The present work was submitted to Lehr- und Forschungsgebiet Educational Technologies at DIPF

An Exploration into the Architecture and Implementation of a Real-time Learning Analytics Engine

Bachelor-Thesis

Presented by

Schwind, Tonio

5195736

First examiner:	Prof. Dr. Hendrik Drachsler
Second examiner:	George Ciordas-Hertel

Frankfurt, September 12, 2019

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed. Equally I confirm that neither this work as whole nor parts of it have been used in an exam or course achievement before.

Frankfurt, September 12, 2019 Tonio Schwind

Abstract

This thesis aims at an exploring implementation of a real-time learning analytics engine based on Kafka and Kafka Streams. A real-time learning analytics engine should be able to make results of analytics run on data by a multitude of sources available near the time of the data's arrival in the system. Amongst these source may be traditional relational databases as well as streams of data from sensors or micro-controllers. Such learning analytics engine would allow for delivering observations and indications to actors in the learning environment whilst the relevant action itself is happening, enabling a fluent interaction of the system and its environment. To build such system this thesis consists of two parts. In the conceptual part the κ-Architecture is evaluated as a derivation of the λ-Architecture. While requirements defined by aspects of big data as well as devops may be well met by the κ-Architecture, doubts regarding its use for a learning analytics engine are articulated. Given the Log-based conception of Kafka, problems of compliance with the GDPR are articulated, though not solved. Next, some conventions for the system are suggested and discussed, includings naming patterns, data structures and formats. This prepares more concretely the second part, the implementation: A dockerized, horizontally scaleable system based on Kafka and Kafka Streams applications will be implemented that reads, transforms and processes two exemplary, heterogenous sources of data: As data with low velocity relations from a Moodle's MySQL database are streamed with Kafka Connect, as data with high velocity frames from a leap motion sensor are streamed with an adapter to its default websocket. The results, some simple statistical evaluations made possible by joins and aggregations of several streams, will be presented in a continous view.

Contents

1 Introduction

1.1 Overview

This thesis aims at an exploring implementation of a real-time learning analytics engine based on Kafka and Kafka Streams. A real-time, or more precisely, a near-real-time learning analytics engine should be able to make results of analytics run on data supplied by a multitude of sources available *near* the time of the data's arrival in the system. Amongst the sources may be traditional relational databases as well as streams of data from sensors or micro-controllers. Such learning analytics engine would allow for delivering observations and indications to actors in the learning environment whilst the relevant action itself is happening, enabling a fluent interaction of the system and its environment. Specifics to the pedagogical techniques, values and ends as well as to canny algorithms working towards these ends are left to further research; this thesis will focus on the architectural, broadly technical implementation. To do so a containerized, specifically, a dockerized system based on Kafka and Kafka Streams applications will be implemented that reads, transforms and processes two exemplary, heterogenous sources of data; the results will be presented in an unspectacular continous view. This system will allow for the horizontal scaling of its applications via docker-compose orders thereby allowing for high throughput at low latency.

The two sources, implemented exemplarily, are different with regard to their velocity, as well as with regard to applied indicators.

1. The first shall represent highly complex data that demands for complex analytics operating over a multitude of different streams and their aggregations – i.e. complex event processing [DP18]; this data has a low velocity. Relations from a Moodle database representing Moodle-User, -forums and -forum-posts will be taken as an example for this class of data.
2. The second class represents data of low complexity that equally demands for simple event processing, by which an operation is meant that operates rather on each of the stream's unit itself than on a multitude of streams and their aggregations. Such data can be handled even under high velocity as each analytical function has low demands on resources. Frames – around a 100/s – from a Leap Motion sensor will be taken as an example.

The implementation, further, will merge both kinds of classes in one indicator. Here, data of high velocity coming from a sensor will be joined with data from the Moodle database that is user-produced. The exemplary implementation therefore realizes two kinds of indicators; one

in which a multitude of joins on low-velocity data is operating, and one in which a stream of high velocity from a sensor is joined with a stream of low velocity.

1.2 Research Questions

The research questions related with this thesis work, i.e. the implementation of a streaming learning analytics engine that operates on two heterogenous sources of data, are:

1. On architecture
   - Is the κ-Architecture from a conceptual perspective an adequate architectural choice with regard to a real-time learning analytics engine?
   - How could a κ-Architecture for a real-time learning analytics engine look like?
   - How does Kafka and Kafka Streams operate, how do they fit into the κ-Architecture in general and the learning analytics engine specifically?
2. On the Learning Analytics Engine
   - What conventions should the learning analytics engine and its applications adhere to?
   - How does devops look like within the suggested Learning Analytics Engine? How does the setup facilitate devops?
3. On implementation
   - Does the exemplary implementation of two heterogeneous data sources and their real-time analysis in a streaming analytics engine pass performance tests? In other words, is the envisioned *real-time* learning analytics engine realizable?

1.3 Outline

This thesis follows a trajectory that continuously increases conceptual resolution, i.e. zooms in onto the final implementation. First, in chaper 2.1 this thesis will be broadly situated as a thesis on big data. In chapter 3, the theoretical core chapter, an overview of the broad architecture applied to the system as well as involed backbone software will be set forth. Both together supply a basis for assessing the architectural decision made *ex ante*. This chapter's first section explains the κ-Architecture as a derivation of the more complex so-called λ-Architecture. This overview allows for an abstract understanding of the management of data, its storage and processing. Its second section takes a dive into the fundamentals of Kafka and Kafka Streams and Kafka's ecosystem: the explanation of the ontological concept of the Event and the Log will provide the basis for terms defining Kafka, like the log, the stream, the stream-table duality etc. Its final section evaluates the decision for the κ-Architecture. While the κ-Architecture based on the Kafka ecosystem fullfils many demands of a real-time big data system, some doubts regarding the κ-Architecture's functionality for the problem of a learning analytics engine will be cast. After a short excursion, in chapter 4, into the problems that the architectural decision and Kafka as a Log pose for the adherence to demands of anonymization and the GDPR, in chapter 5 conventions for the streaming system to be implemented will be defined. These comprise naming patterns as well as considerations concerning the streams as interfaces, i.e. the data

structures that are communicated via streams which constitute the sole *contract* between the system's applications. Given these conventions, in chapter 6 some remarks concerning used technologies beyond Kafka and Kafka Streams are given as well as the concrete implementations and configurations explained – as far as it seems sensible. In the last proper chapter, chapter 7, a how-to for starting the system and results of applied performance tests are presented. Finally, a conclusion with regard to outlined research questions and an outlook is put forward.

2 Contextualizations

2.1 Big Data

Given the heterogeneity of data sources involved in a learning analytics engine as well the sheer size one can envisage, e.g. given a university as the learning environment, this thesis' work is firmly situated in the context of big data. Big data – in the following more precisely, if somewhat roughly defined – is a heading for a kind of data analysis that occured and drew a lot attention as well as capital during the last 15 years. It marks a technological development, both concerning hardware and software, that enabled persisting and analyzing not only huge amounts of data, but especially data of varied kinds. This enabling development, amongst others, was the increasing feasibility of storing massive amounts of data, the development of cloud computing and the designability of distributed systems which both capitalize on and are technological answers to the problems posed by the existence of big data.

For the purpose of this thesis Buyya, Rodrigo, and Dastjerdi [BRD16] will be followed in their definition of big data that is induced from the history of its discourse to reach a broadly defined, all-encompassing term. With Buyya et al. [BRD16], semantically, big data refers to a complex of three aspects. While *Data* refers to the type of objects dealed with, *Statistics* refers to the value of objects as grounds for statements and *Business Intelligence* to the instrumentality of such statements [Fel+15].

Big Data [BRD16; Fel+15]		
Semantics	Detailed Semantics	Pragmatics
Data	Velocity	Massive datasets ETL Statistics
	Variety	
	Volume	
Statistics	Validity	Machine Learning
	Veracity	
	Variability	
Business Intelligence	Visibility	"Cloud computing (cost effective infrastructure)"
	Value	
	Verdict	

This thesis is an instance of the *Data* aspect. It neither conceptualizes on the retrievability of some kind of information from some other kind of information (*Statistics*) nor on the original motivation of educational technologies to analyze some or other information (*Business Intel-*

ligence). The pragmatic coinside to the *Data* aspect, in the above tabular on the right side, therefore marks more concretely this thesis' area of concern. However, insofar as the architectural decisions made here have an impact on the other two operational adequacies, for instance, the setup of a streaming engine determines the availability of frameworks for artifical intelligence operable on streams, they may recur implicitely throughout the text.

In the context of big data, the realizability of distributed systems was made necessary by constraints of computing and storing powers of single server instances. Distributed systems therefore scale computation and storing by distributing processes and states over a set of servers. This brings complexity not only into questions of data consistency over redundant data and of performant retrieval, but also new issues of communication, orchestration or composition.

Technology cluster like the Hadoop environment allowed operationalizing big data in a fault tolerant, distributed way. However, as a batch processing system, what Hadoop not allows for is real-time event processing; increasingly this is of interest [GGR16] and technology is available. This step into event processing systems marks the increased concern with the velocity of data. For real-time or near-real-time, event processing systems fundamentally different conditions are valid, as data is not statically accessible, but enters the system as a *stream* of data. To be able to scale, such streams accordingly have to be devided and distributed over a set of server – the most natural way to scale therefore is horizontally[1], i.e. the distribution of streams or parts of streams of data to full replicas of the system's applications. This allows for a model of parallel execution and as such to handle data of high-velocity, high-volume and high-variability while at the same time keep latency low and the system's availability high [BRD16]; horizontal scaling is a distributable simple answer to streams of big data. As will be made explicit, Kafka is streaming platform that allows for exactly such horizontal distribution of data streams.

2.2 Similar Projects

The following section gives an overview of similar projects publicly communicated and accessible. As was indicated above, many, if not most systems operate on the basis of static data made available by databases, e.g. in a Hadoop/Yarn cluster: Map-reduce processes do the batch processing here. These exist at least since the late 2000s. A real-time or near-real-time analytics systems is not of much younger nature, but has been taken up speed only during the last five to ten years.[2] Accordingly literature on batch processing systems exist, typical examples are analytics operating on health- and medical-data or electricity grids. With regard to the *implementation* of streaming architectures – other than on architectures themselves –, literature is less available. The author could retrieve only a small set of projects being described, most of which further are *either* architectonically rather similar to the thesis' one *or* implement some specifities that are of no interest to a real-time learning analytics engine. Especially the relevant internal structures were irretrievable, as direct access to software repositories and concrete implementations was not trackable.

Vogten and Koper [VK18] suggest a system rather matching the one elaborated on in this the-

[1] Vertical scaling, i.e. the adding of resources to instances of the system, is due the conceptually necessary restrictions of hardware and software only a seeming solution.

[2] This may be roughly deducible from the long list of appearances and increased importance of real-time streaming platforms and processing environments in Apache's ecosystem during this timespan, e.g. Apache Kafka, Apache Spark, Apache Flink, Apache Flume, Apache Druid, Apache Heron, Apache SAMOA etc.

sis. In an internal review paper Schneider et al. [Sch+] suggest another similarly event-based system with focus on *trust*, i.e. the system's ability to handle strong user-rights. Peña, Rua, and Lozoya [PRL16] design a system analysing traffic in real-time with a focus on its elasticity, i.e. its adaption to increasing or decreasing loads; they further compare an implementation of the ELK stack with self-implemented lightweighted java applications. Versaci, Pireddu, and Zanetti [VPZ17] implement a system based on Apache Kafka and Apache Flink to horizontally scale gnome sequencing. These projects all implement a κ-esk architecture and base their data distribution on Kafka, some their analytics on Kafka Streams applications. Partially they differ from this thesis in that they actually sink data into data stores of a serving layer as well as into a storage mechanism based on HDFS – while this thesis suggests to simply persist data, including analytics' results, in Kafka and make it accessible in cached materialized views directly produced and backed by Kafka Streams – as will become clear in the next chapter.

3 Architectural Design and Corresponding Technologies: Kafka's Ecosystem

Of central import to any real-time system that is to be able to handle big data is its architectural design. That architecture sets constraints to used technologies and frameworks, to the way data flows and is stored, to *devops*, etc. The κ-Architecture used in this thesis was first brought forward by Kreps [Kre14c], and is in the sense of a broad architectural suggestion simple. However, as the κ-Architecture is a derivation, more precisely, a simplification of the λ-Architecture, to place it properly an elaboration of the latter is needed.[1] Before, however, the λ- and the κ-Architecture are elaborated on, some broad norms against which the architectures fitness is measured need to be exposed. This chapter's first section should therefore supply an understanding of the κ-Architecture as a simplification of the λ-Architecture; it should *suggest* how the κ-Architecture handles real-time data streams as well as declare on what kind of conditions, especially what kind of distributional mechanisms, it rests. In this chapter's second section, then, technologies satisfying these preconditions are presented. That rather technical section will explain the basics of Kafka and Kafka Streams and elaborate on some of Kafka's further ecosystem. Both will be brought together in a rough evaluation of the κ-Architecture in the chapter's last section.

3.1 Architecture

3.1.1 Architectural Requirements of a Learning Analytics Engine

3.1.1.1 Functional Norms

An architecture is to be chosen according to needs it is supposed to satisfy. While for this thesis the κ-Architecture (more later) was already chosen *ex ante*, it seems sensible to reflect on this choice and consider strengths and weaknesses. For this, some normative orientation is necessary, which will be roughly outlined here.

The architecture has to be able to

1. handle data that enters the system with high velocity.

[1] For a similar argument on architecture and its demands, see Vogten et al. [VK18].

2. handle data that has a high volume.
3. handle data of high variety.
4. process some data in real-time.
5. allow for strong consistency-guarantees.

The points 1. to 3. mark the architecture as one that is to be fit for big data; as such it necessarily implements as a

- scaleable, distributed system with high availability and high throughput.
- system that allows for asynchronous communication.

Point 4. articulates demands for low latency of some transformations. This indeed suggests a departure from traditional architectures allowing for ETL, as it demands the high availability of data together with a low latency of transformations. It makes necessary a

- stream processing engine (which fulfils above demands as well).

Point 5. demands strong guarantees concerning consistency as the system operates in the sensible area of education. False or inaccurate results are to be prevented.

3.1.1.2 Operational norms

Beyond these functional demands, operational concerns matter – and make further functions necessary. The architecture must allow

1. for several teams to be able to act on it.
2. for simple ways to release new versions.

Point 1. equals the demand for a separation of concern as it is known e.g. in agile development. Interdependencies of teams and their programmatic interfaces therefore should be kept as low as possible. A design that allows for each team to handle *devops* itself as well as one that allows for clearly marked and reduced interfaces is preferable to others. Point 2. equals the need for *replayability*, given a stream-based system.

3.1.1.3 Norms derived from research interests

Finally, the architecture should meet research interests in the area of educational technologies. Here it seems sensible to assume the need for

1. the availability of ad-hoc queries and online processing (OLAP) on the data-set to allow researchers experiment on data.
2. the availability of interfaces for ML.[2]

Both points stem from the the aspect of statistics of big data, especially the problem of data mining. Both will be mostly ignored in this thesis.

[2] For an example using Kafka and Ray, see Kato et al. [Kat+19].

3.1.2 The λ-Architecture

The λ-Architecture was originally suggested to handle the tradeoff between latency and throughput of big data analyses, more precisely it was stated as an architectural levelling or solution to the CAP-theorem, or rather, problems with fully incremental databases due to the statements of the CAP-theorem [Mar11; Mar15].[3] For this purpose the λ-Architecture suggests to have a batch processing system that periodically processes *all* data parallel to a stream processing system that processes *real-time* data.

3.1.2.1 Batch processing

While batch processing has a high throughput, it has equally a high latency; a batch process can take from hours to days and therefore allows no real-time or near-real-time event processing. Such batch process operates, for example, in a Hadoop environment; as such it typically stores data in HDFS.
Batch processing delivers some advantages [Mar11; Mar15]:

- "Algorithmic flexibility": The batch processing operates on the full dataset, implying that common algorithms over present data are applicable.
- Common knowledge: The batch processing engine underlying this part of the λ-Architecture, i.e. e.g. a Hadoop cluster, is well known and is a stable system [Mar15].
- "Easy ad-hoc analysis": The data is fully available at runtime, arbitrary queries with random access on that data are possible and do not produce any further complexity.
- Easily changed schemas: As all data is available in one system, this system can run transformations on all data that, among others, can account for changed data structures.

3.1.2.2 Stream processing

To achieve low latency the λ-Architecture foresees a streaming system. This speed or streaming layer processes the data that arrived since the last batch process using incremental algorithms on state and each event.

Once a batch result is produced, it overwrites the intermediate results from the speed layer. For Marz such a dual system has several advantages ontop of the leveraging of latency and throughput of a distributed system and the above name features, some of which are (see again, for a full account Marz [Mar15]):

1. Reduction of Complexity: Since for Marz complexity is mainly involved in the speed layer, errors happening there do not last as the next batch result overwrites the result of the stream processing. The stream layer's complexity therefore is isolated [Mar15].

[3] According to the CAP-theorem there is a necessary negative interdependency between consistency and availability in partitioned, distributed systems such that not both at the same time can be high.

2. Availability of "eventual accuracy" [Mar11; Mar15]: The stream layer, operating with incremental algorithms, in cases of highly complex calculations, can implement fast, approximating functions since the exact results will be stored durably with the next batch results. This is a feature not to underestimate as not every algorithm is or is easily implementable as an incremental algorithm.
3. Performant Stream Processing: The stream layer only operates on a relatively small amount of data, since it only has to account for data that arrived since the last batch processing began.
4. Replayability: A feature stressed by both Marz and Kreps is replayability, i.e. the applicability of new versions of algorithms on data. The λ-Architecture makes all data available in the mass storage, e.g. a HDFS [Kre14c; Mar11].
5. Human-fault tolerance: Replayability as well as the non-durability of stream processing results reduce the risk of permanently damaged data due to programmatic errors.

3.1.3 Critique of the λ-Architecture

Given all its features, the λ-Architecture runs into often diagnosed problems with regard to development and management, i.e. *devops* [Kre14c; Mar15, for a fuller picture, with pros and cons].[4]

1. Foremost, the complexity of developing and managing *two* systems, which do – or shall do approximatingly – the same, only operating on different types of temporalities, i.e. batches of objects respectively streams of objects, is high [Kre14c]. This is an suggested by Kreps as an extreme operational concern that outweights the λ-Architecture's features.[5]
2. There is some, if manageable, complexity involved in the synchronization of the batch processing system with the stream processing system.

Given all the above features, given its ability to make the conflicts of consistency and availability in distributed systems, i.e. the problems posed by the CAP-theorem, manageable, Kreps' critique in effect suggests that they are made possible *just*[6] by externalizing these intrinsic complexities into the two layers' operational consistency.

[4] The claim that the λ-Architecture "beats" the CAP-theorem surely is a little far fetched [Mar11]; it seems to me that it is a conceptual truth that any asynchronous system can only reach *eventual* consistency and has necessarily a tradeoff between availability and consistency – if it only is the availability of the results of batch processing or the potential, even calculated inconsistency between a batch result of time t_B and every event processing result of time $t_B + \lambda$, $given\ \lambda > 0$, an event has been processed between $t_B\ and\ t_B + \lambda$. To be fair, contrary to the language used by Marz [Mar11], his statement is properly read as a statement about the ease of managing eventual consistency (and accuracy) by making the badge process the truth over every incremental stream process [Mar11; Mar15].

[5] Albeit, frameworks that allow code to be compiled into both stream and batch processing byte-code seem to have been developed, with some restrictions on implementable functionalities [Ber14].

[6] A just to be read with care.

3.1.4 The κ-Architecture

The κ-Architecture, as it was suggested by Kreps [Kre14c], tries to find a solution to this problem of complex devops by simplification: it suggests simply to get rid of batch processing. The κ-Architecture therefore is a system composed of pure streaming applications and its processors all operate as event processors.

Kreps, being the core developer of Kafka, suggests this architecture clearly with Kafka in mind (more on Kafka in the section on Kafka and Kafka Streams). The κ-Architecture accordingly foresees a Log-based streaming engine that replaces a distributed database as a storing mechanism. Such Log-based streaming engine enables the following features:

- Scaleability: The log allows for the horizontal scaling of applied algorithms by running multiple instances of the same application parallel, thereby handling big data aspects of high velocity and high volume.
- Replayability[7]: New versions of algorithms are applicable to data by releasing the new versions parallel to existing ones; these new versions will replay each and every event noted in the log beginning with the very first. In effect, this procedure will double (upper bound) needed disk space, memory and traffic, until the new version have replayed fully and the old versions can be removed.

To evaluate the κ-Architecture's abilities to fullfil architectural demands outlined above a better understanding of needed technologies seems necessary.

3.2 Kafka, Kafka Streams and Elements of Kafka's Ecosystem

Kafka and Kafka Streams build the backbone of the κ-Architecture that Kreps had in mind as well as the here developed learning analytics engine. While Kafka is the system that holds data published in streams and makes it available to consuming applications, Kafka Streams is an API that allows accessing data in the form of filters, mappings, joins and aggregations on streams. As they are the engine's engine it is indispensible to supply an explanation of the workings of Kafka and Kafka Streams.

In the following, first, the ontology of Kafka will be circumscribed. Core concepts on an abstract level are the concept of the (single) *Event* and the plurality of events gathered in a *Log*.[8] These concepts are fundamental abstractions of the Kafka ecosystem. After they have been defined, Kafka and Kafka Streams will be elaborated further. Finally, some of Kafka's ecosystem's further applications will be roughly outlined, among them Kafka Connect, the schema registry and KSQL.

[7] Restreaming not from a database, e.g. HDFS as in the λ-Architecture, but from the streaming engine itself became possible with Kafka, which is one of the first Log-based streaming engine as which it allows to have an infinite retention time [DF16].

[8] Capital letters are used to denotate concepts on the ontological level and to distinguish these from the usage of the same terms in the description of specific implementations.

3.2.1 Fundamental Abstractions

3.2.1.1 The Event

The *Event* denotates the reality of a digital object at its most basic, at an *atomic* level through the lense of a specific conception of time. The Event *happens* at a *specific moment of time*; it therefore is temporally indivisible. In contrast to this relation to time, the Event – per notion – is not bound by space, indeed might not be localized at all [CV14]. As such, the Event needs to be registered to be made persistent and referenceable diachronically.

The Event that happens at a moment has two aspects to it.

1. It always is a *notification*. As such it has a signalling character that demands to be registered, and may be reacted on.
2. As this demand points to the Event itself, the Event always also has *meaning*, i.e. comes with data constituting the Event itself [Sto18; Ber18].

These two aspects are of import as well as they are conceptually interdependent: the Event declares that *something* happened *now* and necessarily *what* that something is.

The digital object and its changes – as it was introduced above: perceived through time conceived coarsely – are constituted by a stream of Events. We could go sofar – for our purposes – as to conceive the digital object as Event in its historicity, as Event over time, as diachronic event. In colloquial terms, the (digital) object is the *result* of a history of Events making up (digital) reality as it is *now*.

3.2.1.2 The Log

As the Event was defined above, an object *is* if and only if a series of Events is *happening*; further, given the above definition, such series of Events only can *happen* when each of the series' element is registered. This registry of a series of Events can be called the Log. The Log *only registers* a history of notifications and meanings. This implies a general statement about truth, as is well known in the discussion on Kafka: "The truth is the log" [Hel15; Kre13; Kre14b]:

1. The Log constitutes a linear and directed time which necessitates an order to its Events by the order of registration. If a series of Events references the same object, the latest notification points to the object's state *now* [Hel15].
2. Accordingly, the Log's entries are history. They therefore cannot be changed, they are immutable data.

3.2.1.3 Excursion: Imperative vs Declarative Programming

Unregistered Events are no Events; the necessary registration of Events in a Log conceptually necessitates a fundamental asynchronicity in the system. In a more technical language, published Events on their own have no behavior, they are notifiactions consisting of data. Given the

public character of the Log, something can act upon an Event and can do so at its own pace. A subject, an application acts upon an Event and thereby issues other Events. A system that composes "services not through chains of commands and queries, but rather through streams of events" [Sto18] therefore follows a declarative rather than imperative style of programming.

3.2.2 Kafka

Kafka is an instance of the abstract concept of the Log implemented in Java [Kre13; Kre14b; Kre15a]. In broad terms, "Kafka is a streaming platform" [Sto18; Kre]: Kafka's implementation of the Log allows for *streams* which are consumable, temporally unbounded sequences of structured data. Kafka as a streaming platform fullfils two basic functions of real-time, distributed big data system [Kre15a].

1. Kafka functions well as a data integration system, i.e. as a pipeline that connects different kinds of applications, producing as well as consuming, transforming or sinking applications. Streams are the "shared source of truth" [Sto18] in an asynchronous system of aplications coupled via Kafka. Kafka adheres to the "idea of pure data movement" [Sto18], it follows the logic of "smart endpoints" and "dump pipes" [Fow14].
2. Kafka functions well as a root system for stream processing as it captures Events.

In that, Kafka is more than merely a message broker: Kafka persists messages durably on disk in a distributed, fault tolerant way and makes them available [Kre]. As the lifespan of registered events, i.e. the retention time, can, but must not be restricted, Kafka can hold a full history of events as immutable data in logs [Kre].

Given the level and way of persistence of Events in Kafka, Kafka might be better understood as a distributed database that stores and makes accessible transactions themselves, i.e. Events, that means it might best be understood as a database turned "inside-out" [Kle15; Sto18; Ber18; Kre15a]. However, it differs from databases – and that makes it very much unlike a database – in that Kafka does not allow for random access.

Kafka's logs, i.e. *topics*, are partitioned which means that every specific log is divided into several, distributed Logs which cohere into one *stream*. Kafka's Log is "essentially a set of append-only files spread over a number of machines" [Sto18], where each file is a partition. Kafka guarantees order of time in each partition, not across partitions as this would forfeit the purpose of partitioning – that is horizontal scaling. Further, a stream's partitions can be compacted; while normally, the log holds all published events, a compacted log eventually holds for each message key the last publication only.

A *Kafka Producer* is an application that spits data into a Kafka stream [Conb]. Broadly three things happen:

1. The datum is serialized into a byte-format.
    - While the datum is serialized always into a byte-format, the data structure that is serialized is of import. Data structures build the glue between producers and consumers, they are the sole contract interweaving applications in a pure Kafka system and allow for their "[l]oose coupling" [Sto18].[9]

[9] However, the intent of Kafka systems to publish Events as broadly as possible has a hindsight; the publication,

- Given the import of data structures communicated into Kafka, solutions to the problem of their evolution were developed. While it is possible to publish JSON into Kafka, this has the disadvantage of inflating traffic and disk space. One common alternative to JSON is the usage of Avro formats in Kafka systems; Avro allows for the publication of a highly compacted object by removing meta-data like fieldnames from the message; to this end a versioned schema is linked to the message, which allows for the objects deserialization. As such schema can be stored centrally, it heavily reduces traffic and disk usage.

2. The topic, in which the datum is persisted and consumable from, is determined.
   - A datum without a message key is by default sent round-robin to topics, but topics can be specified explicitely, too.
   - A datum with a message key is identified by the key's hash-value; it therefore is guaranteed that messages with the same key end up in the same topic – a necessity to ensure that the diachronic identity of objects is guaranteed for.
3. The datum is sent to the Kafka broker responsible for the partition, i.e. the partition's leader. The leader's replica reproduce the production eventually. It is configurably how strong the guarantee of persistence to the topic is supposed to be:
   - Ack 0: Neither partition's leader nor any other Kafka broker acknowledges the publication; in case of the write's failure no recovery is possible. This increases throughput.
   - Ack 1 (default): The partition's leader, but not its in-sync-replicas achknowledge the publication to the producer. In case of the leader's failure before replication no recovery is possible.
   - Ack 2: Both the partition's leader and its in-sync-replicas acknowledge the publication, which ensures recovery even in case of the leader's failure but lowers throughput.

A *Kafka Consumer* is an application that reads data asynchronously from a Kafka stream [Cona]. Several processes happen when a consumer starts reading from a topic.

1. Applications consuming from Kafka are necessarily part of a declared *consumer group* which defines a set of applications that consume from a stream and that is coordinated by a designated Kafka broker (one of the stream's partitions' leader). The concept of a consumer group and its coordination ensures that each and every of the stream's partition is consumed from, such that each topic is assigned to exactly one of the consumer group's applications.[10] If the consumer group's formation changes, the Kafka broker "rebalances" the group, i.e. reassigns the partitions.
2. Consuming applications read from Kafka by polling the partition, i.e. the partition's leading Kafka broker. Micro-batches of messages, if available, are read.
3. Kafka broker keep track of the consuming applications' offset on a partition. In case of an initial read or an offset out of reach, consumption can start

i.e. the asynchronous equivalent to the synchronous remote procedure's endpoint, of data structures as large as possible without regard to demands of consuming applications is a known anti-pattern as it increases the coupling and therefore generates potential locks on the sources' development.

[10] To this effect, every consuming application sends heartbeats to the coordinator.

- at the earliest offset.[11]
- at the latest offset (default).
- at a customized offset.

In case of a continued, from another application of the same consumer group overtaken partition, the instance continuing consumption starts from the partition's last offset known. To reach this effect, Kafka consumer need to commit their offsets to Kafka. The offset commit policy is configurable and of import to delivered guarantees. Central configurations are the following:

- Auto-commit (default): The consumer commits its offset periodically according to its commit interval (default 5s). This guarantees at-least-one, but allows for as many duplicates as can be consumed in the commit interval.
- Commit first: Instead of committing after the processing of consumed messages, a commit can be done first. This guarantees at-most-one.

As is intuitable, scaled applications belong to the same consumer group.[12] The concept of a consumer group and its logical effects therefore allow for the horizontal scaling of the system's participants, for its distributedness.

[11] This allows for replayability, an essential feature.

[12] This implies that the number of partitions, and *by implication* the number of Kafka brokers, is an upper bound for the horizontal scale of consuming applications.

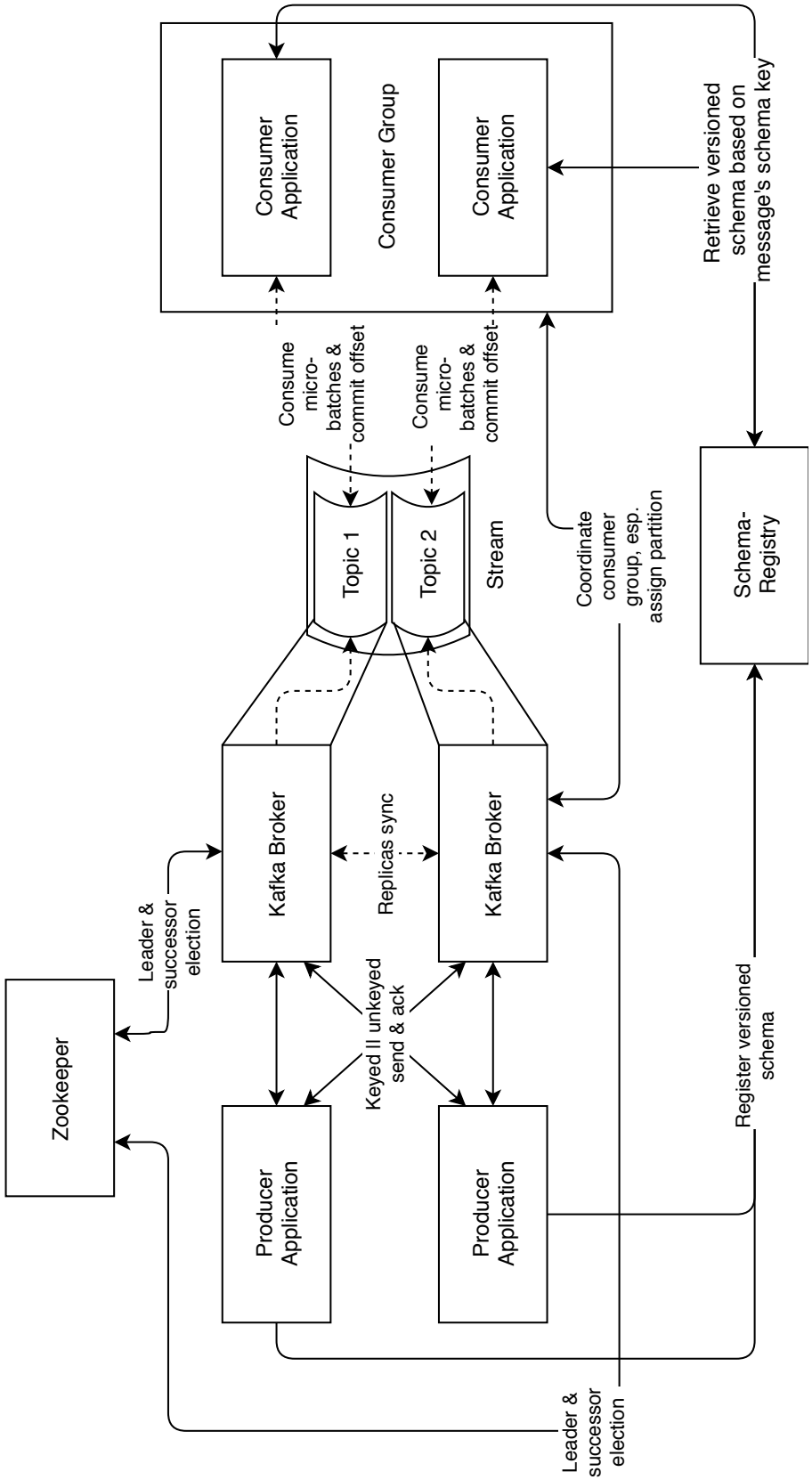

Figure 3.1: A Kafka System with a schema-registry (scaleability = 2).

3.2.2.1 Excursion: Eventual Consistency

As a data integration pipe Kafka enables asynchronous message distribution among an arbitrary number of applications. It channels asynchronously data from an arbitrary producer to an arbitrary consumer. With its asynchronicity comes the high possibility that at any point of time t the state of the system's participants divert, i.e. are inconsistent. This is, for example, the case when two applications of different consumer groups read from the same partition p but are at different offsets; semantically, then, both applications are inconsistent. However, they only are inconsistent with regard to a *fictitious* global system time. Their offset, in effect a position in the timeline constructed by the partition, mark their inconsistency; with regard to the partition's time, i.e. offset, they therefore are consistent. Globally, at any time T, the system will reach *eventual consistency*, which delineates the fact, that – when no further events are published into p – at some time $T + t, with\ t\ \in\ \mathbb{N}$, both applications will reach the same offset and therefore will be actual consistent. As the absence of further publications is fictitious with regard to the concept of a stream, *eventual consistency* may almost be best thought of as consistency "to come" [Law19], a continuous promise of consistency.

3.2.3 Kafka Streams

When Kafka is best understood as an event-database, Kafka Streams is best understood as an API making this data accessible in the form of derived, materialized views, i.e. an API for "'unbundling' the database" [Kle15] that Kafka is. Kafka Streams – a "simple" library – allows for filtering, mapping, aggregating and joining on a stream's message – one by one, not micro-batch [Kre16] – in all their variations. Given the storage mechanisms of Kafka and the derivable views on data with Kafka Streams, together they build a data structure as potent as a database.

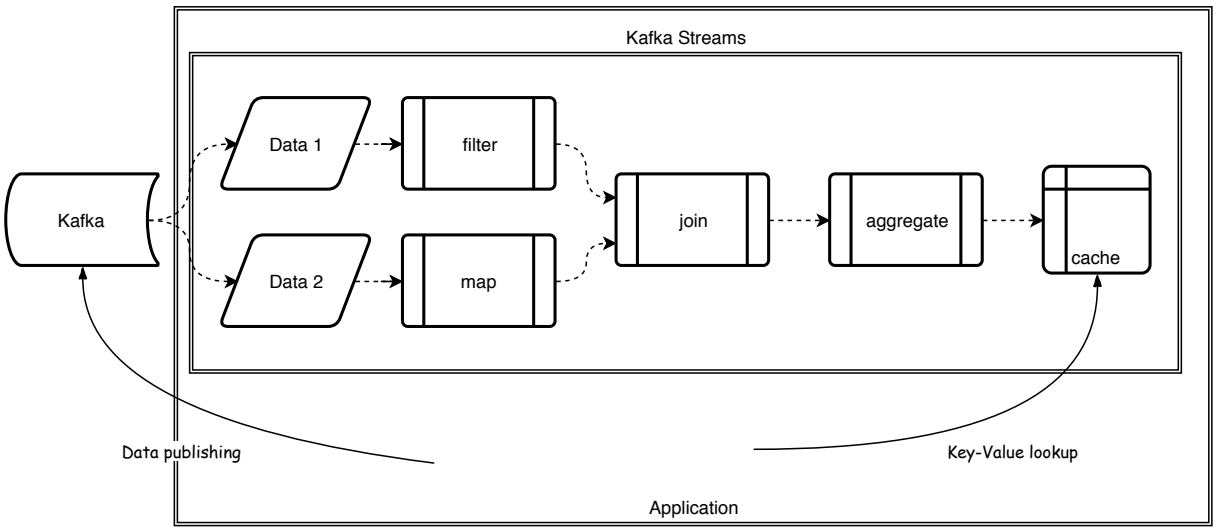

Figure 3.2: The standard application of Kafka Streams: filtering, mapping, aggregating, joining to make a specific view on data accessible to an application

Every stream is a set of logs, i.e. partitions, of data each representing internally a strict sequence in time. Streaming this data and for each message key keeping the last state in effect produces *eventually* the current state of a domain and all its objects; the set of last states equals therefore a state as it is representable in a relational database [Hel15] – this is the so-called *stream-table-duality*. This resulting *table* can be materialized with RocksDB (the default in Kafka Streams) as a KeyValue-Store. A "materialized view [as it is supplied via the Kafka Streams API] is *precomputed*" from the perspective of the application [Kle15, emphasis in original], it is the result of a declared function operating on immutable data supplied by a stream. As is implied, these functions can both be stateless and stateful,[13] and they can be understood as orchestrations of processors operating on the stream. From here it is only a short way to Kafka Stream's full potential. As streams can be *mapped*, a transforming function on each message, *filtered*, a predicate based selection of a stream's messages, *joined*, a mapping merge of two streams by matching message keys, and *aggregated*, a mapping compaction, no further databases or programmatic operations are needed to eventually access domains in their current state respecively derived views. Further, Kafka Streams allows for exactly-once guarantees.

Kafka Streams has two APIs to program against; the Kafka Streams DSL is a highler level descriptive language that suffices for most demands. The Processor API allows for a more fine-grained control to state stores. The DSL is built on top of the Processor API.

[13] "The notification and replication duality that events demonstrate maps cleanly to the concepts of stateless and stateful stream processing, respectively."[Sto18]

3.2.3.1 Excursion: Design Patterns

Stopford [Sto18] suggests that a system using Kafka and Kafka Streams as data storage and data retrieval is pointed by this usage to a set of abstract design patterns:

- "[L]atest-versioned pattern": In this pattern a compacted log of data with Kafka Streams is derived from a normal, non-compacted log holding source data. [Sto18]
- "[E]vent collaboration" pattern: This pattern is a broad architectural pattern that already was implied in the discussion on the κ-Architecture and is suggested by the declarative programming style. A system of applications connected via streams is "choreographie[d]" rather than orchestrated [Sto18]; here a series of events is a branching cascade of events where each event triggers next events. The process of processes is a DAG, a directed acyclical graph, of processors consuming and publishing at their own rate. Stopford suggests the assembly line as an analogy for this architectural design that the Log implies [Sto18].

3.2.4 Kafka's Ecosystem

With Kafka and Kafka Streams the main libraries and applications are exposed. However, since Kafka's initial development a range of further applications became part of what should be called Kafka's ecosystem. These applications target specific needs.

3.2.4.1 Kafka Connect

Kafka Connect is an application that allows for streaming data from existing databases into Kafka and from Kafka into databases, in handles data in and out of Kafka's ecosystem. In doing so it supports Kafka Streams strong guarantee of "exactly once" [Mof18]. Given Kafka's intention to replace other storage mechanisms and persist data itself, Kafka Connect, more specifically, is intended as a means of integration of legacy systems. Streaming data from databases by Kafka Connect standard connectors, e.g. Confluent's JDBC Connector, works by them polling the database regularly; this has at least three disadvantages:

1. Only positive states of the database are communicated into Kafka, especially delete-actions are invisible to such polls.
2. Polling the database regularly weighs heavily on the database's resources and as such on its performance.
3. Polling the database necessarily means having a delay t of $t < polling-interval$ between the actual database transaction and it publication into Kafka.

A variant of connectors hack into databases' transaction logs and misuses these for deriving a change log, which then includes delete-actions as well, published into Kafka [Mof18]. As a learning analytics engine working with legacy databases, like a database used by Moodle, surely intends to operate on delete-actions as well, a CDC connector is the only available technical solution – beside rewriting the software itself such that it publishes into Kafka. Fur-

thermore, this is a solution to the other two disadvantages, too, as the database no longer is polled to gather changes.

3.2.4.2 Schema registry

A schema registry is a server application that allows for storing and retrieving identifyable schemas, e.g. in case an Avro schema is used as the wired data format. Whereas the producer posts a new schema to the schema registry and publishes raw data, without meta information like fieldnames, together with a schema identifier, the consumer deserializes the message by retrieving the relevant schema from the schema registry. In the latter case, caching a relevant schema is possible such that retrieving requests are rare. The schema registry therefore is a central building block of a Kafka system towards a sparse traffic and disk space usage. Both Apache and Confluent offer available schema registries that are easily embeddeable.

3.2.4.3 KSQL

KSQL is beside the Streams DSL and the Processor API a third – probably the simplest and abstractest – way to program over Kafka's Streams. KSQL is a SQL-like syntax for queries and stream processings that runs in its own cluster; streams and views of streams *can* be made persistent. Developed by Confluent it aims at enabling non-developer to act upon streams in a simple manner; for this Confluent provides the KSQL-Cli. It might be especially useful for simple pre-processings of streams. An analytics platform that shall enable non-developers to execute queries on data in Kafka, for example, to reach experimental statistical insights, therefore should make KSQL available.

3.3 Reviewing the Case of the κ-Architecture

With the chapter's first section a broad concept of the κ- Architecture was explicated, the second section undertook a dive into the technology that Kreps had in mind suggesting the κ-Architecture: namely Kafka and Kafka Streams. Given this, the architecture's ability to fullfil outlined demands can be assessed newly.

3.3.1 On κ-Architecture, Kafka and Functional Norms

The κ-Architecture fullfils equal functions with regard to big data as the λ-Architecture. It scales horizontally backed by Kafka's partitioned logs, which scaling should indeed be linear: Partitioning, on which Kafka relies heavily, allows for parallelism. Further, Kafka's fundamentally asynchronous conception that allows for the parallel consumption of streams by different servicse as well as the streams' replication over several instances suggests strong guarantees with regard to availability, latency and throughput: All are questions of horizontal scaling alone. Kafka at least promises that operations on streams holding a full history are still performant.

As such a κ-Architecture with Kafka not only allows for real-time operations, but an equally well handling of big data's aspects.

With Kafka Streams any arbitrary operation on a set of data streams seems realizable as an incremental algorithm. However, the implementation comes partly at costs of high complexity (more later).

3.3.2 On κ-Architecture, Kafka and Operational Norms

As was set forth above, Kreps believes that the κ-Architecture fulfills the functional demands of an architecture for big data processing at least as well as the λ-Architecture while leaving behind the operational complexities of managing two types of systems and their interlocking. With regard to the outlined operational norms, however, the κ-Architecture not only simplifies *devops*, but allows further a strong separation of concern that the λ-Architecture when based on a tight system like Hadoop cannot allow for. The κ-Architecture consists of a multitude of applications that only are composed or choreographied, and that means, only are interdependent via streams of data. Streams and their data structures therefore operate as the sole *contract* between elements of the system; such system therefore lends itself naturally to a software development management where – at the top – one team is responsible for one application. The strong independence of applications operating on public streams of data translates into a possible independence of teams and their devops. This directly implies that the point of dependence is exactly the data structure communicated via a stream. This architectural feature is seamlessly supported by Kafka and its ecosystem, as Kafka Consumer, Kafka Producer and Kafka Streams are libraries useable in any application and their interdependence is situated at interfaces and communicative processes with Kafka broker. A choreographied series of processes suggests a devops highly agile.

3.3.3 On κ-Architecture, Kafka and Research

The absence of statically available data as well as the absence of random access to that data seem burdens with regard to requirements articulated by research interests. However, at least with KSQL an application is available that allows a simplified access to streams that is non-persistent. Experimentations are possible. With regard to ML, it seems that the last years brought a boost as well in ML based on streams; if so, Kafka will be supported well.

3.3.4 A Word of Caution: Doubting κ as Being Fit for Analytics

While these lines of argumentation seem to make conceptually a case in favor of the κ-Architecture, there is one thought that casts doubt on the applicability of the κ-Architecture for a learning analytics engine.

A learning analytics engine regularly has to engage in complex analytics that operates on a multitude of datasources, generates complex mappings, joins, aggregations and filters. Given a

traditional database, irrelevant of being a relational, broad column or document or indeed some other static data supplying database, such processings are unproblematic, indeed databases are data structures developed for exactly this use. In a streaming engine these datasources probably are published in different streams; a complex event processor therefore has to operate over a multitude of streams which is by definition a complex procedure given the time-based character of events. In such cases a streaming engine like Kafka produces a high complexity of stream reshuffling under the hood which has an impact both on resources and on the developer's understanding.

With regard to this thought, it might be worthwhile to consider that Kafka's and Kafka Stream's main area of application is the composition of micro-services, where Kafka is responsible for data accessibility and Kafka Streams operates as a filtering, joinging, mapping and aggregating entry point into data. But this is something other than complex analytics. It is for this reason that Narkhede (one of the core developers of Kafka and Kafka Streams at the company Confluent) states (in 2016) in a comparison of Kafka Streams and Apache Flink:

> "The gap the Streams API fills is less the analytics-focused domain and more building core applications and microservices that process data streams." [Nar16][14]

It seems not sensible to expect a learning analytics engine to be needed to produce *mainly* real-time analytics.[15] Rather it seems probable that most of the interests regarding data has no strong time concern. Further, it is conceptually indeterminable how to weigh the complexity involved in stream processing against the complexity of a simultaneity of batch and stream processing. But it seems intuitable that the greater the DAG of processing nodes becomes the higher the complexity becomes. Cascades of stream processing applications likely become confusing with regards to estimated orders of time across streams, something for which a learning analytics engine probably cannot not have some regard, since it has direct impact on global consistency at any point of time. If that should be true, the above doubt becomes stronger: Why not, then, apply a kind of λ-Architecture or – indeed – just implement a parallelity of batch and stream processing, depending on the local functional demands, both sinking data in a distributed database?[16]

[14] Kreps stated a similar sentiment [Kre16].

[15] For a "systematic" decision model – based on supplied fault tolerances, message semantics, and latencies – on which stream processing to choose, see Dendane et al. [Den+19].

[16] Apache Flink may be a valid alternative to Kafka Streams in that case: it operates intimately with YARN, can consume with strong guarantees from Kafka and sink processed data into HDFSs. Further, it allows for both writing batch and stream processings. Apache Druid may also be worth looking into.

4 A Very Short Excursion: The Problems of Anonymization, Pseudonomizations and Trust – Given a Log

Given the possibly personal and sensitive data involved, the interest in personalized analytics *and* the applicability of the GDPR, demands of anonymization and pseudonomization need to be kept in mind. These pose inherently a problem to any Log-centered approach insofar as any log is composed of *immutable* data. Data in the log is persisted and is not supposed to be changed; any changed data is supposed to be attached to the end of the log thereby representing a newer state. While any demand to apply transformations to the current or a future data (e.g. anonymizations or pseudonomizations) is unproblematic as they can simply be applied to data at entrypoint, the demand to apply transformations *ex post* to data, i.e. after the publication, and to all instances of that data, i.e. globally, all streams downwards, is inherently adverse to the Log – it equals the rewriting of history. At time of writing the following possibilities seem to be considered to be able to deal with the "right to be forgotten":

1. Compacted-log approach: If all relevant data, i.e. data which needs to be able to be "forgotten", is held in a *compacted* log, the datum to be forgotten can be overwritten with an appropriate replacement by its publication under the same message key [Sto17]. This has the disadvantage that every relevant stream needs to be compacted – potentially a rather extreme loss, given that Kafka's value stems from exactly being ably to be a Log, i.e. hold a history of changes.
2. Masking approach: All relevant data can be pre-processed by a masking processor that replaces personal data with a foreign key which points to an object in a compacted stream of personal or sensitive data that can be cleaned according to the first approach. At the system's access points that personal or sensitive data could be merged back into the streams of evaluated data.
3. Encryption approach: All relevant data can be pre-processed in an encryption stream processor such that only encrypted data is persisted by Kafka. The encryption key can be held in a compacted log, and become forgotten such that the original datum is irretrievable [Leb18a; Leb18b; Han17]. This has the advantage of allowing immutable data to be held in every log, i.e. a full history is at least formally present. However, it has the disadvantage of hitting heavily on the system's performance as each processing of a message has to retrieve the encryption key, decrypt the message and encrypt the output to be published again.

However, even if that right to be forgotten is appropriately implementable, a trusted learning analytics has a higher self-obligation with regard to users' configurability of access to their data than the first and the third approach can fulfill. A trusted learning analytics should allow for users to apply a fine-grained configuration of the availability of CRUD operations on their data. This implies a higher complexity as it demands for a structured approach to existing, formerly "immutable" data. What this stronger position demands for is something like an encrypted object to which access is enabled and allowed via an *access control stream* that holds something like access control lists [Ale+].

It obviously is the case that the GDPR and its norms are at odds with immutable data and as such with some of the features that make Kafka potent. Given the complexity of the issue, in this thesis it will be ignored beyond this excursion.

5 The System's Conventions

Given that a decision for a specific architectural design was made, and given that Kafka operates as this architecture's distribution- and persistence-mechanism, the next questions concern the internals of Kafka, or more exactly conventions concerning these internals: The aim of developing a Log-based architecture for learning analytics demands for the development of a unifying organization of incoming streams of data that are heterogeneous in structure, volume and meaning. In other words, a meaningful abstraction over a multitude of unbounded sequences of data is to be found, where meaning is given to objects of streams by their potential analytical value – assuming the learning analytics context. While the κ-Architecture is simple enough and the heavy load of abstraction is done by Kafka, several aspects are to be considered, among them norms concerning the construction of streams, data formats and structures used for communication between participants as well as naming patterns. These different aspects intermingle in the following somewhat, as interdependent aspects they are not categorically separable. Broadly two rules apply:

1. Systemwide a single, common data format is to be decided for in advance [Kre15b].
2. The naming of topics should reflect at least the event type such that developer can derive it broadly [Kre15b].

What follows is a set of considerations and suggestions that will be adhered to in this thesis' implementation part. However, as conventions these can and *indeed* should evolve with further experimentation and implementation cycles.

5.1 Data Format

While there are a couple of possibilities to choose from, two data formats lend themselves for Kafka.

1. Avro: Avro is the *de facto* standard of Kafka. It is a highly compact data format in the sense that one only transfers actual values, together with a schema-identifier that is used for deserialization. As was put forward above, the Kafka ecosystem offers schema registries that allow for efficiently retrieving as well as updating schemas.
2. JSON: JSON is a simple, easily handeable format which has the advantage of being widely known, widely supported as well as being human readable. Further, JSON serializer and deserializer are relatively easy to implement. However, JSON serializes metadata like field information together with actual values. This inflates disk usage and traffic heavily. The exemplary learning analytics engine implemented in this thesis nonetheless

will use JSON – for reasons of simplicity. A final implementation should definitely use Avro; if this were a system going to be released, the thesis fell short of standards by using JSON.

5.2 Data Structures: Pros and Cons of XApi

While it is possible, und not directly unreasonable, to stream events of heterogeneous data structures under one topic, it may be desirable to impose a unifying data structure on these events and abstract over the original source. This may be the case as the data structures published represent the contract between producers and consumers. This could go to the extreme of imposing a single data structure on the whole system. However, several considerations have to be accounted for, least of which are neither the resulting size of transformed data nor the costs in terms of latency or throughput.

XApi is such a data structure that allows for unifying all kinds of events as experiences;[1] here an event is represented as a subject-verb-object (-context) structure. While XApi seems to crystallize as a *de facto* standard in the context of learning analytics, XApi seems mainly to be modelled towards the ends of cross-application conveyability and human readability. Further, XApi is modelled such that it best is persisted in a document database. As human readability is *not* of concern to data structures in Kafka and Kafka Streams as data structures here represent only a contract between applications[2], the question of XApi or not rests

1. on the question of its quality as a low-latency, low-size, etc. contract.
2. on the costs of transforming each source-stream into a XApi structure.
3. on the existence of reusable software operating on XApi structure.

At least regarding the first point XApi fails; its adaptability is bought by allowing many optional field – which necessarily hits upon performance when using Avro as data format. Furthermore, the full information, e.g. concerning an agent, is not for every processor or processing step of relevance; rather than dragging superfluous data around, if an operational interest appears, the relevant streams holding that data are joined. Given a static data supplier this is of no concern, given streams the necessary completeness of XApi objects heavily inflates needed resources. Concerning costs of transformation, this thesis will stay agonistic; while compared to here implemented indicators the transformation of original data structures into XApi statements is indeed rather *costly*, this might become negligible in future implementation. The existence of reusable software is unknown to the author. Accordingly, this thesis has no strong opinion on the question of used data structures, however a strong tendency towards the rejection of XApi seems to lend itself, given especially above first observation. Nonetheless, this thesis' implementation transforms original sources in an intermediate step into XApi statements.

5.3 Topic Organization

[1] According to some, to be xApi-conformant, an LRS has to have access to all statements and these have to be in JSON [Joh+17]; that norm is conceptually impossible given above architectural decisions.

[2] Human readability can be dealed with in a separated cluster of processors offering end-points for external services.

5.3.1 Broad Horizontal Division

In general, there seem to be broadly two approaches to divide the learning analytics engine's streams semantically.

1. A *source-centered* approach would organize logs according to their sources, e.g. given streams from Moodle database relations and streams from a lecture-attendance app, topic names would reflect their origin. This naive approach fails as an *abstraction* over sources – which is one central feature of Kafka as a data distribution platform. In effect it constructs instances of streams that hold mergeable data structures and necessitates unnecessary multiple stream consumptions. However, in concise and constrained contexts it might be a viable option (more later).
2. When meaning is attributed to streamed objects by their potential analytical value, then the organizing structure should reflect analytical core concept, i.e. should reflect an *analysis- or state-centered* approach. Starting point for the organization of streams can and should therefore be the analytical distinction of the micro-, meso and macro-level [Shu12]. [3] This distinction has well known meaning to developers of a learning analytics platform and divides meaningfully the stream horizontally.

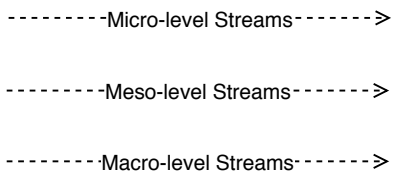

Figure 5.2: Macro-, Meso-, Micro-Streams

[3] A conservative attribution of a stream to an analytical level in favor of the micro- over the meso, and the meso- over the macro-level should be applied. This seems sensible as aggregations over streams of the micro- respectively the meso-level can be events in streams of the meso- respectively macro-level.

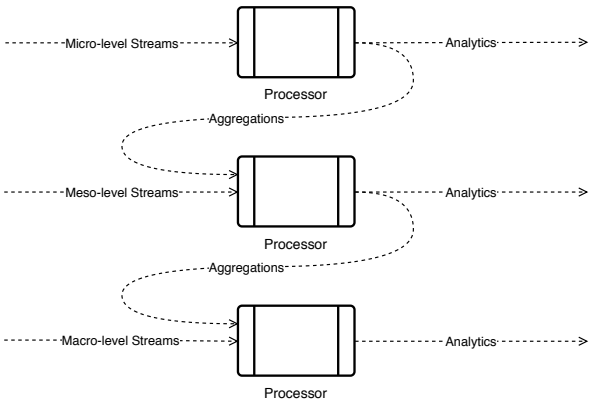

Figure 5.1: Macro-, Meso-, Micro-Streams with Horizontal Consumption

5.3.2 Naming Pattern

Kafka topic names allow for ASCII alphanumerics, '.', '_' and '-'; however, Kafka warns that topic names with '.' and '_' can collide. Furthermore, any naming should reflect at least the following requirements: It should

1. be readable and meaningful for a human being.
2. reflect current processing states as well as held data.
3. be simple.

Usually such pattern reflects a hierarchy of attribution, from left – the most general attributable meaning – to right – the most concrete. In the following the '.' separates attributes, while the '-' functions simply as a sign making longer names readable. The '_' will have a special purpose, explained below.

A first broad reflection on the processing state suggests that name patterns differentiate streams that are part of the Learning Analytics Engine from those that are not. Input streams start being part of the Learning Analytics Engine when they are pre-processed, which in this thesis' case would mean, when they are transformed into a unifying data structure such as XApi.

5.3.2.1 Pre-Processed Streams' Naming Pattern

Inasmuch as pre-processed streams have – from the perspective of the system – no processing state and the most important information about them is that they hold data from some external source, the *source-centered* approach should be applied. Furthermore, it seems advisable to prefix these streams to identify them as input streams – as which they are of central concern for radical replayability. The following pattern emerges:

$$\begin{aligned} &< input\text{-}stream\text{-}prefix > . < source\text{-}root\text{-}namespace > . \\ &< source\text{-}root > . < source > \end{aligned} \tag{5.1}$$

with a context-free grammar for source-root that captures an arbitrary depth:

$$\begin{aligned} &< source\text{-}root > \ \rightarrow \ string \\ &< source\text{-}root > \ \rightarrow < source\text{-}root > . < source\text{-}root > \end{aligned} \tag{5.2}$$

Defining the input-stream-prefix as '__' and given an input stream from the mdl_forum_posts relations of the Moodle database of the Goethe University of Frankfurt, the topic could look like

$$\_\_.goethe\text{-}university\text{-}frankfurt.db.moodle.mdl\_forum\_posts$$

This naming pattern is somewhat restricted by the configurations Kafka Connect connectors allow for. Above "mdl_forum_posts", as it actually is in the implementation, is a default naming

defined by the connector, it is derived from the database-relation's name and no simple way to explicitely define it was found.

If that stream is transformed into an XApi statement, information about the source is – or definitely should be – embedded explicitely in the data, as the statement's actor is attributable, in this example, to a database holding moodle data of the Goethe University of Frankfurt. Streams internal to the learning analytics engine therefore no longer have to reflect that original source, indeed, they should not.

5.3.2.2 Processed Streams' Naming Pattern

Internal to the learning analytics engine all streams are unified by a common data structure, e.g. XApi. Further, streams can be distinguished by their attribution to the micro-, meso- or macro-level. Finally, it is to be kept in mind, that

1. processors could make necessary streams representing different states of processing.
2. an orthogonal dependency may exist, i.e. processors stream to other processors, e.g. from a micro- to a meso-processor.

The following represents a first suggestion.

- For data streams that are only pre-processed:

$$< Learning\text{-}Analytics\text{-}Engine\text{-}prefix > . \\ < analysis\text{-}level > . < stream\text{-}data > \tag{5.3}$$

- For data streams that are used internal to a processor-cluster, representing different states:

$$< processor\text{-}cluster\text{-}prefix > . \\ < Learning\text{-}Analytics\text{-}Engine\text{-}prefix > . < analysis\text{-}level > . \\ < stream\text{-}data > . < stream\text{-}indicator\text{-}data > \\ \_ < processor\text{-}state > \tag{5.4}$$

  with a context-free grammar for processor-state that captures an arbitrary depth:

$$< processor\text{-}state > \ \rightarrow \ string \\ < processor\text{-}state > \ \rightarrow < processor\text{-}state > . < processor\text{-}state > \tag{5.5}$$

- For resulting streams of indicators:

$$< Learning\text{-}Analytics\text{-}Engine\text{-}prefix > . < analysis\text{-}level > . \\ < stream\text{-}indicator\text{-}data > \tag{5.6}$$

This alone does not clarify a lot; it still is unclear, what the meaning of $< stream\text{-}data >$ respectively $< stream\text{-}indicator\text{-}data >$ is. One first intuition would suggest to produce streams per XApi verb, another one – more in line with the initial analytical levels – to produce streams

per actor-type, such as student, teacher, lecture attendancees etc. However, both are non-optimal: XApi verbs are too numeric[4]. While the sheer number of streams might be unproblematic if Kafka can be scaled accordingly, such number of streams becomes counterintuitive. Further, it rather seems sensible to divide streams that become too packed at a later stage and let the system evolve incrementally. The second variant would enforce probably a clustering too coarse. The suggestion supported here is the following: a *clustering over verbs* should characterize streams such that for example a stream $discussion$ captures all statements with verbs like "posted" or "replied".[5] This clustering can both reflect semantic proximity of verbs as well as shared analytical interests. It should be kept in mind, even if given by XApi, that a common data structure for a single topic is best practice, as otherwise deserialization costs due to the retrieval of schema information increase.

Defining the Learning-Analytics-prefix as 'edutec' and Processor-Cluster-Prefix to as a single '_', the above Moodle data example results in the following:

- A pre-processed mdl-forum-posts stream:

$$edutec.micro.discussion$$

- A processor-internal streams, e.g. for the case of reply-time per user where a stream for the reply-time per discussion and a stream for the aggregated reply-time over all discussion would be needed:

$$\_.edutec.micro.discussion.reply\text{-}time\_per\text{-}discussion$$

$$\_.edutec.micro.discussion.reply\text{-}time\_per\text{-}discussion.aggregated$$

- The resulting stream:

$$edutec.micro.reply\text{-}time$$

Finally, given the need for replayability, i.e. the need for a parallel release of a new version, all streams should be suffixed with a version number:

$$< streamname > \_ < versionnumber >$$

[4] A public repository of verbs [XAp] already holds more than 500.

[5] Maybe this cluster should reflext XApi in the way that the name reflects an experience respectively an action, maybe instead of $discussion$ it should be $discussed$, $stated$ or $articulated$.

6 Implementation

6.1 Programming Environment

6.1.1 Major Technologies

The technologies used in the learning analytics engine are briefly presented below:

- **Angular 8:** Angular 8 is the latest of Angular versions. Angular is a library which allows for writing web-based applications in Typescript which are compiled into Javascript. It delivers an injection framework, composeable groups of html-templates and typescript files, etc.
- **Docker and Docker-Compose:** Docker is a software that allows for containerized applications; each application can be build into an image that has a stack of necessary (system) libraries build into it. Docker-compose is a tool for orchestrating dockerized application in a file.
- **Javascript** Javascript is a high-level general-purpose programming language. It is used here to write a lightweight adapter that consumes from LeapMotion's websocket and publishes that data into Kafka.
- **Kafka:** See section 3.2.2.
- **Kafka Connect with CDC Connector:** See section 3.2.4.1.
- **Kafka Streams:** See section 3.2.3.
- **MySQL 5:** MySQL is a standard relational database.
- **NgRx:** Ngrx is a library implementing a Redux-like state management for Angular applications. It allows for implementing a central data store on which pure functions triggered by actions or effects of actions execute changes. As data is stored as Observables, components can subscribe to these changes.
- **Python 3:** Pyton is a high-level general-purpose programming language. It is used here for scripts which publish mock data into the moodle database. These scripts are used as well for performance tests applied to the learning analytics engine.
- **RocksDB:** RocksDB is a fault-tolerant key-value-store used by Kafka Streams by default. It caches the key-value records and, if cache begins to overflow, persists entries to disc.

- **Spring Boot:** Spring Boot is an opinionated implementation of Spring. At the center of Spring stands an injection framework that allows for a fast and clean implementation of server applications written in Java.
- **Spring Cloud Stream** Spring Cloud Stream is an opinionated abstraction for asynchronous, message driven applications. It abstracts with so-called binders over some existing message brokers like Kafka or RabbitMQ and facilitates the consumption and publication to message brokers with configuration-file based opinionated default-implementations.
- **WebSocket:** WebSocket is a communication protocoll operating in layer 7 of the OSI model; it calls into standard HTTP ports. Websockets allow for bidirectional communication between endpoints as well as a push-like model where data from the server is communicated via the socket into the endpoint.

Some specific remarks on when, where and why some of these technologies are applied might be necessary.

1. The author started the implementation process with Spring Cloud Stream and hit repeatedly walls: one problem with opinionated abstractions is the increasing invisibility and the higher demand to the developer's understanding. The implementation process took up speed and success only once Spring Cloud Stream was abolished – which happened far too late – and the Spring Boot applications implemented consumption from and production to Kafka with the Spring Kafka and Apache Kafka Streams libraries. Indeed, the simplification of Spring Cloud Stream sits mainly in the abstraction over the message broker, however, since Kafka was already decided for in this project, this abstraction only produces a software overhead and a complication to understanding.
2. There are several docker images for Moodle available; the selection was more or less arbitrary, leaving out the decision for a MySQL database. This decision for a MySQL database was mainly driven by the availability of a CDC connector.

6.2 Project Structure

The project structure is an instance of the general structure for a κ-Architecture as it was set forth above. From a general point of view the architecture look roughly like figure 6.1.

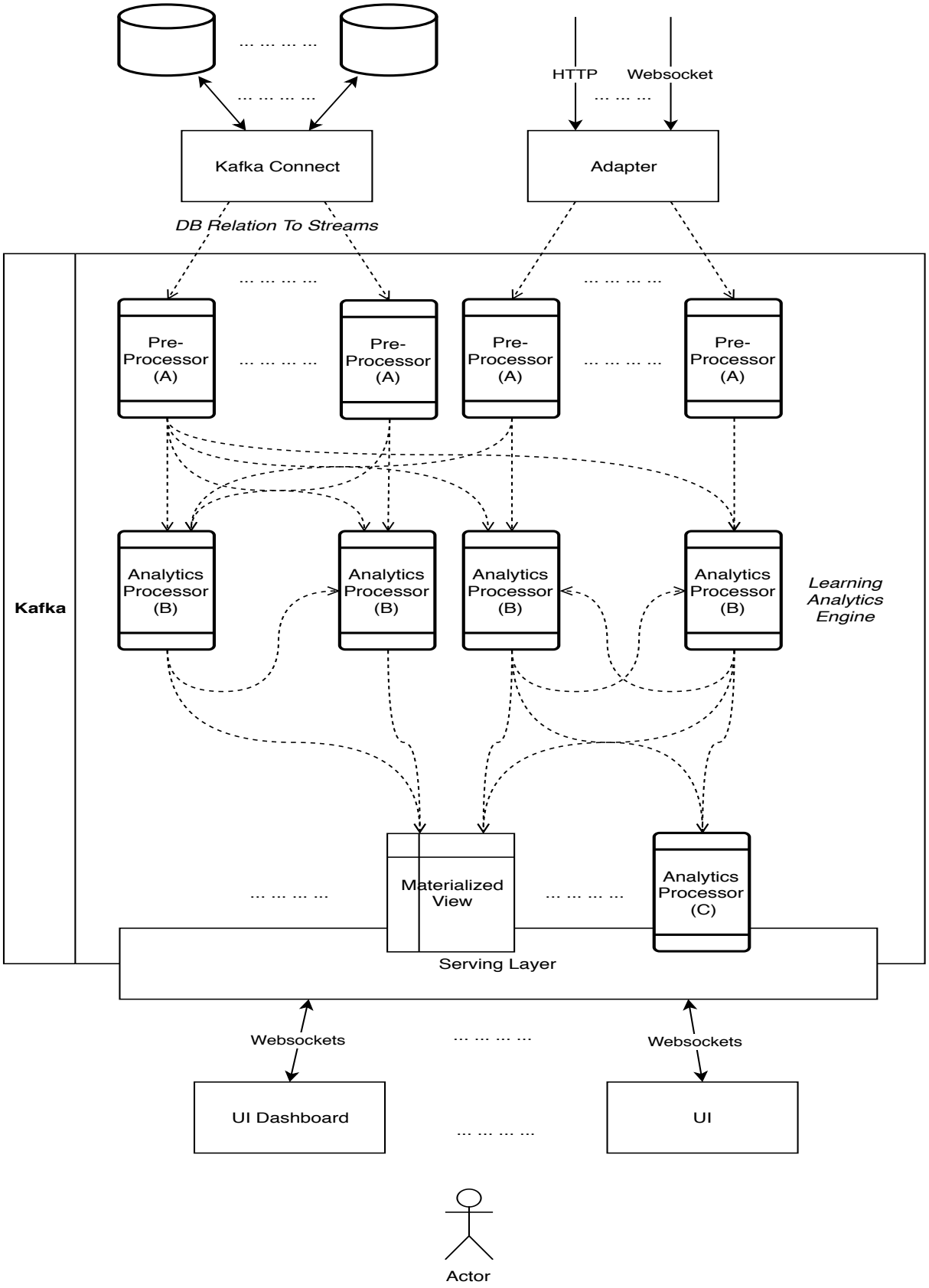

Figure 6.1: Architectural Overview

More concretely, the docker-compose file starts the following applications resp. containers:

- **indicator-ui**
- **moodle-xapi-transformator**
- **discussion-evaluator**
- **assessment-evaluator**
- **serving**
- **mysql-db**
- **moodle**
- **connect-debezium**
- **zookeeper**
- **kafka**
- **control-center**
- LeapMotionToKafka is a non-dockerized pure Javascript application.

The implemented system, a concrete instance of the above architectural overview in figure 6.1, looks like figure 6.2.

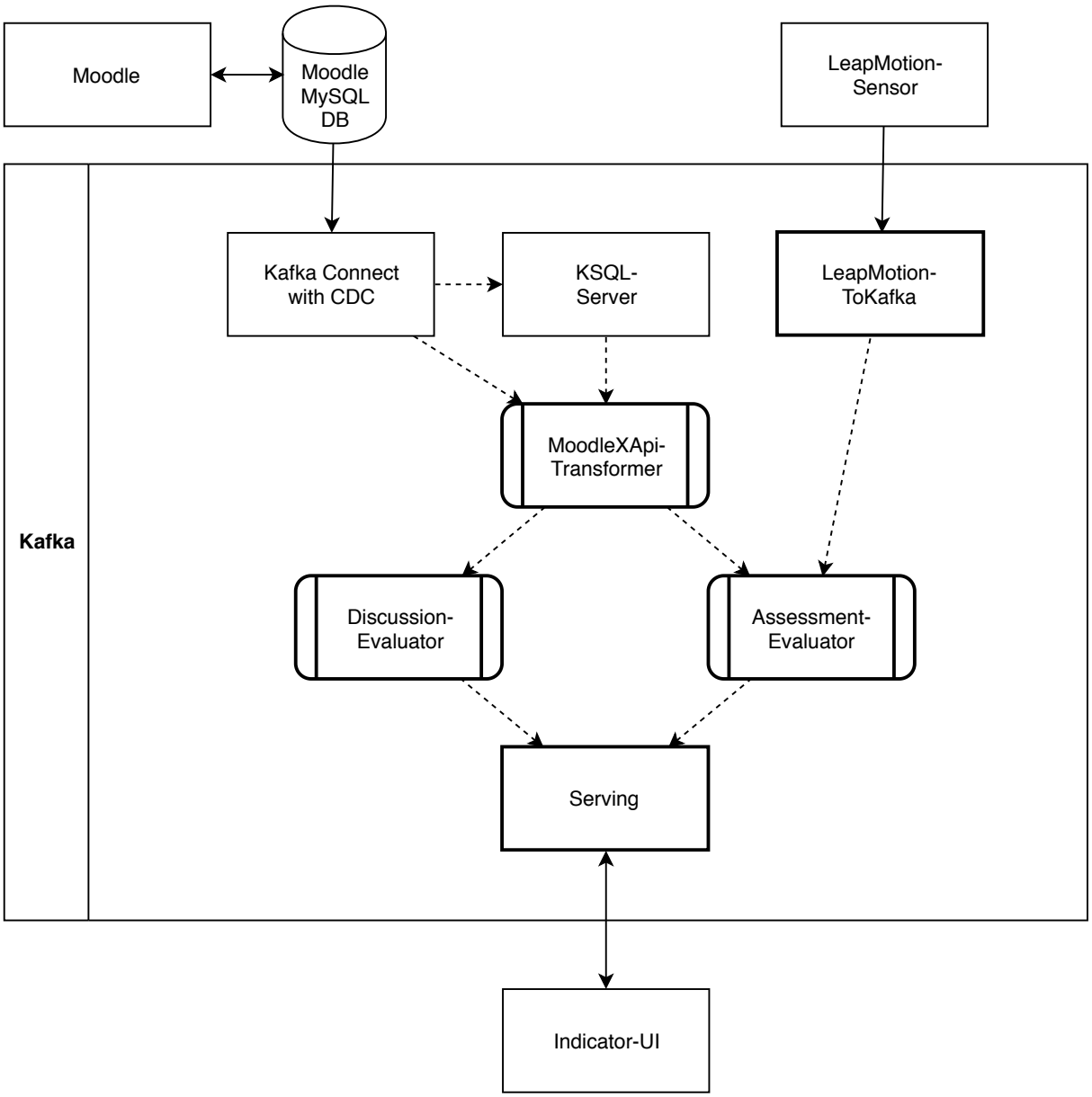

Figure 6.2: Architectural Implementation

6.3 Application Workflow

The application's workflow consists mainly of two processes. The first, see figure 6.3, describes the flow of data from the Moodle database's relations mdl_forum, mdl_forum_post and mdl_user via Kafka Connect into an application that transforms the data into XApi statements. These XApi statements are then consumed by another application that calculates some simple statistics, which are published and consumed by an application in the serving layer. A UI connects and reads data via a websocket.

The second process, see figure 6.4, describes the flow from a LeapMotion sensor, whose websocket is consumed and published into Kafka. Further Moodle database relations, namely mdl_quiz and mdl_quiz_attempt are streamed via Kafka Connect; the process is here the same as above. However, instead of preprocessing them purely in a Java application, they are preprocessed as well by persisted KSQL-Operations, which also use a defined function written in Java. Published XApi statements are window-joined in an application consuming the leap motion frames as well, producing simple statistics. Again, these are consumed by an application in the serving layer and send via websocket to the UI.

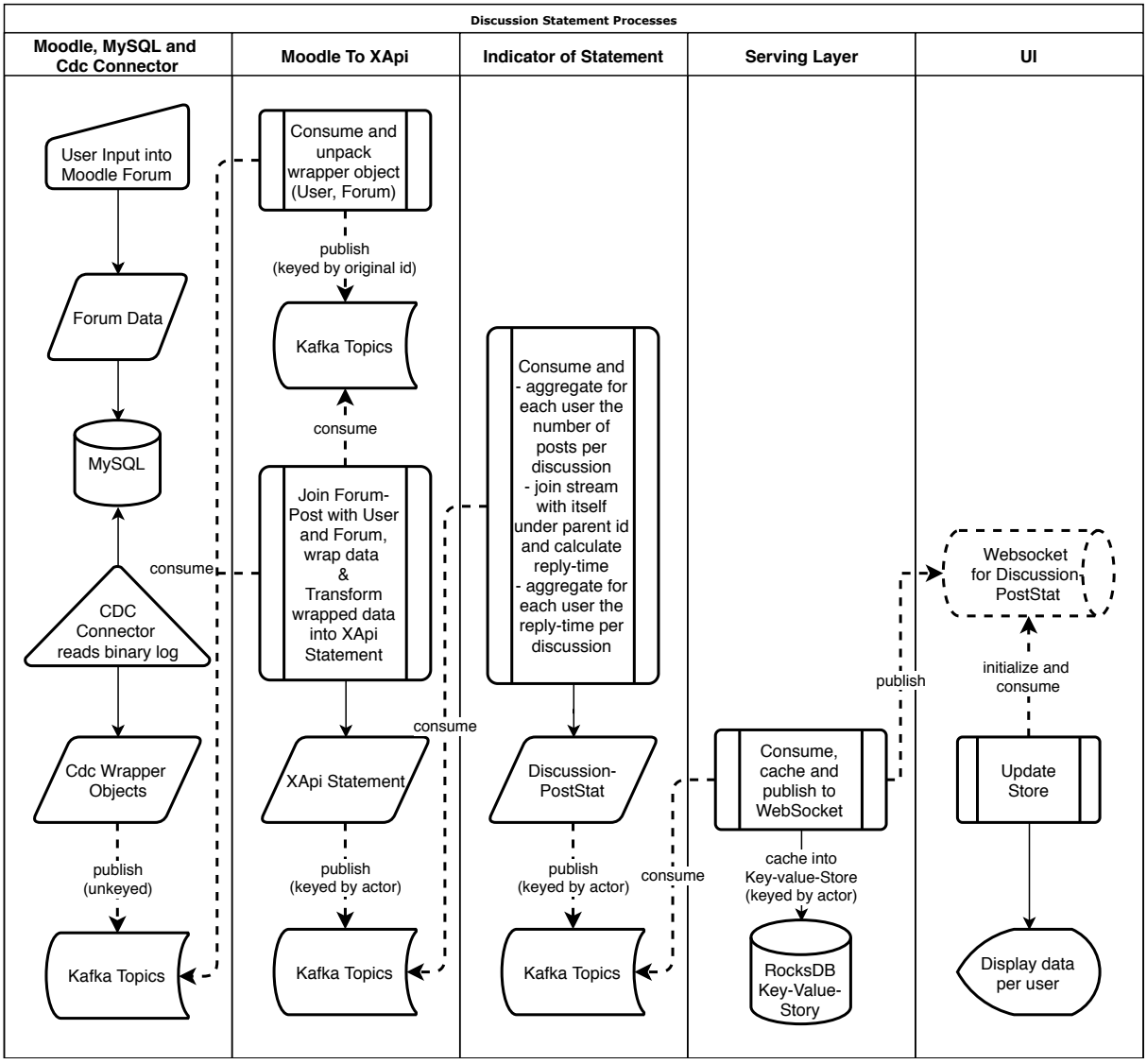

Figure 6.3: Data Flow

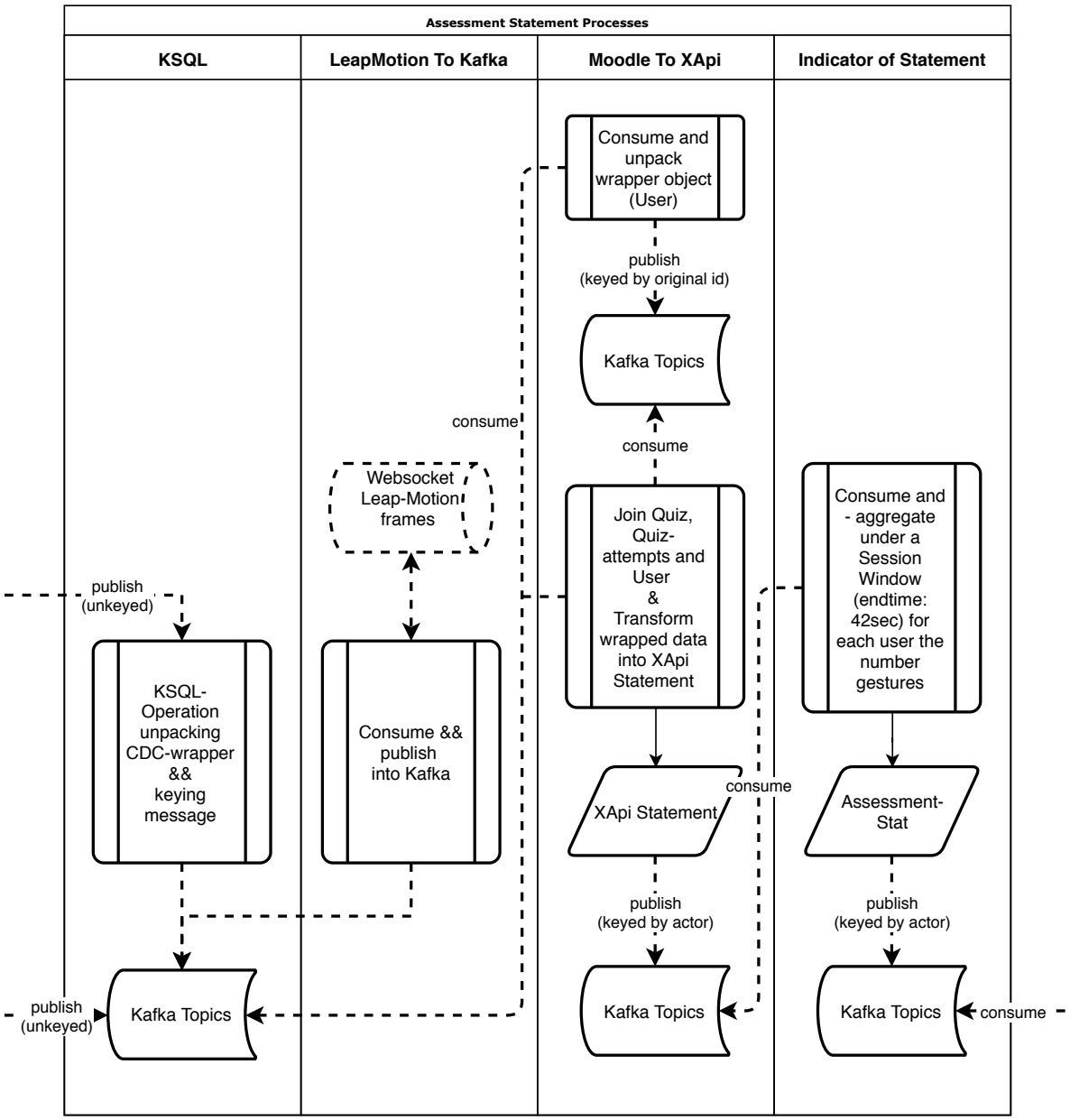

Figure 6.4: Data Flow 2

6.3.1 MySQL Database

The MySQL database is initialized with its binary log enabled per row. An initial user, which is used by the Kafka connector, is inserted, too.

6.3.2 Kafka Connect with Debezium's CDC MySQL Connector

The Kafka Connect server application with the Debezium CDC MySQL Connector allows for starting a connector which is configured the following way:

```
curl -X POST connect-debezium:8083/connectors -H "Content-Type: application/json" -d '{
  "name": "moodle-connector",
  "config": {
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "database.hostname": "mysql-db",
    "database.port": "3306",
    "database.user": "debezium",
    "database.password": "dbz",
    "database.server.id":"77",
    "database.server.name": "__.goethe-universitaet-frankfurt.db",
    "table.whitelist": "moodle.mdl\_forum,moodle.mdl\_forum\_posts,moodle.mdl\_user,moodle.mdl\_quiz,moodle.mdl\_quiz\_attempts,moodle.mdl\_quiz\_feedback,moodle.mdl\_quiz\_grades",
    "database.history.kafka.bootstrap.servers": "kafka:29092",
    "database.history.kafka.topic": "__.goethe-universitaet-frankfurt.db.moodle.cdc.schema.changes",
    "include.schema.changes": "false",
    "value.converter":"org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable":"false",
    "key.converter":"org.apache.kafka.connect.json.JsonConverter",
    "key.converter.schemas.enable":"false"
  }
}'
```

Listing 6.1: The defined connector

That connector will read the MySQL's binary log and publish data from the whitelisted relations into topics with a naming pattern:

$$< database.server.name > . < database.name > . < relation >$$

6.4 Implemented Applications

6.4.1 MoodleXApiTransfomer

The application started as container *moodle-xapi-transformer* is a Spring Boot application that consumes data from kafka topics as they are published by the CDC Connector to Moodle's MySQL database. Relevant consumption topics are:

$$\_\_.goethe - universitaet - frankfurt.db.moodle.mdl\_forum$$

A topic holding CDC wrapper objects that represent a transactional change, i.e. the object's former state, the new state and the source where the object is a representation of a Moodle-Forum as it is persisted as a relation in Moodle's database. Messages are published without being explicitely keyed, i.e. they are being published round-robin.

$$\_\_.goethe-universitaet-frankfurt.db.moodle.mdl\_user$$

A topic holding CDC wrapper objects that represent a transactional change, i.e. the object's former state, the new state and the source where the object is a representation of a Moodle-User as it is persisted as a relation in Moodle's database. Messages are published without being explicitely keyed, i.e. they are being published round-robin.

$$\_\_.goethe-universitaet-frankfurt.db.moodle.mdl\_forum\_posts$$

A topic holding CDC wrapper objects that represent a transactional change, i.e. the object's former state, the new state and the source where the object is a representation of a Moodle-Forum-Post as it is persisted as a relation in Moodle's database. Messages are published without being explicitely keyed, i.e. they are being published round-robin.

The MoodleXApiTransformer, written with Kafka Streams DSL, consumes streams of Moodle-User, -Forum and -Forum-Post and unpacks the CDC wrapper object. The Forum-Post object holds original foreign keys to a Forum and a User; the application joins together these streams based on the original foreign keys and gathers the relevant objects in a wrapper object. Finally, this wrapper is transformed into a XApi Statement, based on its data, which then is published to the topic with the actor's name as message key:

$$edutec.micro.discussion$$

As was above explained, this topic gathers data that fit into being part of a discussion, e.g. "user writes a threadopening forummessage", "user comments on a forum message", "user authors a blog post", "user comments on a blog post" etc [STD16].

Further, the application consumes Moodle-Quizes and -Quiz-attempts. These are preprocessed by a KSQL server. Unluckily, the server forbids '.' in the topic names and makes them automatically uppercase.

$$\_\_GOETHEUNIVERSITAETFRANKFURT\_MDLQUIZATTEMPTS\_UNPACKED\_KEYED\_BY\_USER \tag{6.1}$$

A topic holding unpacked Moodle-Quiz-attempts; the stream is keyed with the id of the user who is attempting the quiz.

$$\_\_GOETHEUNIVERSITAETFRANKFURT\_MDLQUIZ\_UNPACKED\_KEYED$$

A topic holding unpacked Moodle-Quizes keyed with their original id.

Both topics are joined together with the Moodle-User stream into a wrapper object, which finally is transformed into a XApi statement, which in turn is published to the topic with the user's name as message key:

$$edutec.micro.assessment$$

As was above explained, this topic gathers data that fit into being part of an assessment.

6.4.2 Discussion-Evaluator

The application started as container *discussion-evaluator* is a Spring Boot application that consumes data from kafka topics and evaluates these streams. Relevant consumption topics are:

$$edutec.micro.discussion$$

See above.

The discussion-evaluator consumes XApi Statements and produces simple indicators, such as the statistics representing the number of posts per user, the number of posts per user per discussion, the reply time per user and the reply time per user per discussion. The resulting analytics object, that exists then for each actor having authored a Statement, is published under the actor's id to the topic:

$$discussion - analytics$$

Adhering to the outlined conventions, this topic gathers analytics result on individual learners or participants with regard to *discussions*. Further, this topic is not being compacted, as itself functions in a complete learning analytics engine as an input stream to processors that target e.g. meso-level analytics for which a history of changes might be of interest.

6.4.3 LeapMotionToKafka

The application LeapMotionToKafka is a Javascript application based on node packages; it is started locally as it is a simple adapter that consumes from Leap-Motion's local websocket frames and publishes these into a Kafka topic:

$$\_\_.goethe - universitaet - frankfurt.myo.frames$$

It is assumed that this publication uses as message key the user's id as it is identifyable in the Moodle database. This is fictituous, but necessary to be able to combine both data sources.

6.4.4 AssessmentEvaluator

The application started as container *assessment-evaluator* is a Spring Boot application that consumes data from kafka topics and evaluates these streams: It counts occurences of gestures in a session window, i.e. as long as there is no gap of arriving leap motion frames longer than, in the concrete case, 42 seconds, all gestures are aggregated into one result. If a pause in the stream occurs that is longer than 42 seconds, a new session starts and the resulting values keyed for the user are overwritten.

Relevant consumption topics are:

$$\_\_.goethe - universitaet - frankfurt.myo.frames$$

See above.

$$edutec.micro.assessment$$

See above.

The discussion-evaluator consumes XApi Statements derived from streams from the Moodle database as well as the stream that holds LeapMotion frames. It session-window-joins these two different kind of streams together and produces some kind of statistical measure which is published under the actor's id to the topic:

$$assessment - analytics$$

6.4.5 Serving Layer

The application started as container *serving* is a Spring Boot application whose sole purpose is to make analytics results available to multiple UI's via REST-interfaces and WebSocket. Relevant consumption topics are:

$$discussion - analytics$$

See above.

$$assessment - analytics$$

See above.

The application consumes from these streams, caches the data to a RocksDB and publishes it via a Websocket to any connected user interface application.

6.4.6 Indicator-UI

The application started as container *indicator-ui* is an Angular 8 application which connects per websocket to the serving layer. It represents all analytical data.

7 Evaluation

7.1 User Guide

7.1.1 Initial startup

Installations Install docker, docker-compose, node.js, leap-motion driver. For Mockdata production, install via pip pymysql.[1]

Start containers Start the containers. To do so 'cd root/software/docker', where the docker-compose file is located.

```
1  docker-compose up -d mysql-db
2
```

Listing 7.1: Start MySQL container

This will start a containerized MySQL dabase whose binary log is enabled and for which a user with admin rights is inserted, which the Kafka Connect connector impersonates. Further, a moodle installation setup is dumped and loaded on boot; user login is: 'admin', password: 'Admin_2019'.[2]

```
1  docker-compose up -d moodle kafka zookeeper connect-debezium indicator-ui
   moodle-xapi-transformer discussion-evaluator assessment-evaluator ksql-
   server
2
```

Listing 7.2: Start other containers. The Spring Boot applications are configured such that they automatically create necessary topics in Kafka. A Java application's jar is mounted into the KSQL-server as well a KSQL-file that operates on some streams.

```
1  docker-compose exec connect-debezium /bin/bash
2  ls /scripts
3
```

Listing 7.3: Do optional check: Into the Kafka-Connect's docker container a folder with scripts is mounted; to check if a create-mysql-connector.sh script is listed.

```
1  docker-compose exec connect-debezium bash -c '/scripts/create-mysql-
   connector.sh'
```

[1] This guide assumes that the user is operating on a Linux system and has access to root.
[2] To start a completely fresh Moodle instance, remove 'root/software/kafka-connect/init-db/x_backup.sql'.

```
2
```

Listing 7.4: Start the Kafka Connect CDC connector.

See output The containers' start takes a while. After that: Browse 'localhost:4300/user-statistics'

7.1.2 Some admin Commands for Kafka Connect

```
curl <server>:<Kafka Connect debezium port>/connectors/
```

Listing 7.5: List available connectors

```
curl -X DELETE <server>:<Kafka Connect debezium port>/connectors/<connector name>
```

Listing 7.6: Delete connector

7.1.3 Some admin Commands for the Kafka broker

After Kafka broker and Zookeeper have been started, relevant topics need to be created. While in this setup, the applications are creating their topics themselves, in a producion environment this may be unwanted. Any topic can be created inside any running Kafka broker container:

```
docker-compose exec <broker container name> kafka-topics --create
--topic <topic name> --bootstrap-server <broker container
name>:<port> --partitions <#partitions> --replication-factor
<#replications>
```

Listing 7.7: Create topic

```
docker-compose exec <broker container name> kafka-topics --zookeeper
<zookeeper container name> --list
```

Listing 7.8: List topics

To consume in the console from an existing topic from the earlist offset on:

```
docker-compose exec <broker container name> kafka-console-consumer
--bootstrap-server <broker container name>:<port> --topic <topic
name> --from-beginning
```

Listing 7.9: Console consumer

7.2 Test Results

All applications consuming from Kafka streams are tested by unit tests. Beyond these, click tests and measured scripts publishing data with known results were executed.

In his general naiveté regarding the actual speed of todays computers and in his specific naiveté regarding the actual speed of Kafka, this thesis' author initially planned to execute performance tests regarding end-to-end latency on the simple processing pipelines he implemented. When finalizing the implementation he experimentally inserted incrementally 100.000,

500.000 and 1.700.000 Moodle Forum-Posts in the MySQL-database with a Python script on a single thread. These Forum-Posts' timestamps were compared with each result's timestamp: on the scale of seconds there is no measurable difference between the first and the 1.700.000th Forum-Post regarding end-to-end latency. Indeed, on the scale of seconds there is no measurable end-to-end latency. And to anyone better affiliated with Kafka and Kafka streams this should come at no surprise [Kre14a].

8 Conclusion

8.1 Research Questions Revisited

The research questions outlined in the introduction were handled throughout the text. With regard to the architectural questions posed the κ- Architecture was described as a simplification of the λ-Architecture. This simplification promises to fullfil demands of a system that is able to process big data in real-time without externalizing complexities into processes of development and operation. The κ-Architecture should allow for a horizontal scaling accross a multitude of servers that makes the system able to handle data with high availability, low latency and high throughput. Albeit, as was made clear above, it does so on costs of global consistency – according to the CAP-Theorem necessarily so – and increased conceptual complexity: each partition of a stream may supply a coherent history of objects, neither across partitions nor accross consumers of the same partition can consistency be guaranteed. The resulting *eventual consistency* may pose as no great problem concerning a multitude of microservices each handling a more or less decoupled domain, however, given a DAG of processing nodes that together shall build a learning analytics engine, consistent downstream availability may become a concern. Transactional guarantees across streams may become necessary, if an order to the arrival or presence of a upstream-processed results is presupposed – as is e.g. in Stream-Table joins where an update of the right side does not trigger a message downstream but only an update of the internal value-store. The absence of guarantees across partitions and streams enables parallelity and therefore scaleability, but given complex event processing – as it is demanded by a learning analytics engine – and, consequentially, the multitude of joins or lookups in other streams and tables this architectural setup becomes cumbersome as each join and lookup necessitates a new repartitioning of the stream. While it may be true, that this repartitioning is more or less fully done by Kafka itself, such that programmers can concentrate fully on business logics, it still produces a complexity – albeit a hidden one. This complexity not at last hits on resources demanded by the system, which given Kafka's performance may be ignorable. However, it may be for this reason that both Kreps and Narkede suggested that Kafka Streams is build for microservices rather than for analytics.

While this internal architectural setup of Kafka, being the κ-Architecture's backbone, may doubt the architecture's usefulness as a learning analytics engine, operational requirements as well as requirements derived from the statistics aspect of big data seem well supported by a κ-Architecture based on Kafka. With Kafka Streams' being a library useable in any application, this setup allows for a highly agile, multi-team devops; each team could develop and operate a set of applications being embedded in a DAG of processors consuming from and publishing

to public streams of data. With regard to data mining this thesis stayed agnostic. But it seems likely that, given the market position of Kafka as a streaming platform, if interfaces for ML exist or will be build, they will allow for operating on Kafka's streams and tables. Finally, with KSQL and its CLI a tool is provided that allows non-developers to experiment on streams of data and gain research insights.

With regard to conventions this thesis suggested a range of naming patterns that aimed at facilitating the operational control over data and its flows. Considerations concerning data structures used inside the system suggested that the use of XApi is not a good idea. It not only inflates the size by its many optional fields, it also suggests to communicate objects that in some sense are complete with regard to an *experience*. This implies that an object holds excessive data from source onwards with regard to actual requirements of operations downstream in the DAG of processors – which indeed is contrary to the architectural idea as well as contrary to a sparse use of resources. Indeed, every data structure that implements Resource Description Frameworks or that demands for the fullfilment of norms defining the presence of data will encounter this problem. Given this problem of norms, it seems most sensible to model carefully for use cases given while having a concept of possible evolutions in mind.

All in all, a performant, distributable and fault-tolerant learning analytics engine seems realizable implementing a κ-Architecture based on Kafka and Kafka Streams applications. However, given above doubts a final evaluation may follow only after some further work will have been done.

8.2 Outlook

This thesis implemented a system that asynchronously consumes two heterogenuous data sources in form of Kafka-streams, applies some simple transformations as well as simple statistical measures to these streams and presents the results in a continuous view updated via websockets. The theoretical considerations together with the experience gathered during the implementation suggest some specific points for further research:

1. The κ-Architecture based on Kafka and Kafka Streams applications has advantages, and possible disadvantages. A similar system to the one implemented here, but using a different technology stack, e.g. Flink instead of Kafka or Druid instead of Kafka Streams applications, and indeed applying batch processing where senseful, would be a good for a more detailed assessment. Such would be a departure from the pure κ- and, indeed, rather follow a λ-Architecture. Such second way will pose its own problems, but may be better fitting to the usecase of a learning analytics engine. Given Kafka's performance, this may less be a question of performance, but one of devops.
2. The performance expectable from such learning analytics engine may be better to assess given complex *real*, as in actually used, indicators. These may become comparatively more complicated to those implemented here, such that the above statements are to be reviewed.
3. The question of internal data structures respectively data models remains largely unsolved. This thesis suggests – though not strongly – to reject XApi as a leading data model, and it assumes that a usecase-oriented modeling with possible evolutions in mind might be best. However, further experimentations might suggest otherwise. In any case,

given Kafka as a data storage mechanisms, data models can be evolutionized as well as revolutionized, a *final* decision therefore is not necessary, and indeed may be counter-productive.

4. The huge question of how to handle users' rights defined by the GDPR as well as sensible data in general, that was only outlined here, needs an answer. Some possible ways to go were mentioned. However, if they prove to be untenable or tenable only at high costs, this problem would be better accessible given a static storing mechanism rather than Kafka, as data provided statically could be managed – which in turn may be an argument in favor of non-κ-esk architecture. Further research here is necessary.
5. Both security and the system's distribution over several servers was completely ignored in this thesis and should be explored. Both should impose no serious problems, since Kafka is developed with distribution in mind as well as it supplies security mechanisms.

Source Code

[Sch] Tonio Schwind. *Source Code*. URL: https://github.com/toschio/bachelor-thesis (visited on 09/10/2019).

Bibliography

[AA19] Sinan AY and M. Ali Akçayol. “A Comprehensive Analysis of Architectures and Methods of Real-Time Big Data Analytics”. In: *Lecture Notes on Information Theory* 5.1 (2019), pp. 7–12. ISSN: 23013788. DOI: 10.18178/lnit.5.1.7-12.

[Ale+] Ben Alex et al. *Spring Security Documentation: Domain Object Security*. URL: https://docs.spring.io/spring-security/site/docs/5.0.0.RELEASE/reference/htmlsingle/{\#}domain-acls.

[Ber14] Pere Ferrera Bertran. *Lambda Architecture: A state-of-the-art*. 2014. URL: http://www.datasalt.com/2014/01/lambda-architecture-a-state-of-the-art/ (visited on 06/22/2019).

[Ber18] Tim Berglund. *Building Streaming Microservices with Apache Kafka*. 2018. URL: http://www.youtube.com/watch?v=Hlb-Ss3q3as.

[BRD16] Rajkumar Buyya, N. Calheiros Rodrigo, and Amir Vahid Dastjerdi. *Big Data. Principles and Paradigms*. Cambridge: Elsevier, 2016.

[Cona] Confluent. *Kafka Java Consumer*. URL: https://docs.confluent.io/current/clients/consumer.html (visited on 07/04/2019).

[Conb] Confluent. *Kafka Java Producer*. URL: https://docs.confluent.io/current/clients/producer.html (visited on 07/04/2019).

[CV14] Roberto Casati and Achille Varzi. *Stanford Encyclopedia of Philosophy, Epistemology*. Ed. by Edward N. Zalta. 2014. URL: https://plato.stanford.edu/entries/events/.

[Den+19] Youness Dendane et al. “A quality model for evaluating and choosing a stream processing framework architecture”. In: (2019), pp. 1–13. arXiv: 1901.09062. URL: http://arxiv.org/abs/1901.09062.

[DF16] Ted Dunning and Ellen Friedman. *Streaming architecture : new designs using Apache Kafka and MapR streams*. 2016. URL: http://proquest.tech.safaribooksonline.de/9781491953914.

[DP18] Miyuru Dayarathna and Srinath Perera. “Recent Advancements in Event Processing”. In: *ACM Computing Surveys* 51.2 (2018), pp. 1–36. ISSN: 03600300. DOI: 10.1145/3170432.

[Fel+15] Gernot Fels et al. “Technik”. In: *Praxishandbuch Big Data. Wirtschaft - Recht - Technik*. Ed. by Joachim Dorschel. Wiesbaden: Springer, 2015.

[Fow14] Martin Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[GGR16] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. "Data Stream Management: A Brave New World". In: *Data Stream Management*. Ed. by Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Heidelberg: Springer, 2016, pp. 1–13.

[Han17] Alex Hanway. *A deeper dive into GDPR: Right to be forgotten?* 2017. URL: https://blog.gemalto.com/security/2017/08/16/deeper-dive-into-gdpr-right-to-be-forgotten/ (visited on 08/12/2019).

[Hel15] Pat Helland. "Immutability Changes Everything". In: *7th Biennal Conference on Innovative Data Systems Research (CIDR 2015)*. Asilomar, California, USA, 2015. URL: cidrdb.org/cidr2015/Papers/CIDR15{_}Paper16.pdf.

[Joh+17] Andy Johnson et al. "Enabling Intelligent Tutoring System Tracking with the Experience Application Programming Interface (xAPI)". In: *Design Recommendations for Intelligent Tutoring Systems*. Ed. by R. Sottilare et al. 2017, pp. 41–45.

[Kat+19] Kasumi Kato et al. "Construction Scheme of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka". In: *EPiC Series in Computing* 58 (2019), pp. 368–377.

[KK15] Martin Kleppmann and Jay Kreps. "Kafka , Samza and the Unix Philosophy of Distributed Data". In: *IEEE Data Engineering Bulletin* 38.4 (2015), pp. 4–14.

[Kle15] Martin Kleppmann. *Turning the database inside-out with Apache Samza*. 2015. URL: https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/ (visited on 07/05/2019).

[Kre] Jay Kreps. *It's Okay To Store Data In Kafka*. URL: https://de.confluent.io/blog/okay-store-data-apache-kafka/ (visited on 07/07/2019).

[Kre13] Jay Kreps. *The Log: What every software engineer should know about real-time data' s unifying abstraction*. 2013. URL: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying (visited on 04/25/2019).

[Kre14a] Jay Kreps. *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*. 2014. URL: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[Kre14b] Jay Kreps. *I Heart Logs*. Sebastopol, CA: O'Reilly, 2014.

[Kre14c] Jay Kreps. *Questioning the Lambda Architecture - O'Reilly Media*. 2014. URL: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (visited on 06/02/2019).

[Kre15a] Jay Kreps. *Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform (Part 1)*. 2015. URL: https://de.confluent.io/blog/event-streaming-platform-1 (visited on 08/04/2019).

[Kre15b] Jay Kreps. *Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform (Part 2)*. 2015. URL: https://de.confluent.io/blog/event-streaming-platform-2 (visited on 08/04/2019).

[Kre16] Jay Kreps. *Introducing Kafka Streams: Stream Processing Made Simple*. 2016. URL: https://de.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/ (visited on 05/02/2019).

[Law19] Leonard Lawlor. *Jacques Derrida*. Ed. by Edwad N. Zalta. 2019.

[Leb18a] Daniel Lebrero. *Kafka, GDPR and Event Sourcing*. 2018. URL: https://danlebrero.com/2018/04/11/kafka-gdpr-event-sourcing/ (visited on 08/12/2019).

[Leb18b] Daniel Lebrero. *Kafka, GDPR and Event Sourcing - Implementation details*. 2018. URL: https://danlebrero.com/2018/04/11/kafka-gdpr-event-sourcing-implentation/ (visited on 08/12/2019).

[Mar11] Nathan Marz. *How to beat the CAP theorem*. 2011. URL: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html (visited on 06/02/2019).

[Mar15] Nathan Marz. *Big data: principles and best practices of scalable real-time data systems*. Shelter Island, NY: Manning, 2015. ISBN: 9781617290343. URL: http://proquest.tech.safaribooksonline.de/9781617290343.

[Mof18] Robin Moffat. *No More Silos : How to Integrate Your Databases with Apache Kafka and CDC*. 2018. URL: https://www.confluent.io/blog/no-more-silos-how-to-integrate-your-databases-with-apache-kafka-and-cdc.

[Nar16] Neha Narkhede. *Apache Flink and Apache Kafka Streams: a comparison and guideline for users*. 2016. URL: https://de.confluent.io/blog/apache-flink-apache-kafka-streams-comparison-guideline-users/.

[PRL16] Miguel Angel López Peña, Carlos Area Rua, and Sergio Segovia Lozoya. *A "Fast Data" Architecture: Dashboard for Anomalous Traffic Analysis in Data Networks*. 2016. DOI: 10.1109/ICDIM.2016.7829756.

[Sch+] Jan Schneider et al. *Adopting Trust in Learning Analytics Infrastructures*.

[Sch] Tonio Schwind. *Source Code*. URL: https://github.com/toschio/bachelor-thesis (visited on 09/10/2019).

[Shu12] Simon Buckingham Shum. "Learning Analytics". In: *Policy Brief* November (2012).

[STD16] M. Scheffel, S. Ternier, and H. Drachsler. *The Dutch xAPI Specification for Learning Activities (DSLA) – Overview*. 2016. URL: http://bit.ly/DutchXAPIspread.

[Sto17] Ben Stopford. *Handling GDPR with Apache Kafka: How does a log forget?* 2017. URL: https://www.confluent.io/blog/handling-gdpr-log-forget/.

[Sto18] Ben Stopford. *Designing Event-Driven Systems. Concepts and Patterns for Streaming Services with Apache Kafka*. Sebastopol: O'Reilly, 2018. ISBN: 9781492038221. DOI: 10.1016/B978-1-85617-751-1.00013-6. URL: https://linkinghub.elsevier.com/retrieve/pii/B9781856177511000136.

[VK18] Hubert Vogten and Rob Koper. "Towards Big Data in Education: The Case at the Open University of the Netherlands". In: *Frontiers of Cyberlearning* (2018), pp. 125–143. DOI: 10.1007/978-981-13-0650-1. URL: http://link.springer.com/10.1007/978-981-13-0650-1.

[VPZ17] Francesco Versaci, Luca Pireddu, and Gianluigi Zanetti. *Distributed stream processing for genomics pipelines*. 2017. URL: https://doi.org/10.7287/peerj.preprints.3338v1.

[XAp] XApi. *XApi Vocabulary*. URL: http://xapi.vocab.pub/verbs/index.html (visited on 08/21/2019).