives a terminal event and therefore its link value is inserted into the terminated store. Then the graph data is no longer needed and is being removed. This takes place in the *forward* method after determining whether the published event is a terminal, see lines 10 and 16. Data from a buffer store is removed after required prerequisites are received and the buffered records are published, see line 18.

6

EVALUATION

The analysis of possible event interleavings in the Lakeside Mutual applications, presented in Chapter 4, shows that lack of clearly defined event-based constraints and lack of enforcement of those raises a risk to data integrity. To evaluate the proposed solution, *Stream Constraints* is applied to the Lakeside Mutual architecture, and by enforcing the event-based constraints attempts to remediate the unsafe application behaviour. This chapter describes the conducted experiments and presents obtained results.

The evaluation experiments intend to answer the following questions:

- What is the effect of enforcing event-based constraints on data correctness compared to no invariants and a baseline solution?
- How do the identified constraint categories handle invariants of real-world applications?
- What is the performance, in terms of data latency, of *Streams Constraints* compared to other solutions?

6.1 STREAM CONSTRAINTS IN LAKESIDE MUTUAL

The evaluation is performed on two scenarios: (i) policy updates and deletions, described in Section 4.1.1, (ii) insurance quote expiration, described in Section 4.1.2. There are three microservices involved in the process: the Policy Management as a data producer in both of the cases, the Risk Management as a consumer in (i) and the Customer Self-Service as a consumer in (ii). Figure 31 presents the use of *Stream Constraints* on top of the Kafka-based communication among the microservices.

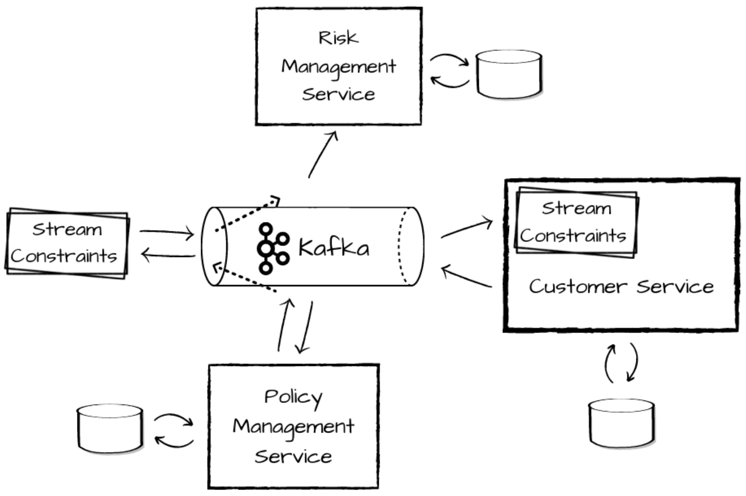


Figure 31: Application of *Stream Constraints* in the Lakeside Mutual application

	Number of Events	
Type of Event	**Min**	**Max**
Policy Update	0	5
Policy Deletion	0	2

Table 2: Characteristics of policy events generated for each of the policies

Both of these scenarios were included in the experiments because of their individual characteristics. The first case involves Risk Management, a Node.js application that is not directly compatible with Kafka Streams and therefore requires the stream with enforced constraints to be saved back to the message broker. This use case gives also the possibility to illustrate the causal and terminal constraints on the example of policy entities. The second scenario uses the *Stream Constraints* as an integrated component of the written in Java Customer microservice. It is also a good example of application window-based constraints to events that are not causally related. Additionally, it involves combining two separate Kafka topics, in contrast to a single topic in the policy events scenario.

6.2 POLICY UPDATES AND DELETIONS

The first experiment is performed on three different configurations:

1. **Default**: the Risk Management does not maintain any invariants and consumes messages without restrictions,
2. **Baseline**: a baseline solution implemented directly in the Risk Management that consumes messages and periodically applies order and terminal constraints on batches of messages,
3. **Stream Constraints**: applying *Stream Constraints* and enforcing explicitly defined event-based constraints on the stream that is then consumed by the Risk Management.

6.2.1 *Data Generation*

The data records were generated artificially, directly to the `policy-events` Kafka topic in order to create an isolated environment with high intensity of event interleavings. The generation was performed according to the following procedure. For a fixed number of policies ($n = 1000$) the events were randomly generated according to the possible minimum and maximum number shown in Table 2. The numbers were selected in a way to simulate various stream anomalies, for instance policy deletion without former creation or duplicates in deletions. As a result, 2513 events were generated for 905 different policies. Some of the policies were assigned 0 events so the resulted number is lower than the initial number of policies equal to 1000. The events were then randomly shuffled to obtain interleavings of updates and deletions.

In each of the experiments, the events are sequentially published with a 1 ms interval and each individual message has a publication timestamp assigned. At the end, after 1000 ms delay, the data generator publishes one additional message that plays an important role in advancing stream time that results in closing any remaining open stream windows.

6.2.2 *Baseline Solution*

For the purpose of evaluation, an additional baseline solution was developed that also enforces message order constraints, but only through batch processing. The Risk Management microservice in the baseline version contains an additional application logic that stores the incoming events in an in-memory queue, periodically dequeues a batch of events, and then applies order constraints. Each of the batches is grouped by the policy identifier and has the following order applied:

```
const order = [ 'UpdatePolicyEvent', 'DeletePolicyEvent' ]
```

In the next step, any duplicates of the delete events are discarded. The altered events are then sequentially handled by the data manager, see Algorithm 2.

Algorithm 2: Baseline solution for applying constraints

```
Input: queue
Order of events: order
Event to deduplicate: deduplicate
1 grouped ← queue.groupBy(message → message.key);
2 enforced ← [];
3 forall events ∈ grouped do
4     sorted ← events.sortBy(order);
5     deduplicated ← sorted.unique(deduplicate);
6     enforced.push(deduplicated);
7 foreach message ∈ enforced do
8     handle(message)
```

6.2.3 *Stream Constraints*

The *Stream Constraints* solution is a simple Kafka Streams application that consumes messages from the `policy-events` topic, applies the event-based constraints, and publishes the resulted stream back to the message broker, see last lines in Figure 32. The Risk Management service then consumes the `policy-events-constrained` topic.

```
val windowConstraint = new WindowConstraintBuilder[String, PolicyDomainEvent]
    .before(((_, e) => e.type == "DeletePolicyEvent", "policy-deleted"))
    .after(((_, e) => e.type == "UpdatePolicyEvent", "policy-updated"))
    .window(Duration.ofSeconds(1))
    .swap

val constraints = new ConstraintBuilder[String, PolicyDomainEvent, String]
    .prerequisite(((_, e) => e.type == "UpdatePolicyEvent", "policy-updated"),
      ((_, e) => e.`type` == "DeletePolicyEvent", "policy-deleted"))
    .windowConstraint(windowConstraint)
    .terminal(((_, e) => e.type == "DeletePolicyEvent", "policy-deleted"))
    .redirect("policy-events-redirect")
    .link((_, e) => e.policyId)(Serdes.String)
    .build(Serdes.String, policyEventSerde)

val builder = new CStreamsBuilder()

builder
    .stream("policy-events")(Consumed.with(Serdes.String, policyEventSerde))
    .constrain(constraints)
    .to("policy-events-constrained")(Produced.with(Serdes.String, policyEventSerde))
```

Figure 32: Definition of the event-based constraints related to policy updates and deletions

The defined constraints are of three types. First of all, there is a causal constraint that requires at least one update event (equivalent to a creation of the policy) to be processed before a deletion event. Secondly, a terminal constraint is defined for the deletion events together with a redirect Kafka topic. Additionally, there is one window constraint that detects an inverse order of policy events, so

	Update After Delete	Delete With No Update	Duplicate Delete
Default	361	141	331
Baseline	323	141	253
Stream Constraints	0	0	0

Table 3: Number of policies that suffer from invariant violations per experiment configuration

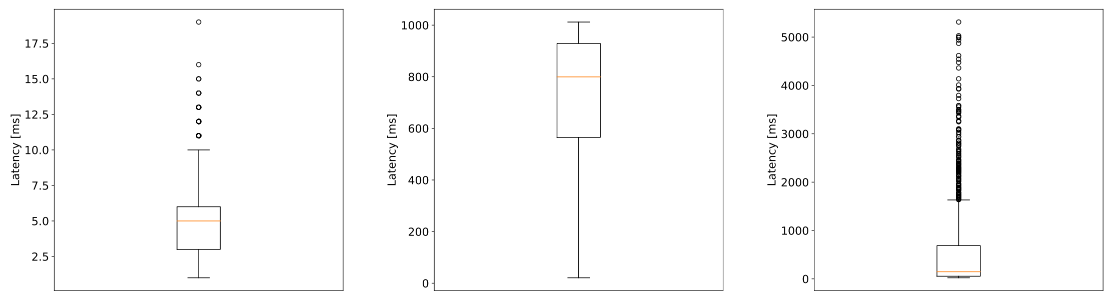
a deletion before an update, within a 1 second time window and applies the *swap* action. The exact definitions of the constraints are provided in Figure 32.

6.2.4 *Results*

The analysis from Section 4.1.1 reveals that the Risk Management is subject to the following types of invariant violations: updates after deletions that result in the state overwrite, silent acceptance of a deletion with no prior update (considering that the first occurrence of an update event is treated as a creation event), and duplicates in deletions. Occurrences of those anomalies during experiments were detected and counted. The exact numbers of policies that encountered the violations in each of the experiment configurations are presented in Table 3.

In the Default scenario, a total of 554 out of 906 policies were exposed to at least one of the invariant violations. The Baseline solution managed to decrease this number to 513 by changing the order of events and removing delete duplicates in each of the batches. There is no difference in the number of Delete With No Update violations as the Baseline is a purely stateless solution that does not store information about processed records nor interacts with the internal database and therefore is not able to determine the state of a given policy.

Applying the *Stream Constraints* on the policy events stream managed to eliminate all the violations. However, the high concentration of anomalies in the stream resulted in 551 events being published to the redirect topic. These events occurred after the terminal event and fell outside the 1-second window that could potentially reverse the order. Additionally, 212 events from 141 policies, see Table 3, remained buffered in the state stores of *Stream Constraints* as a result of a missing prerequisite. The remaining 1751 events were published to the output topic and consumed by the Risk Management service.



(a) Default (b) Baseline (c) Stream Constraints

Figure 33: Latency of processed events in the three experiments configurations

The performance of each configuration is evaluated by measuring the events' latency. The latency is expressed as an end-to-end time delay between publishing the message (timestamp assigned directly before sending the message to message broker) and consuming it (timestamp assigned directly before handling the message in data manager). The results presented in Figure 33 show that both the Baseline and *Stream Constraints* solutions introduce a significant overhead compared to the Default scenario.

	Mean	**Median**	**Standard Deviation**
Baseline	722 ms	800 ms	251 ms
Stream Constraints	559 ms	147 ms	789 ms

Table 4: Statistics of events latency for the Baseline and Stream Constraints experiments

When no additional restrictions are applied the consumer is usually able to handle the events within less than 10 *ms*. Enforcing any type of constraints considerably increases this delay.

However, as presented in Table 4, the statistics of latency differ between the Baseline and *Stream Constraints*. The mean latency value is in favour of the *Stream Constraints* but the biggest difference is seen in the latency median. The significantly higher value in the Baseline solution is a result of the message buffering that involves all incoming records, regardless of their type, and applying the constraints on the buffered batch. The *Stream Constraints* solution is able to detect which messages are not liable to constraints and that contributes to lower latency in general. Another important difference is highlighted by the value of standard deviation which is significantly higher in the configuration that uses *Stream Constraints*. The latency values are there far more dispersed due to enforcing the prerequisite constraint. High values are a result of delete events being buffered within the processor until corresponding update events are received and the experiment data indicates that this can take up to a few seconds. In the Baseline solution, the latency varies only from a few milliseconds to slightly more than 1000 *ms* which is the interval of periodical dequeuing.

6.3 EFFECTS OF LAZY AND EAGER WINDOWS

Section 5.4.1 introduced two alternatives of the window constraints: eager that are enforced upon the first occurrence of the *after* event, and lazy that wait for the full window duration. In the next experiment, the impact of window type on the number of invariant violations and event latency was evaluated.

The experiment was conducted in the same environment of policy updates and deletions and the window duration was set to 1 second. Both of the configurations did not produce any errors in the output stream but there was a difference in the number of events that were published to the output. The full window managed to preserve 1779 events so 28 events more than than the eager window. These 28 events are policy updates that have been additionally included to open windows. In the configuration with an eager window, those events were published to the redirect topic as they arrived after the terminal event. The cost for that was an increased latency as each individual delete event was buffered for 1 second, see Figure 34 for comparison.

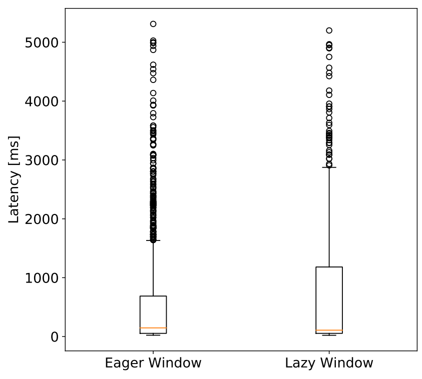


Figure 34: Latency of processed events for constraints with eager and lazy windows

6.4 INSURANCE QUOTE EXPIRATION

Next experiments were conducted on the insurance expiration scenario and simulated interleavings of `InsuranceQuoteExpired` and *PolicyCreated* produced by the Policy Management microservice. Each type of events is produced to a separate Kafka topic. The data generation followed a similar procedure as in previous experiments. For each of the insurance quotes ($n = 2000$) 0 or 1 event regarding policy creation was generated and 0 or 1 expiration event. That resulted in 2028 events that were followed by one additional record for stream time advancement. The events were shuffled to simulate various orders and delays.

The experiment involves two configurations:

1. **Baseline**: a solution implemented in the Customer Self-Service microservice that consumes data from two topics and puts the records in a common queue. The queue is periodically analyzed and cleared out. If in a given batch an expiration event has been consumed before a policy creation event for the same policy the expiration event is discarded.

2. **Stream Constraints**: applying *Stream Constraints* and enforcing a single window constraint that detects occurrence of insurance expiration and policy creation events within 1 second and applies the *dropBefore* action, as provided by the exact definition shown in Figure 35.

The Default configuration was not included in this experiment because of the different consumer strategy it employs that makes the comparison unreliable. In the Default scenario, the insurance expiration and policy creation are separate event streams with separate consumers while in the included configurations the dataflows are combined together and consumed by a single recipient.

```
val windowConstraint = new WindowConstraintBuilder[String, InsuranceQuoteEvent]
    .before(((_, e) => e.isInstanceOf[InsuranceQuoteExpiredEvent],
        "insurance-quote-expired"))
    .after(((_, e) => e.isInstanceOf[PolicyCreatedEvent],
        "policy-created"))
    .window(Duration.ofSeconds(1))
    .dropBefore

val constraint = new ConstraintBuilder[String, InsuranceQuoteEvent, Long]
    .windowConstraint(windowConstraint)
    .link((_, e) => e.getInsuranceQuoteRequestId)(Serdes.Long)
    .build(Serdes.String, insuranceQuoteEventSerde)

val builder = new CStreamsBuilder()

builder
    .stream(Set("insurance-quote-expired-events", "policy-created-events"))
        (Consumed.with(Serdes.String, insuranceQuoteEventSerde))
    .constrain(constraint)
    .foreach((_, event) => handleEvent(event))
```

Figure 35: Window constraint definition for the insurance quote expiration

The generated events created 1531 unique insurance quotes of which 240 were subject to constraint violation. There is only one type of constraint violation for this experiment scenario and has the form of an insurance expiration event followed by a policy creation. The Baseline solution managed to decrease the number of affected policies to 194. In the *Stream Constraints* configuration 119 invalid insurance quotes were observed. The constraint enforcement did not eliminate all anomalies as some of the policy creation events fell outside the defined window. The effect of different window sizes on the number of eliminated anomalies is further evaluated in Section 6.4.1.

Delays of events in the Baseline solution vary from 0 to 1000 *ms* with a distribution close to uniform. The nontypical latency distribution for the *Stream Constraints* highlights the main key points of the system and the experiment itself:

- More than 1000 events were processed with minimal delay. Those were the events related to policy creation as they are not liable to the defined constraint.
- There are no events with latency between the minimal value and 1000 *ms*. This is a result of dropping the detected expiration events upon receiving policy creation events.
- The rest of insurance expiration events have latency higher than 1000 *ms* so the defined window duration. For those no *after* event was detected.
- There are events with a significant delay close to 2000 *ms*. These are closely related to the design of the experiment. Stream processing solutions are designed to operate on an unbounded stream of data, however, for the purpose of our evaluation, the stream of events was truncated. To remediate this effect and simulate the continuous dataflow the last record was followed by an artificial event delayed by 1 second, responsible for time advancement and closing any remaining open windows. That resulted in some of the events being significantly longer in the *Stream Constraints* buffer, see the rightmost bars in Figure 36.

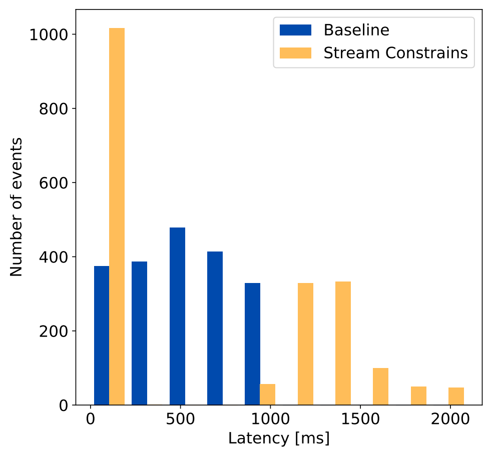


Figure 36: Histogram of events latency for the Baseline and *Stream Constraints* solutions

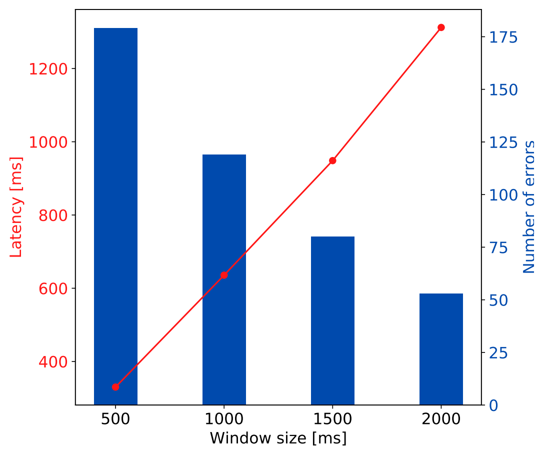


Figure 37: Mean latency and number of invariant violations for different sizes of window applied to *Stream Constraints*

6.4.1 *Effects of Window Sizes*

To examine the effect of window size on the number of violations and event latency we conduct an additional experiment with the *Stream Constraints*. The results are presented in Figure 37. As expected, the number of invariant violations decreases with increasing window size as more event pairs can be connected, but the overhead is manifested in the event latency. The mean values increase from less than 300 *ms* in 500 *ms* window to more than 1200 *ms* for the 2 seconds window.

7 DISCUSSION

In this Chapter we discuss the meaning of the evaluation results and implications and limitations of the *Stream Constraints*.

7.1 EVALUATION RESULTS

It is clear that any type of invariant enforcement on the event stream introduces additional overhead to the message-based communication between microservices. The cost is associated with the latency of the events that need to be processed within a context. Both the Baseline and the *Stream Constraints* solutions postpone the production of data records for consumers until after a certain state determination is made. However, only the *Stream Constraints* manages to eliminate or at least significantly decrease the number of invariant violations.

It should be pointed out that the conducted experiments assumed very pessimistic scenarios with an extremely high concentration of anomalies. In more typical, real-world use cases, the Baseline queuing solution would cause unnecessary delays to all of the events and would fail to detect errors in most cases. The *Stream Constraints* gives a guarantee that the causal constraints (prerequisite and terminal) are enforced and offers higher chances to eliminate anomalies in windows without influencing events that are not subject to constraints.

Furthermore, *Stream Constraints* is able to preserve a wide range of invariants for events that are causally related or happen to be interleaved because of concurrent operations. It allows the developers to move the implicitly encoded constraints from application logic to the system level and makes code reuse straightforward. Although *Stream Constraints* is specifically designed to operate in the Kafka environment, the concept is independent of underlying technology choices in microservices and therefore does not require designing and implementing consistency mechanisms from scratch for each new service, as is the case in the Baseline solution.

Nevertheless, the *Stream Constraints* depends greatly on the practitioners' knowledge and expertise, especially when it comes to defining window constraints. As illustrated by Figure 37, there is no single right choice for defining the constraints and some of the applications may prefer data integrity assurance at the cost of increased delay while others need to carefully balance the trade-off.

7.2 REDIRECT TOPIC

The experiments on policy updates and deletions described in Section 6.2 show how the events that occur after terminal events are redirected to the alternative topic. It should be noted that a relatively high number of those is a result of the artificially generated data and such intensity in a real-world application could indicate serious data integrity issues in the producer. Nevertheless, the *Stream Constraints* employs a concept of *dead letter queue* to give the developers a chance to investigate the unexpected events and respond to them if necessary. Although a dead letter queue usually requires manual intervention, exploiting it in the *Stream Constraints* gives more insight into the data correctness and allows the developers to spot potential threats to the data integrity. Publishing such records could otherwise overwrite the state of the consumer or a significant update could be silently lost.

7.3 RECORDS CORRELATION

The *Stream Constraints* enforces the event-based constraints on a single stream. However, the defined constraints can possibly concern events produced to multiple Kafka topics. The scenario involving insurance expiration and policy creation events, evaluated in Section 6.4, is an example of that. The former events are produced to the `insurance-quote-expired-events` topic and the latter to the `policy-created-event`. In order for the *Stream Constraints* to be able to enforce the constraints, those topics need to be merged together. From the perspective of the Kafka Streams application, this does not pose a problem as it is possible to declare multiple source topics that the records are consumed from, see line 17 in Figure 35. Although there is no ordering guarantee for records from different topics, all needed constraints will be enforced by subsequent *Stream Constraints* operators. Nevertheless, such scenarios influence the logic of the message consumers.

When enforcing a constraint *Stream Constraints* has the possibility to publish the records to different topics. However, the order of publishing will not determine the order of consuming and processing the messages in the separate consumers. The record that is published first may be processed at a later time than the record that was originally published later, for instance, because of a consumer lag. In order to maintain the enforced order, it is required that the altered stream is consumed by a single consumer that is able to detect and handle multiple event types.

Another challenge that is faced by the *Stream Constraints* is data partitioning which is a main concurrency mechanism in Kafka [4]. Usually, a Kafka topic is divided into 1 or more partitions that can be concurrently consumed by multiple instances of a consumer. Each individual partition is consumed by exactly one consumer. The data records are assigned to partitions based on their keys or by using a round-robin strategy if records have no keys. Attaching keys to messages will ensure records with the same key will be placed in the same partition in a topic. In practice, it is often desired that messages concerning the same entity, for instance a policy in the Lakeside Mutual, land in the same partition in order to avoid race conditions caused by multiple instances of the consumers.

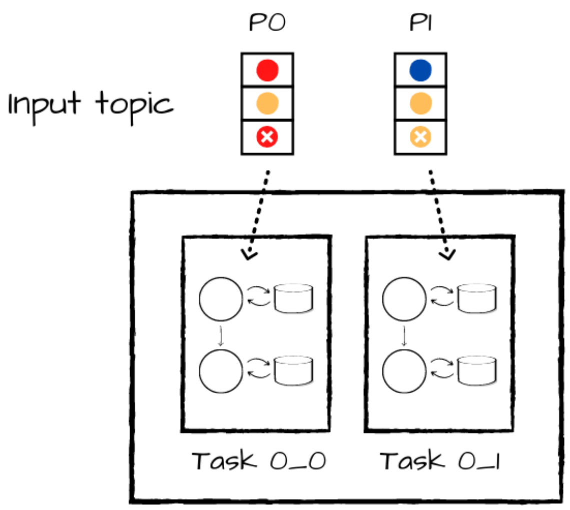


Figure 38: Diagram of a *Stream Constraints* application with two tasks that consume a two partitions topic with keyless records

However, if the records are randomly partitioned more consideration is needed for applying the *Stream Constraints*. Kafka Streams applications operate through a fixed number of stream tasks based on the input topic partitions, with each task being assigned one partition [18], see example in Figure 38. It is crucial for the *Stream Constraints* to have a full projection of the state, so all the events that were processed for a given entity. If the events are put randomly across all the partitions multiple tasks may have processed a subset of the events and therefore have only a partial projection of the state.

For instance, the *Task 0_0* in Figure 38 sees the yellow entity as active while the *Task 0_1* processed the terminal event and will discard any future records for that entity. Enforcing constraints in such environment is not possible.

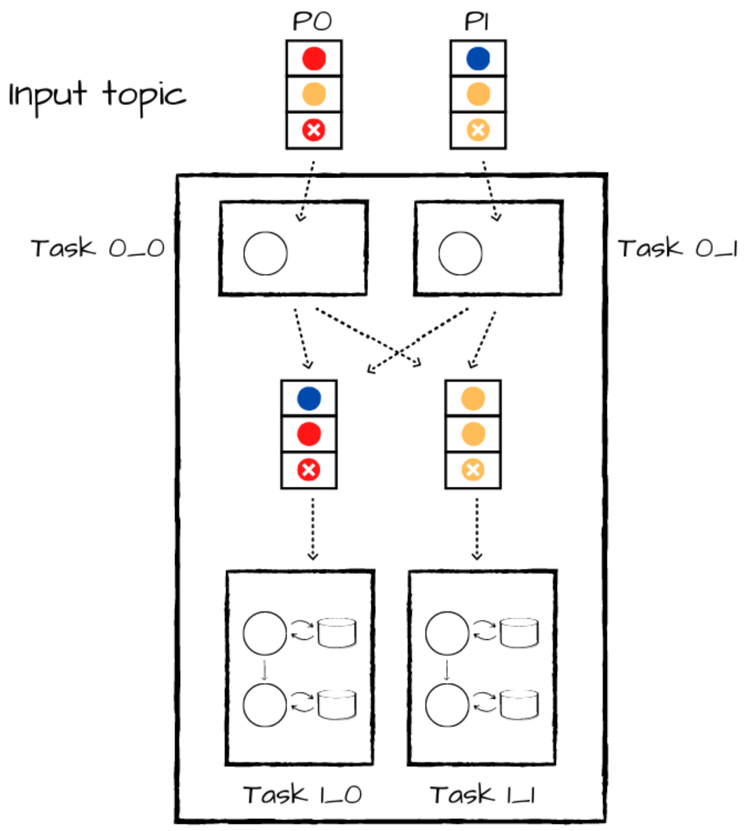


Figure 39: Diagram of a *Stream Constraints* application that includes a repartition process that put events concerning the same entity to the same partition

To address that an additional processing step is needed that will cause the input topic to be repartitioned according to the *link* identifier, see Figure 39. After that, the partitions that can be processed in parallel, see the *Task 1_0* and *Task 1_1*. Kafka Streams provides tools to perform the repartition that can be integrated into a topology with *Stream Constraints*, see example in Figure 40.

```
builder
    .stream("orders")
    .selectKey((_key, value) => value.key)
    .constrain(constraint)
    .to("altered-orders")
```

Figure 40: Streams topology with an additional repartition step

Similar requirements apply to consuming more than one input topic. There is one additional requirement, namely, the number of partitions in all topics needs to be equal. This is necessary for a seamless combination of input topics assuming that they use the same key values and partitioning algorithm, for instance the default implementation that assigns the partition according to the following formula:

$$partition = hash(key) \ \% \ number\ of\ partitions$$

If no partitioning strategy is applied it is necessary to perform the repartitioning step, as described for a single topic.

7.4 NOTION OF ORDER AND TIME IN STREAM

The *Stream Constraints* heavily relies on the order of messages and changes in this order are, next to discarding messages, the main means of enforcing the application invariants. By default, Kafka preserves the order of messages within a partition and each individual message within a given partition gets assigned a unique, increasing integer value of an offset [4]. *Stream Constraints* manipulates the order by preventing some of the messages from being output and publishes them only after some other records have been processed. That results in changes in the offsets and therefore the order.

Apart from the offset, each of the messages has a timestamp assigned. To process the records, Kafka Streams retrieves this timestamp using a timestamp extractor [18]. If no custom implementation is provided that could potentially retrieve a timestamp from the event payload then it returns a timestamp that has been assigned by Kafka. It can be one of two types:

- broker time, the time when the Kafka broker received the message,
- producer time, the time when a Kafka producer sent the message.

The timestamps of processed records create a *stream time* on which window constraints of the *Stream Constraints* operate. If there are no new records processed all the windows remain open as the processor does not know what will be the next timestamp. However, as the stream processing operates on continuous data, such difficulties usually do not arise.

The *Stream Constraints* focuses on altering only the order of messages as the analyzed message consumers from the case study repositories do not show any examples of logic that would rely on message timestamps. For instance, in the Risk Management or the Customer Self-Service in the Lakeside Mutual messages are consumed sequentially and handled according to the specified logic.

Nevertheless, if the *Stream Constraints* were to rely only on the order of records the window constraints must have been expressed in a maximum number of records that can be processed between a given pair of events. This approach seems to be counterintuitive and introduces a number of factors that need to be additionally considered like the exact number of messages that are being produced or the number of partitions. Therefore, the *Stream Constraints* builds upon the well-defined and popular concept of stream windows that use timestamps of the messages.

Messages within a partition are sorted by both offset and timestamp only when the records use the broker time. In any other scenario, for instance when the records employ the producer time, the timestamps may not be in order. This is a result of possible retries in publishing some of the messages or using multiple instances of a producer.

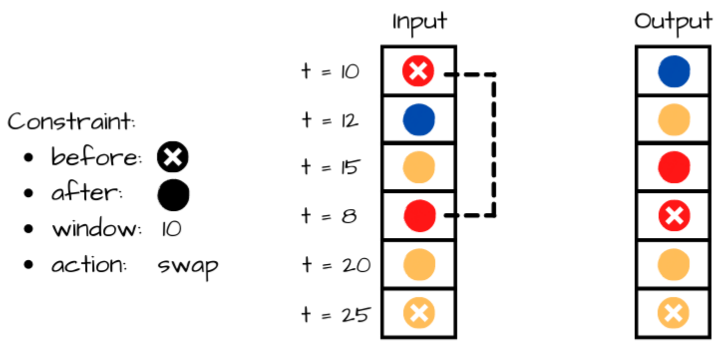


Figure 41: Applying a window constraint on a out-of-order *after* record

The *Stream Constraints* supports out-of-order records that are received before the stream time advances by the window duration time. In that case, an absolute value of the difference on timestamp is calculated and, if the difference is less than the window, then the specified action is applied. An example of that scenario is presented in Figure 41. Different colors in the stream represent different link values and each individual event has its timestamp assigned by a producer. The window constraint presented on the left is applied to the input stream that contains an out-of-order event, the red record with timestamp 8.

The *Stream Constraints* does not support late out-of-order records so, if the *stream time* advances by the window size, the *before* record will be published. Nevertheless, it is advised to use the broker time assignment that unambiguously eliminates those issues.

8 CONCLUSION

This chapter presents conclusions on enforcing event-based constraints in event-driven microservices and its effect on data consistency. The summary reflects on the limitations of the research and presents recommendations for future work on the topic.

This research aimed to investigate means of achieving data consistency in event-driven microservice architectures. By analyzing popular open-source repositories the research has shown what the common assumptions and the ad-hoc mechanisms to handle event processing are. In order to allow developers to replace the suboptimal solutions encoded in the application layer, we propose the *Stream Constraints*, system-level support to define application invariants through event-based constraints. The *Stream Constraints* provides abstractions and guidelines to reason about data correctness and integrity in distributed systems. The constraints enforcement system developed on top of Kafka Streams minimizes the effect of possible event interleavings and ensures that the output stream published to the consumers preserves the defined invariants. The experimental evaluation on the example of the Lakeside Mutual application shows that the *Stream Constraints* is able to efficiently eliminate anomalies in event-based communication.

8.1 FUTURE WORK

The *Stream Constraints* allows developers to define event-based constraints in three categories: causal, terminal, and window-based. Those types were identified after the analysis of open-source repositories and the constraints cover a wide range of use cases of real-world microservice applications. However, the proposed constraints do not exhaust all possibilities of event relations and application invariants. To further expand the proposed solution and improve its usability, future studies could introduce new types of constraint categories. Next constraint categories could draw on identified relations between events defined by Hildebrandt et al. [15] such as the *response* relation that defines which events must happen after a given event occurrence. Another research direction is to accommodate the number of processed events in the constraints. Some application invariants may require that there is only a single event of a given type processed, for instance an event reflecting the creation of an entity, while other types may occur multiple times such as events representing changes in the state of the entity.

The causal and window constraint abstractions provided by *Stream Constraints* are limited to defining only pairs of event types. Although the pairs may be combined through common nodes into more complex definitions, further research is needed to extend the abstraction and support more than two inputs per constraint.

Further research is also needed to provide better support for the constraints that have not been fulfilled. The experiments on policy events described in Section 6.2 have shown that events with a missing prerequisite will be buffered in anticipation of the precondition. However, such event may never come due to a failure in the producer or other scenarios that result in data loss. The developers should have the possibility to specify how the system reacts in a situation when a constraint cannot be fulfilled for a long period of time, for instance whether the events should be indeed buffered infinitely or the system should issue an alert after a given timeout. The *Stream Constraints* employs a basic functionality of the redirect topic that allows developers to investigate and manually handle anomalous events. Nevertheless, more advanced solutions are needed that could for instance accept a custom processing logic that will be triggered for each discarded event.

Kafka Streams provides tools for querying the state stores that are used in the stream processors. As the *Stream Constraints* solution heavily depends on the state stores for saving views of processed events and buffering records, practitioners could consider exploiting those querying functions in order to provide the developers with a better understanding of the current state of the processing.

BIBLIOGRAPHY

[1] *ActiveMQ Documentation*. (Accessed on 2021-05-07). URL: https://activemq.apache.org/components/classic/documentation.

[2] Peter Alvaro et al. “Consistency without Borders”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: Association for Computing Machinery, 2013. ISBN: 9781450324281. DOI: 10.1145/2523616.2523632. URL: https://doi.org/10.1145/2523616.2523632.

[3] *Apache Flink Documentation*. (Accessed on 2021-05-10). URL: https://ci.apache.org/projects/flink/flink-docs-master.

[4] *Apache Kafka Documentation*. (Accessed on 2021-05-07). URL: https://kafka.apache.org/documentation.

[5] *Apache Spark Streaming Documentation*. (Accessed on 2021-05-10). URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL Continuous Query Language: Semantic Foundations and Query Execution”. In: *VLDB J.* 2 (Mar. 2004). DOI: 10.1007/s00778-004-0147-z.

[7] Peter Bailis et al. “The Potential Dangers of Causal Consistency and an Explicit Solution”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391251. URL: https://doi.org/10.1145/2391229.2391251.

[8] Valter Balegas et al. “IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases”. In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), pp. 404–418. ISSN: 2150-8097. DOI: 10.14778/3297753.3297760. URL: https://doi.org/10.14778/3297753.3297760.

[9] Valter Balegas et al. “Putting Consistency Back into Eventual Consistency”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

[10] Gianpaolo Cugola and Alessandro Margara. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Comput. Surv.* 44.3 (June 2012). ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: https://doi.org/10.1145/2187671.2187677.

[11] *Data considerations for microservices - Azure Architecture Center*. (Accessed on 2021-05-05). URL: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations.

[12] Dotnet-Architecture. *eShopOnContainers*. URL: https://github.com/dotnet-architecture/eShopOnContainers.

[13] Andrei Furda et al. “Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency”. In: *IEEE Software* 35.3 (2018), pp. 63–72. DOI: 10.1109/MS.2017.440134612.

[14] *Graph Visualization Software*. (Accessed on 2021-05-15). URL: https://graphviz.org.

[15] Thomas Hildebrandt and Raghava Rao Mukkamala. "Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs". In: *PLACES* 69 (Oct. 2011). DOI: 10.4204/EPTCS.69.5.

[16] Haruna Isah et al. "A Survey of Distributed Data Stream Processing Frameworks". In: *IEEE Access* 7 (2019), pp. 154300–154316. DOI: 10.1109/ACCESS.2019.2946884.

[17] Pooyan Jamshidi et al. "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35. DOI: 10.1109/MS.2018.2141039.

[18] *Kafka Streams Documentation*. (Accessed on 2021-05-10). URL: https://kafka.apache.org/documentation/streams/.

[19] *ksqlDB Documentation*. (Accessed on 2021-05-10). URL: https://docs.ksqldb.io/en/latest.

[20] Rodrigo Laigner et al. *Data Management in Microservices: State of the Practice, Challenges, and Research Directions*. 2021. arXiv: 2103.00170 [cs.DB].

[21] Rodrigo Laigner et al. "From a Monolithic Big Data System to a Microservices Event-Driven Architecture". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 213–220. DOI: 10.1109/SEAA51224.2020.00045.

[22] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[23] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*. 2016. arXiv: 1605.03175 [cs.SE].

[24] James Lewis and Martin Fowler. *Microservices*. (Accessed on 2021-05-02). URL: https://martinfowler.com/articles/microservices.html.

[25] Shan Lu et al. "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics". In: *SIGOPS Oper. Syst. Rev.* 42.2 (Mar. 2008), pp. 329–339. ISSN: 0163-5980. DOI: 10.1145/1353535.1346323. URL: https://doi.org/10.1145/1353535.1346323.

[26] David C Luckham and Brian Frasca. "Complex event processing in distributed systems". In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998), p. 16.

[27] Antonio Messina et al. "A Simplified Database Pattern for the Microservice Architecture". In: June 2016. DOI: 10.13140/RG.2.1.3529.3681.

[28] Brenda M Michelson. "Event-driven architecture overview". In: *Patricia Seybold Group* 2.12 (2006), pp. 10–1571.

[29] Microservice-API-Patterns. *Microservice-API-Patterns/LakesideMutual*. URL: https://github.com/Microservice-API-Patterns/LakesideMutual.

[30] Microsoft. *Communication in a microservice architecture*. (Accessed on 2021-05-01). URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[31] Hannes Obweger et al. "Entity-Based State Management for Complex Event Processing Applications". In: *Rule-Based Reasoning, Programming, and Applications*. Ed. by Nick Bassiliades, Guido Governatori, and Adrian Paschke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 154–169. ISBN: 978-3-642-22546-8.

[32] Claus Pahl and Pooyan Jamshidi. "Microservices: A Systematic Mapping Study". In: Jan. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[33] Dan Pritchett. “BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability.” In: *Queue* 6.3 (May 2008), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128. URL: https://doi.org/10.1145/1394127.1394128.

[34] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. “Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective”. In: *IEEE Software* 35.3 (2018), pp. 36–43. DOI: 10.1109/MS.2018.2141028.

[35] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. ISBN: 9781617294549.

[36] Chaitanya Rudrabhatla. “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture”. In: *International Journal of Advanced Computer Science and Applications* 9 (Jan. 2018). DOI: 10.14569/IJACSA.2018.090804.

[37] Sentilo. *Sentilo*. URL: https://github.com/sentilo/sentilo.

[38] Facebook Open Source. *RocksDB*. (Accessed on 2021-05-16). URL: https://rocksdb.org.

[39] Nicolas Viennot et al. “Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741975. URL: https://doi.org/10.1145/2741948.2741975.

[40] Eugene Wu, Yanlei Diao, and Shariq Rizvi. “High-Performance Complex Event Processing over Streams”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 407–418. ISBN: 1595934340. DOI: 10.1145/1142473.1142520. URL: https://doi.org/10.1145/1142473.1142520.