Project Number 780251

D5.5 Event Publishing and Monitoring Architecture (Final Version)

Version 1.0
8 July 2020
Final

Public Distribution

University of York

Project Partners: **Alpha Bank, ATB, Centrum Wiskunde & Informatica, CLMS, Edge Hill University, GMV, OTE, SWAT.Engineering, The Open Group, University of L'Aquila, University of Namur, University of York, Volkswagen**

PROJECT PARTNER CONTACT INFORMATION

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L'Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

DOCUMENT CONTROL

Version	Status	Date
0.1	First draft	27/05/2020
0.2	First full draft	17/06/2020
0.3	Changes based on internal feedback	02/07/2020
0.4	Changes based on internal feedback	03/07/2020
1.0	Incorporated suggestions from internal reviewers	08/07/2020

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

TABLE OF LISTINGS

EXECUTIVE SUMMARY

This document presents the final version of the high-performance architecture developed for data analysis and monitoring in TYPHON polystores. The architecture offers facilities for the authorisation of data access and update events and for the extraction of analytics of interest. It builds on scalable and fault-tolerant technologies such as Apache Kafka [1] and Apache Flink [2].

In this final document we present the improvements made to the architecture since the version presented in the previous deliverable (D5.3) as a result of the feedback received in the mid-term project review and the continuous feedback by the consortium partners. We also explain in more detail aspects of the architecture that were only outlined in the previous deliverable.

More specifically, beyond the presentation of the data event publishing and monitoring architecture, this deliverable focuses on explaining in detail the event authorisation mechanism, it discusses security concerns raised during the interim review meeting while an overview of existing related work is also presented. It also presents a requirements coverage matrix with a description on how each requirement is supported by the developed architecture. A new generator of simulated database events is also presented. This new simulator can generate fast streams of database events and it does not rely on the ingestion of data stored in CSV files like its previous version. This is essential for evaluating the scalability of the proposed architecture using controlled experiments. It also helps with highlighting the difference of our approach in defining analytics based on database events and not on already persisted data (e.g., in CSV files or databases) as other Business Intelligence (BI) and analytics tools (e.g., Tableau [3]) are able to do. The experiments evaluating both the authorisation chain and the analytics facilities demonstrated the linear scalability of the proposed approach and even distribution of workload across the cluster.

1. Introduction

1.1 Overview

The analytics feature of TYPHON is based on Polystore-triggered events which are produced every time a TyphonQL command is executed. Beyond the possibility of producing analytics based on database events, the analytics component also offers a mechanism of blocking the execution of commands that do not meet user-defined criteria. In order to achieve both these features the Polystore generates two types of events. The first, called PreEvent throughout this document, holds information about a query that it is about to be executed. PreEvents are authorised using configurable application-specific logic. If the execution of a query is authorised, the TyphonQL engine will be notified and execute it. Otherwise, the query will be rejected and thus not executed against the polystore. After the successful execution of the query, a second type of event, called PostEvent, is generated. PostEvents contain information about which data entities were affected by the execution of the TyphonQL command and are consumed by the analytics infrastructure to extract useful insights relevant to the business needs.

As TyphonQL commands such as Delete and Update statements, change values of existing data (rows, documents, etc.) we introduce a change data capture (CDC) mechanism. Using this CDC mechanism, analytics developers are able to refer to previous values of the affected entities increasing the potential scenarios that can be implemented using the proposed analytics architecture to extract information of interest.

To accommodate the large number of pre/post events that polystore-backed applications are expected to generate in real-world scenarios, the proposed architecture is implemented on top of proven big-data-capable frameworks such as Apache Flink [4] and Apache Kafka [1]. Flink is used for distributing the processing/execution workload of analytics applications while Kafka stores and dispatches the generated events in a form of a distributed log. All the necessary infrastructure needed for the Apache Flink and Kafka components is instantiated automatically with the creation of a TYPHON polystore. In addition, our proposed architecture includes rich abstractions to allow analytics experts develop analytics applications with minimal effort.

1.2 Document Structure

The rest of the document is structured as follows. In Section 2 we summarise the events structure while in Section 3 we present the analytics infrastructure and the authorisation mechanism, and discuss relevant security concerns. In Section 4 we discuss the deployment capabilities of the analytics architecture using Docker and Kubernetes. In Section 5 we present the scalability evaluation of the proposed architecture using a custom-built e-commerce application event simulator. Finally, in Section 6 we conclude and present directions for future work.

2. Data Analytics Events Structure

This section summarises the data analytics events structure metamodel, which was initially proposed in D5.2. An evolved version is presented here highlighting the changes since the last version.

2.1 METAMODEL

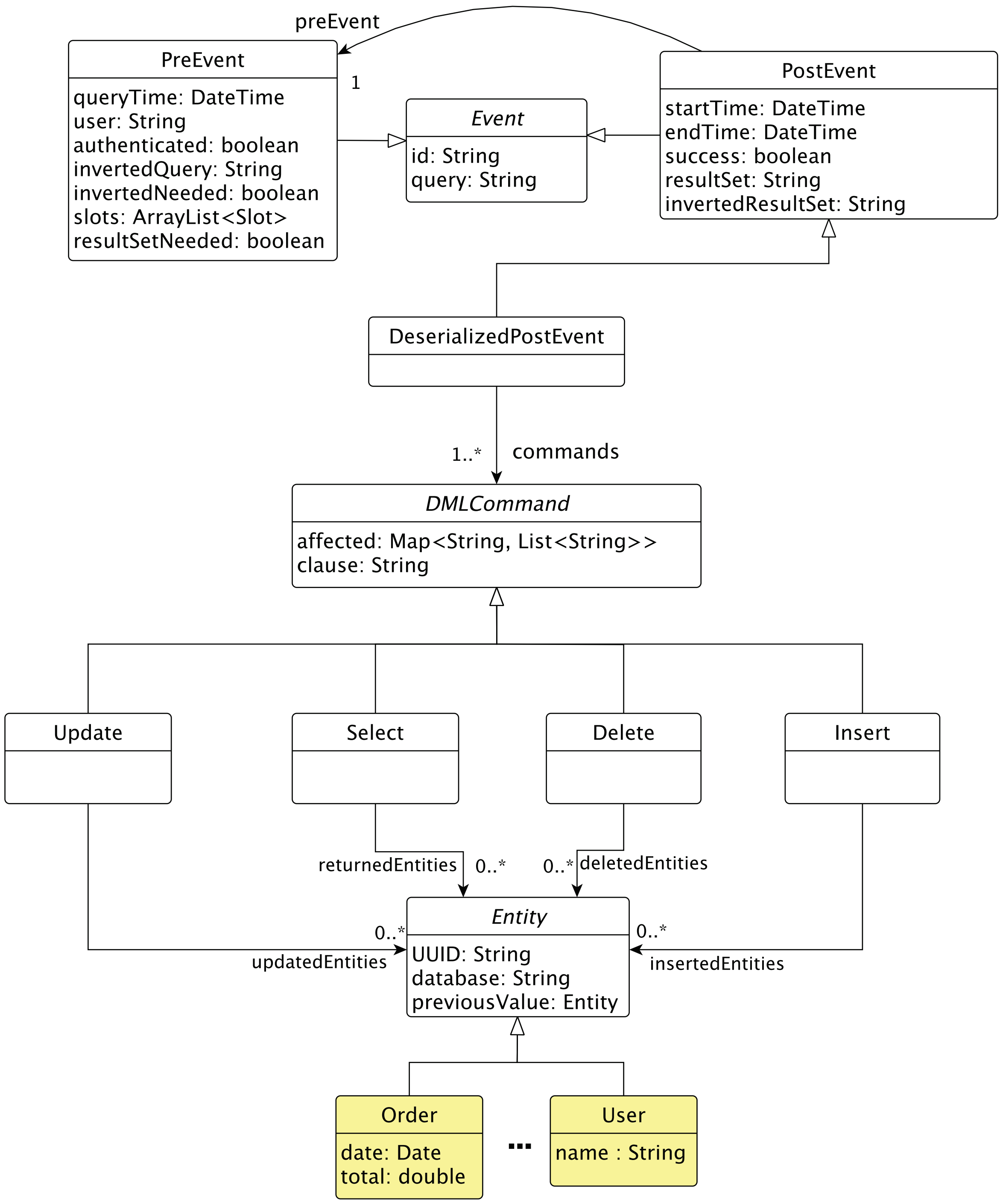


Figure 1: The Data Event Metamodel

Figure 1 presents the data analytics event structure metamodel. When a TyphonQL query arrives for execution to the TyphonQL engine a *PreEvent* object is created. It is of interest to allow developers to reject commands at the polystore level (as opposed to the application level). For example, NLP can be used to detect and block queries that are about to create comments/reviews that include profanity or unwanted URLs, regardless of the application that attempts to execute them. The information stored in this PreEvent object will be used as input to authorisation tasks defined by application developers for approving or rejecting the execution. If the event is approved, the command will be executed and a *PostEvent* will be generated by the TyphonQL engine.

Both PreEvents and PostEvents have a unique *id* and store the TyphonQL *query* that generated them. The time when the query arrived for execution to the polystore is stored in the *queryTime* attribute of PreEvents. A boolean flag, named *authenticated*, stores the result of the decision on if the query is approved for execution or rejected. Two more attributes, namely *invertedQuery* and *invertedNeeded*, facilitate the capture data change (CDC) mechanism supported by the analytics architecture (see Section 3.3). The *resultSetNeeded* boolean property declares if the the polystore needs to store the result of the execution of the commands in the PostEvent object after it is executed. The *slots* list acts as an extension mechanism to be utilised by polystore-backed applications. It hosts key-value pairs of any custom properties that analytics developers need to pass to the analytics workflow to accommodate their requirements when manipulating the events. A use of this feature is demonstrated in the scenario presented in the evaluation (Section 5.1.4)

PostEvent instances will point to their corresponding PreEvent instance (*preEvent* reference in the metamodel). PostEvents also hold timestamps of when the execution started (*startTime*) and when it ended (*endTime*). Query time and start time might be different; a command might have arrived for execution (queryTime) but TyphonQL might have started its execution (startTime) later due to a delay incurred by the time the authorisation task(s) for this query need(s). PostEvents store a *success* flag declaring whether the execution of the query by the TyphonQL engine was successful or not. Finally, the result set returned from the execution of the command against the Polystore is stored in the *resultSet* attribute. Also, the result set for the *invertedQuery* used for the CDC mechanism (see Section 3.3) is stored in the *invertedResultSet* attribute.

In order to simplify the implementation of analytics, the architecture offers access to strongly-typed objects. The analytics architecture consumes PostEvent objects and through an automated *deserialisation* approach creates such strongly typed objects. These objects, which are all instances of the *DeserialisedPostEvent* type referencing the type of the data manipulation language commands (*DMLCommand*) of TyphonQL query through the *commands* reference. Each DMLCommand[1] will store a list of the entities that were affected (e.g., the specific table in a relational database) in the field *affected*. As some queries only affect specific "fields" of the data structure (e.g., columns in a relational database) the affected property is a Map in which the key is the name of the data structure (e.g., the name of the table) pointing to a list of all the fields (e.g., columns) that were affected. For example, in the following TyphonQL query "*from Product p select p.name where p.@id =#123*" the *DMLCommand* will be of instance *Select,* while the *affected* field will be the map *Product->name*.

In addition, each DMLCommand will refer to the data entity that was manipulated (e.g., User, Order, etc.). The entities are automatically generated from information taken from the TyphonML domain model (highlighted in yellow in Figure 1). Depending on the type of the DMLCommand, the label of this reference changes (e.g., *InsertedEntities* for Insert DML commands). The *Entity* objects hold the Universally Unique Identifier (UUID) of the affected entity (e.g., the UUID assigned by TyphonQL when a new entity is created/inserted) and the name of the database that this entity is stored.

For example, for the query *"from Product p select p.name where p.@id = #123"* the DeserialisedPostEvent structure shown in Figure 2 is created.

[1] DMLCommands refer to the data manipulation language TyphonQL queries. The two terms are used interchangeably throughout this document.

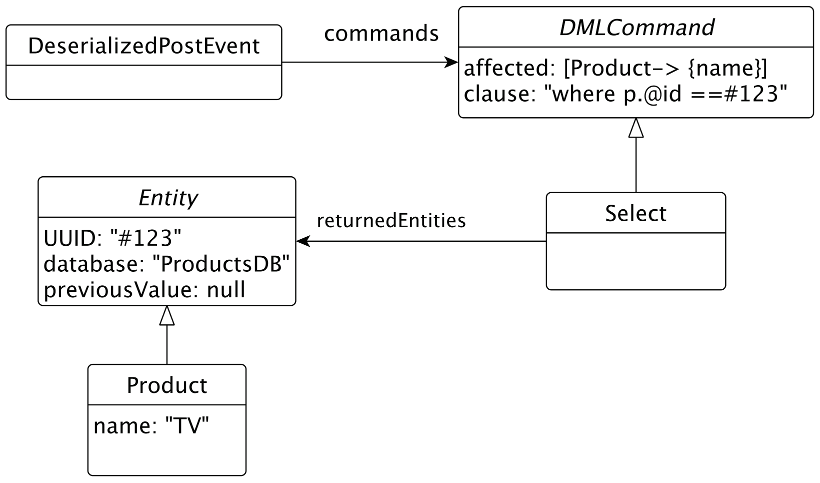


Figure 2: The structure of the deserialised post event for the query "from Product p select p.name where p.@id == #123"

In the case of Update/Delete commands, and in order to keep track of the old values of the affected entities (part of the CDC mechanism), an attribute called *previousValue* is used to host such information. The CDC mechanism is explained in detail in Section 3.4.

3. Data Event Publishing and Processing Architecture

This section presents an overview of the developed polystore data event publishing and processing architecture. The high-level architecture was introduced in D5.2/5.3 and it is summarised here. In this document we focus on the event authorisation part of the architecture (Section 3.2), the facilities that provide strongly typed objects to the analytics developers and a change capture mechanism (discussed in Section 3.3); which are new features developed since D5.3.

3.1 Overview

The high-level architecture for polystore events analytics in TYPHON is illustrated in Figure 3.

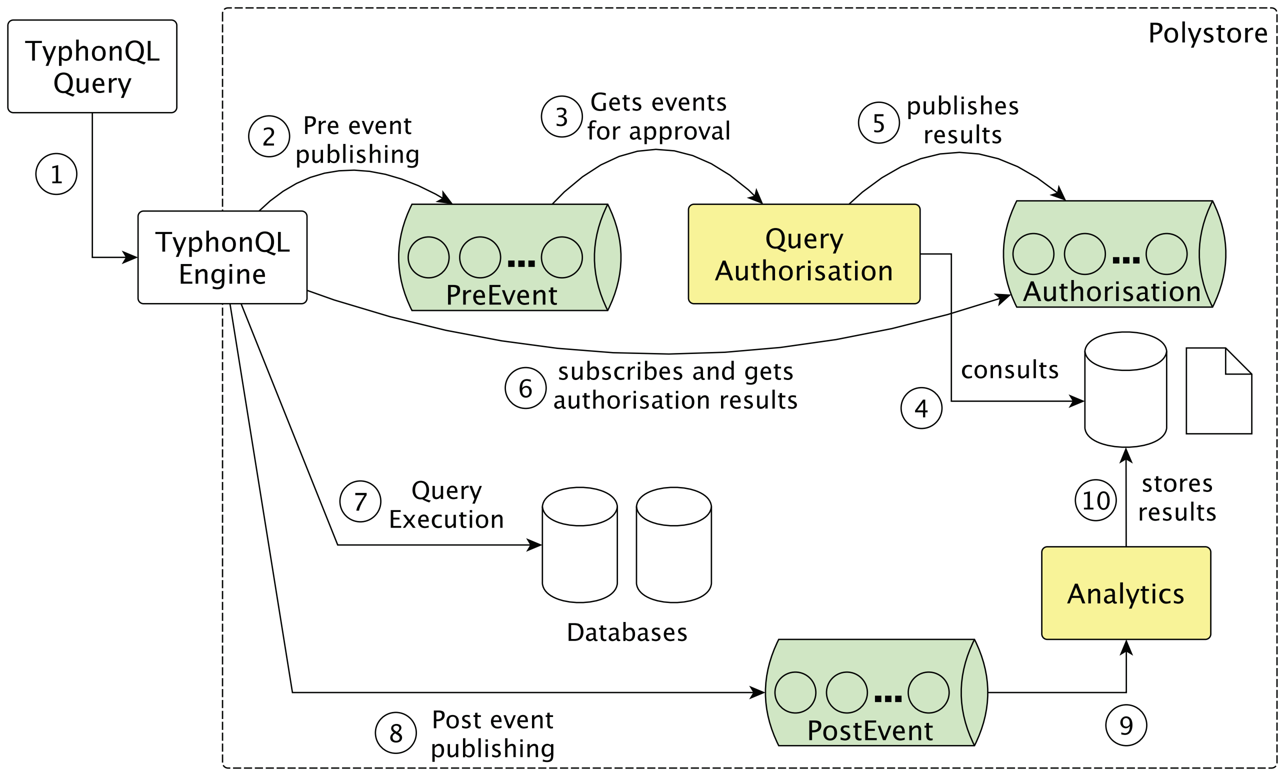


Figure 3: The proposed architecture for polystore event-based analytics

Following Figure 3, events undergo ten stages that are distributed among two main interleaved phases; *authorization* and *analytics*. The authorization phase involves validating if a new incoming query will be allowed to be executed against the polystore, based on rules defined in the analytics suite. The queries arriving for execution will take the form of PreEvent objects as these were defined in the previous section. The rules defined in authorisation tasks can be based either on hardcoded conditions (e.g., value of a specific field is above a threshold) or on information extracted from the history of previous events processed through the stream processor. For example, application developers could specify that login queries should be rejected if there have been more than three unsuccessful attempts for the same account in the last 60 minutes in the polystore (regardless of the application that produced them). This ensures that any malicious or unintended activity that does not follow the polystore's business rules will be detected and rejected as the last example highlights. Such a checking mechanism would be possible to be achieved at the application layer. For example, the checking mechanism could be fired every time the login button is pressed and check the authorisation logic before submitting the command for execution to the database. An advantage though of using the authorisation in the form of validating/rejecting incoming queries is that many applications that connect to the same Polystore can reuse those authorisation tasks instead of having to implement and maintain the code in each application.

The second phase is the analytics, which involves the continuous consumption of event messages for analysis which are generated from the TyphonQL based on DML commands that were authorised and executed. The tasks developed as part of this stage will consume messages in the form of PostEvent as these were described before. Below are the stages an event will go through as it progresses within the proposed architecture.

1) In this stage, a query is passed by a user to the TyphonQL engine for parsing and execution.
2) TyphonQL will publish a pre-execution event (PreEvent) for the incoming query, push it to a pre-event queue and wait for the authorization decision of this event.
3) A stream processor dedicated to authorization, will consume messages from the authorization queue to apply the required authorization checks before generating an authorization decision of an event.
4) In addition to any rules/checks the authorization stream processor has to apply, it can also consult previously extracted analytics, if there is any linked information that could indicate malicious or abnormal activity in relation to the new incoming event.
5) Following the application of the required authorization checks and the consultation of any historical data of interest produced by the analytics tasks, the authorization stream processor publishes its authorization decision to an authorisation queue.
6) TyphonQL receives the authorization decision it was waiting for in step 2.
7) Based on the outcome of the authorization decision generated in step 6, TyphonQL will execute the query received at step 1 or reject it and produce an appropriate exception to its caller.
8) In case the query is executed by TyphonQL, a post-execution event (PostEvent) will be generated and pushed to the analytics queue.
9) The analytics stream processor will consume the (post) event, deserialise it to a DeserialisedPostEvent object to which the relevant analytics could be applied.
10) The results of the analytics can be stored/published using different mechanisms (e.g., a database, a filesystem, a web-service).

A detailed discussion on the implementation of the analytics architecture can be found in D5.3. In the following section we focus on the event authorisation infrastructure's concepts and implementation which has not been described in previous deliverables.

The main subsystems involved in the proposed architecture are:

- Messaging queues/logs (coloured in green in Figure 3): these operate as the communication middleware between different subsystems to manage the messages of events generated at each stage of the event flow to be consumed by the relevant subscriber. These logs are implemented using Apache Kafka [1].
- Stream processors (coloured in yellow in Figure 3): such processors are responsible for analysing executed queries and extracting information of interest. Our implementation builds on Apache Flink [4] for the development of event processors.

3.1.1 State-of-the-art

Apache Kafka is an open-source and horizontally scalable, high-throughput, durable, fault-tolerant publish/subscribe messaging queue originally developed at LinkedIn [5]. Apache Kafka is used by many major technology firms such as Uber, Spotify, Expedia and others. In Kafka, one queue can be split into multiple *topics* to separate messages that belong to different categories/feeds. Compared to RabbitMQ [6] and ActiveMQ [7], Kafka offers a higher throughput per topic especially when the message routing logic is simple as in our architecture [8]. It also offers long term storage meaning that the same event can be consumed multiple times (replay of messages), acting more like a message log than a routing queue, making it ideal for offline analytics [8].

Apache Flink is a distributed processing engine that supports both *bounded* (batch) and *unbounded* (stream) data flows. Flink is designed to run streaming applications at any scale [9]. Application developed in Flink can be parallelized into any number of tasks which can be executed in a distributable and concurrent manner offering in practice unlimited scalability. Compared with Apache Spark [10], where streaming environment is based on micro-batching (i.e., incoming events are collected in small batches and sent for processing together), Flink offers native streaming capabilities, i.e., events are sent for processing individually, the moment they arrive. This leads to minimal latency, which is desirable in cases such as the authorisation one described in this document, where an event should be processed as soon as possible to avoid delaying the execution of other commands. However, it comes at the cost of harder implementation of fault tolerance as *checkpoints* for which events have arrived for processing should be kept individually for each event and not for a whole micro-batch. Flink performs very similarly to Apache Storm [11] in terms of execution time, however the latter (Storm) lacks important features like windowing, event time processing and aggregation. Except for the aforementioned advantages of Flink compared to its competitors, Flink is becoming a widely adopted parallel processing framework as it is being used in big technology firms such as Uber, Lyft, Alibaba, Yelp and others [2] which use it to process trillions of events every day [12].

Different systems [13] have been proposed and developed to capture database related evens to mostly allow replication or migration of databases. Connectors (e.g., KafkaConnect [14]) are registered to databases' specific mechanisms to extract already stored data and identify changes. These are published in message queues or logs (e.g., Apache Kafka). Sinks connected to these logs are then executing (replaying) the changes to store the data in another database system. Approaches like Maxwell's Deamon [15] and Oracle GoldenGate [16] monitor the database's log (i.e., binlog) to extract events but these are restricted to use only on relational databases. Debezium [17] offers an event capturing mechanism that supports both relational and non-relational databases. It builds atop KafkaConnect however, for some databases of interest (e.g., MongoDB, Cassandra), only capture changing commands (i.e., insert, update, delete) and not select queries. Also, Debezium supports a limited number of databases while it has to be deployed in each machine hosting a database (for example it has to be deployed separately in each node in a Cassandra cluster). Another overhead is that Debezium is not able to capture events in standalone MongoDB servers as these do not store logs of changes (i.e., oplogs). Thus, it requires deployment of MongoDB databases as MongoDB replica sets, which act as nodes sharing the *same* data two or more times. The Confluent platform [18] is a real-time event streaming application that uses Apache Kafka for storing events. It supports over 100 source and sink connectors to databases and filesystems [19] each of which support different level of granularity of the events that can be captured.

The aforementioned approaches support either a limited number/types of databases or limited amount of information from the databases supported. In addition, some of them require duplication of data or storing of unrelated events (e.g., the SQL binlog stores, except the DML commands, DDL commands, too). In addition, all of the approaches require the use (and development in case it is not available) of a custom connector for every database and database type in the system. Except the fact that such connectors might not be possible to be implemented of the database offers a related mechanism that can be exploited for that use, the different connectors can acquire different levels of information from each database type based on what information the database can offer. In addition, these connectors act separately in each database. If a single TyphonQL command affects more than one database, a common scenario in polystores, then the matching of these events coming from different databases but triggered from the same command is a difficult - if possible at all - task. Finally, to the best of our knowledge, none of the approaches offer a pre-execution event capturing and authorisation mechanism.

The latter can be achieved with the use of database triggers as those presented in D5.1. However, not all databases allow the execution of custom logic *before* the actual execution of the commands thus such a feature can only be used with some of the databases in the polystore. Most of the database systems allow the implementation of custom logic based on event triggers after the actual execution of the command. However, this comes with the drawback of having to define specific triggers for each type of command and table/document affected separately which does not allow the creation of a single event that contains all the information needed for the extraction of analytics if a single polystore command affects multiple tables/documents within the same database and across the different databases.

To conclude, our approach offers both a *before (PRE)* and *after (POST)* execution event capturing mechanism. Authorisation and analytics tasks can be developed having access to the data and databases the command affected, no matter if the latter had impact on multiple entities and different types of databases as it is based on the single database manipulation syntax (that of TyphonQL). Also, the latter allows future support of event capturing for any new database added to the polystore that is supported by polystore querying language without requiring developers to implement specific database event capturing/triggering mechanism which will require extra effort and is only possible if the newly added database offers a mechanism to offer the information of interest. Finally, in the case of migration of data from one type of database to another, the authorisation/analytics tasks do not need to be redeveloped to use the database specific event triggering syntax.

3.2 EVENT AUTHORISATION

3.2.1 Overview

The event authorisation architecture is based on the concept of *authorisation tasks*. Each authorisation task contains the business logic that defines if a query should be executed or not against the polystore. However, it is important to note that not all authorisation tasks are necessarily applicable to all queries arriving for execution. For example, in an ecommerce scenario, an authorisation task that contains the logic for approving/rejecting the registration of new users is only applicable to “insert” queries on the “user” entity. As a result, application developers need to provide, except of the approval/rejection logic, the logic that checks if the authorisation task is applicable to the incoming PreEvent (created for the query).

In the proposed authorisation architecture, all the tasks are part of an *authorisation chain*. A PreEvent arriving for authorisation, visits authorisation tasks one after the other, unless a previously visited task has already rejected the execution of that event. If a task has approved the execution of the query or is not responsible for checking the query it passes the PreEvent to the next task in the chain. A PreEvent is executed if it has been approved (or ignored) by all the tasks in the chain. In the following section we present the architecture and the implementation of the authorisation chain and tasks in detail.

3.2.2 Implementation

Developers can provide the aforementioned logic by implementing an abstract class (namely *GenericAuthorisationTask)* which is part of the analytics infrastructure. More specifically, they need to implement two methods for each authorisation task: i) the *checkCondition(Event event)* and ii) *shouldIReject(Event event)*.

The first method (i.e., *checkCondition(…)),* checks if the task is responsible for approving or rejecting a query. It returns a Boolean value which flags the result of the evaluation (i.e., true means that the task is responsible, false means that it is not). The second method (i.e., *shouldIReject(…)),* includes the logic the defines if a query should be approved or rejected. The *shouldIReject* method is called if and only if the *checkCondition* method of the task evaluated to true. The *shouldIReject* method returns a Boolean value stating whether the query should be rejected or not (true means that it should be rejected, false otherwise). Figure 4 shows the flow of an approved event through an example authorisation chain.

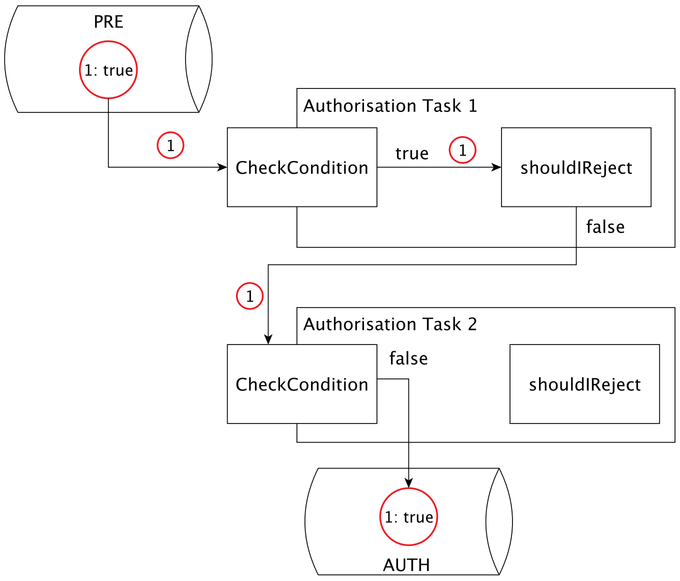


Figure 4: An example of an authorised event

In Figure 4, PreEvent “1” arrives for authorisation in the first authorisation task and it is checked against the “checkCondition” method of the task. The outcome is “true” meaning that this authorisation task contains logic that it is applicable to queries such the one included in event 1. Thus, the event is passed to the “shouldIReject” method of the task. For the sake of this example, task 1 does not reject event 1 thus it is passed to the “checkCondition” method of the next task in the chain (i.e., authorisation task 2). The “checkCondition” of this task evaluates to false which means that this specific task is not applicable to approve or reject such queries included in the event. As there is not task left in the authorisation chain, the event is published to the AUTH queue having its authorised flag set to “true” as none of the tasks rejected it.

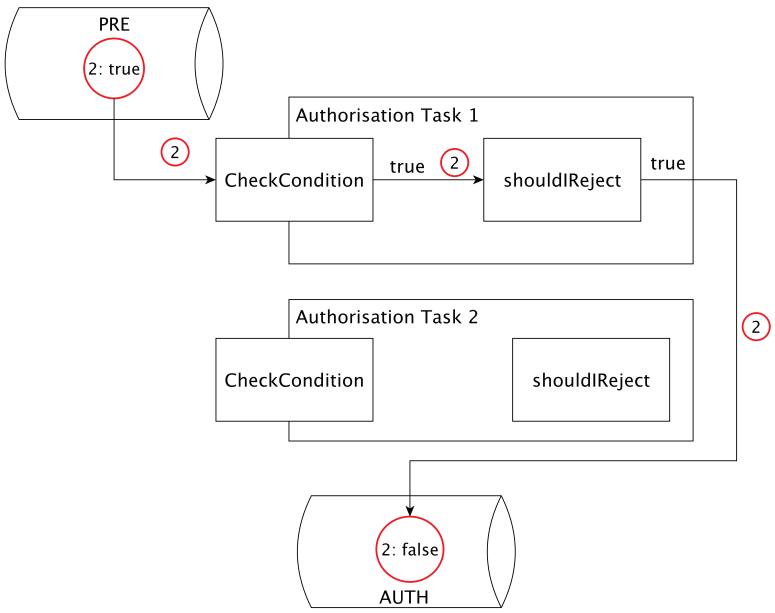


Figure 5: An example of a rejected event

Figure 5 presents the flow of a rejected event through the authorisation queue. Event “2” arrives from the PRE queue and is passed to the “checkCondition” method of authorisation task 1. The method evaluates to “true”, thus the event is passed to the “shouldIReject” method of the task. In this example let’s assume that task 1 rejects the execution of the query included in the event 2 based on the logic defined the method. As the event has already been rejected, there is no reason to be evaluated from other tasks, thus it is automatically pushed to the AUTH queue having its authorised flag set to “false”.

3.2.3 Orchestrator

The authorisation chain is based on the concept of *Side Outputs* available in Flink. Each stream of data in Flink can be transformed to another stream in which the data is grouped using *tags* based on some logic defined in the transformation operator. Figure 6 shows an example of using Flink’s side outputs.

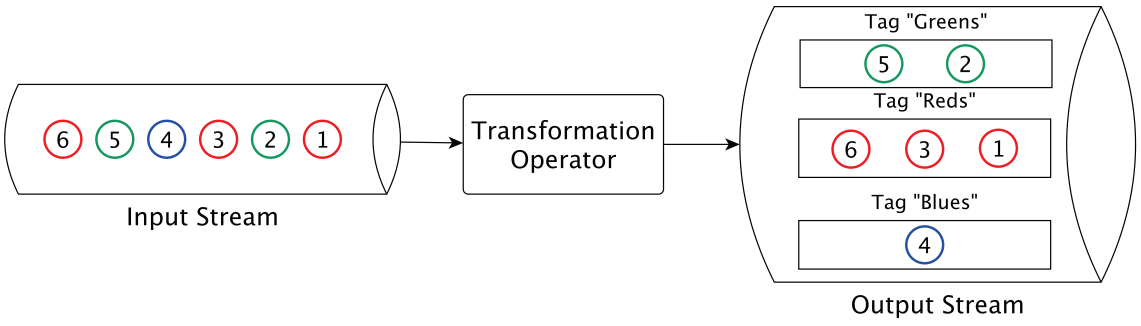


Figure 6: Example of using Flink's Side Outputs

The data in a Flink stream pass through a transformation operator (e.g., a Process operator) that separates them based on an attribute (i.e., color). The data is pushed in another stream which has three groups, tagged as “Greens”, “Reds” and “Blues”. Using Flink’s functionality, one can apply any operator on one or more of the grouped items of the stream.

The analytics architecture, by exploiting the side outputs functionality of Flink, automatically tags PreEvents into specific groups that facilitate the orchestration of the flow of events within the authorisation chain. More specifically, all rejected events, no matter which task rejected them, end up in a group tagged “Rejected”. The events that were either approved or not checked (because of the “checkCondition” method returning “false”) by a task are placed under the group which is tagged by the name of the Task. Those are given as input to the next task in the chain where the process is repeated.

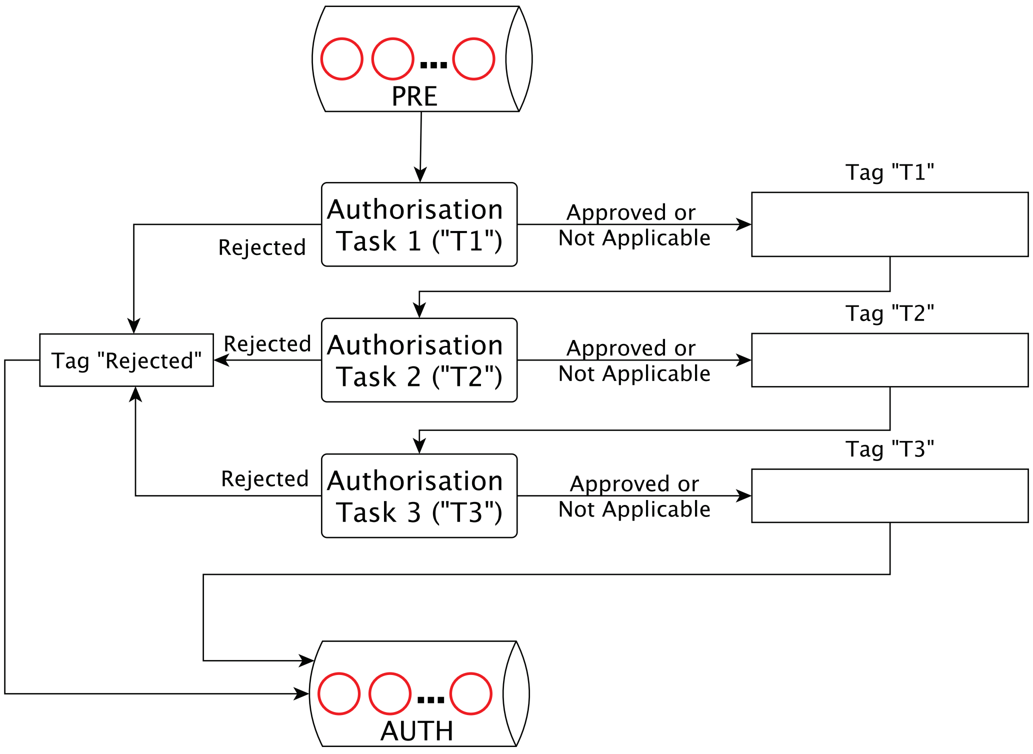

PRE
Authorisation Task 1 ("T1")
Approved or Not Applicable
Tag "T1"
Rejected
Authorisation Task 2 ("T2")
Approved or Not Applicable
Tag "T2"
Tag "Rejected"
Rejected
Authorisation Task 3 ("T3")
Approved or Not Applicable
Tag "T3"
Rejected
AUTH

Figure 7: Example of events flow in the authorisation task chain using side outputs

Figure 7 presents an example where three authorisation tasks exist in the authorisation chain. The first task in the chain “T1” consumes directly from the PRE queue. If the task is not responsible for checking such events or if it approves it, it passes it to the output stream to the group tagged as “T1”. If the event is checked and rejected, it is pushed to the group “Rejected”. Task “T2” consumes events coming from the group “T1” which is the task preceding in the chain. In the same manner as before, if the event is approved or not applicable, it is passed to a group tagged “T2”. In contrast, if it is rejected, it is passed to the “Rejected” group. The same process is applied for task “T3” which consumes from task “T2” and publishes the rejected events to the same “Rejected” group. As “T3” is the last task in the chain, it publishes the approved events directly to the AUTH queue. These are all the events that were approved for execution by all the tasks in the queue either because the tasks were not responsible for checking them or because they passed successfully the authorisation logic of the tasks. All the tasks collected in the “Rejected” group are published to the AUTH queue, but they have their authorised flag set to false.

```
// The PRE queue
DataStream<Event> dataStream = StreamManager.initializeSource(PRE,
PreEvent.class);

// The “Rejected” tag
OutputTag<Event> rejectedOutputTag = new OutputTag<Event>("Rejected") {};

// Instances of the Authorisation Tasks implemented by the user
AuthoristionTask1 T1 = new AuthoristionTask1();
AuthoristionTask2 T2 = new AuthoristionTask2();
AuthoristionTask3 T3 = new AuthoristionTask3();

// The tags for the approved/not applicable group for each task
OutputTag<Event> T1OutputTag = new OutputTag<Event>(“T1”) {};
OutputTag<Event> T2OutputTag = new OutputTag<Event>(“T2”) {};
OutputTag<Event> T3OutputTag = new OutputTag<Event>(“T3”) {};

// The run method is provided by the GenericAuthorisationTask
SingleOutputStreamOperator<Event> splitStream1 = T1.run(dataStream, T1);
SingleOutputStreamOperator<Event> splitStream2=
T2.run(splitStream1.getSideOutput(task1OutputTag), T2);
SingleOutputStreamOperator<Event> splitStream3=
T3.run(splitStream2.getSideOutput(task2OutputTag), T3);

// The final group (“T3”) is collected and pushed to the AUTH queue
DataStream<Event> finalApprovedStream =
splitStream3.getSideOutput(task3OutputTag);
StreamManager.initializeSink(AUTH, finalApprovedStream);


// The rejected events are collected and pushed to the AUTH queue
DataStream<Event> finalRejectedStream =
splitStream1.getSideOutput(rejectedOutputTag)
              .union(splitStream2.getSideOutput(rejectedOutputTag)
              .union(splitStream3.getSideOutput(rejectedOutputTag));
StreamManager.initializeSink(AUTH,finalRejectedStream);
```

Listing 1: The java orchestrator for the authorisation tasks execution

A part of the orchestrator Java application for the above example is shown in Listing 1. The *run(…)* method that is implemented in the *GenericAuthorisationTask* class, is responsible for calling the transformation method (i.e., a Flink “process” operator) for re-directing the events to the appropriate groups based on the results of each task’s “checkCondition” and “shouldIReject” methods.

A code generator was developed to produce the orchestrator Java class shown in Listing 1 using a model-to-text (M2T) transformation. The input to the M2T transformation is a model conforming to the metamodel shown in Figure 8.

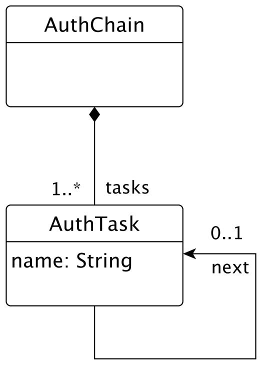


Figure 8: The authorisation chain metamodel

Each authorisation chain (i.e., *AuthChain)* consists of a number authorisation tasks (i.e., *AuthTask)*. Each authorisation task has a name which is the name of the implementation class and a reference *next* pointing to the exactly one[2] task in the chain. A part of the M2T transformation written in the Epsilon Generation Language [20] is shown in Listing 2. The generator iterates through all the tasks and creates their class instances and the equivalent output tags (lines 1-7). Then in lines 10-27, streams that include the "approved" elements for each task and callers to the *run(…)* method of each task are created. In line 29, the stream of the last task in the task (which includes all the approved element) is defined. Finally, in lines 33-43, the stream that includes all the rejected elements is created by combining (through the *union* operator) the rejected tasks from each task.

```
 1  [% for (task in  chain.tasks) { %]
 2      [%=task.name%] [%=task.name.ftlc()%]  = new [%=task.name%]();
 3      OutputTag<Event> [%=task.name.ftlc()%]OutputTag = new OutputTag<Event>
 4          ([%=task.name.ftlc()%].getLabel()) {};
 5
 6  [%}
 7  %]
 8
 9  [%
10  var firstTask = AuthTask.all().selectOne(t|not(AuthTask.all().next.contains(t)));
11  var currentTask = firstTask;
12  var previousTask = null;
13  %]
14  SingleOutputStreamOperator<Event> [%=currentTask.name.ftlc()%]SplitStream =
15      [%=currentTask.name.ftlc()%].run(dataStream, [%=currentTask.name.ftlc()%]);
16
17  [%
18  while (not (currentTask.next == null)) {
19      previousTask = currentTask;
20      currentTask = currentTask.next; %]
21      SingleOutputStreamOperator<Event> [%=currentTask.name.ftlc()%]SplitStream =
22      [%=currentTask.name.ftlc()%].run([%=previousTask.name.ftlc()%]SplitStream
23      .getSideOutput([%=previousTask.name.ftlc()%]OutputTag), [%=currentTask.name.ftlc()%]);
24
25  [%
26  }
27  %]
28
29  DataStream<Event> finalStream = [%=currentTask.name.ftlc()%]SplitStream
30      .getSideOutput([%=currentTask.name.ftlc()%]OutputTag);
31
32  [%
33  var finalRejectedStream = "";
34  for (task in  chain.tasks) {
35      finalRejectedStream = finalRejectedStream + task.name.ftlc() + "SplitStream
36          .getSideOutput(rejectedOutputTag)";
37      if (hasMore) {
38          finalRejectedStream = finalRejectedStream + ".union(";
39      } else {
40          finalRejectedStream = finalRejectedStream + ");";
41      }
42  }
43  %]
```

Listing 2: The orchestrator M2T transformation

We opted for the use of a model to define the authorisation chain, instead of other means (e.g., a file that lists the names of the tasks) as this can act as an extensibility mechanism in the future. Polystore developers might update the metamodel to include in the chain/tasks metadata of interest. For example, a field that denotes the priority level of each task might be useful to re-arrange the chain taking into account historic execution time for each, rejection rate and priorities.

[2] As the last task does not have a following task the multiplicity of the reference is 0..1

To summarise the authorisation chain creation and configuration, developers need to
1) Create new task in the model, positioning it to the correct place in the chain using the *next* reference.
2) Create a Java class for the authentication task by implementing the provided interface
3) Run the generator to produce the runner class
4) Redeploy the polystore to include the new task created

3.2.4 Security Concerns

There are two main types of security issues to consider with regards to the Analytics work package: ones related to the use of internal communication channels (Kafka) and ones related to accessing the Typhon polystore.

Regarding data passing through the internal communication channels of the analytics engine, the security aspects can be managed by the channel itself (Kafka). Kafka offers both authentication and data encryption mechanisms that can be leveraged to disallow any access to the internal queues to anyone other than the analytics engine, should this be required. Furthermore, the system can be configured to only allow access to its channels through the internal network itself, should the analytics engine reside alongside the Kafka brokers. In this case there is no external access possible, further limiting any avenues of attack, as access to the system itself would be one of the few remaining ways to gain access to the Kafka channels.

With respect to the latter, logic both in the authentication and analytics code is able to access all data passing through the polystore. It may be the case that an organisation wants to separate their data into levels and allow certain users access only to certain elements. Since polystores does not support user-based permissions or a similar data separation policy at this stage as this was beyond the scope of the TYPHON project, the analytics infrastructure cannot enforce more fine-grained user-level access control.

3.2.5 Related Work

Database and big data security and authentication is a topic that has been investigated for some time, largely through the use of various security or database audit frameworks.

Apache Metron[3] [21] is a security analytics framework that works with big-data sources such as Apache Kafka. It contains tools such as a profiler, that can read from such sources and set up a profile for applicable elements (similar to the *checkCondition(Event event)* method of our tool). This profile can then be used to analyse the incoming data to create meaningful metrics, for example by using other Metron packages such as its statistical package. This approach is similar to the authentication component of our architecture and gives us confidence that our approach for handling polystore authentication is grounded in work that is already tried and tested.

Database audit frameworks such as ApexSQL audit and compliance tools[4] or MySQL Enterprise Audit[5] allow for the monitoring of activity on relational databases and create audit trails that can be used to pinpoint security or malicious use of the system. Such systems often use policies to define what should be monitored, offering a higher level of

[3] https://metron.apache.org/
[4] https://www.apexsql.com/sql-tools-audit.aspx
[5] https://www.mysql.com/products/enterprise/audit.html

abstraction for creating such rules, as well as more fine-grained filtering to lower the overhead and persistence sizes of such logs. Compared to our architecture, these technologies are more mature but are limited to auditing a single type of database each. Building upon this knowledge, our authentication system offers the fine-grained approach of monitoring specific classes of polystore queries.

3.3 DESERIALISATION

As described in Section 2.1, the analytics engine offers access to strongly-typed Java objects for consumption by the code running the analytics logic. This section presents how these objects are created.
When a PostEvent is consumed from the Post queue by the analytics engine, to be processed by the Flink code that has been written by the analytics experts, is contains the following data structures of interest:

- *resultSet* This is the JSON string returned from TyphonQL that holds the results of the execution of the query from which this *PostEvent* has been created, in their native format.
- *inverted*R*esultSet* This is a JSON string holding the results obtained for any additional queries that were created by the analytics engine in order to obtain further data about the query, that was unavailable through the results of the query itself. This is done through the concept of *inverted selects*, which will be discussed in detail in the next Section 3.4. With regard to the creation of strongly-typed Java objects, this string can be considered of the same structure as the one in the *resultSet*.

The first job of the analytics component that consumes these *PostEvent*s is hence to deserialise these JSON strings into Java objects representing the types found in the TyphonML model the data experts had defined for the polystore. Below is a summary of the steps that are performed:

- A model-to-text transformation is executed, creating the appropriate Java classes from the TyphonML model, all of which extend the common *Entity* superclass.
- A *DeserialisedPostEvent* is created, extending the *PostEvent* with a list of (DML) commands; each of these commands will either be an update, a select, a delete or an insert.
- These commands will each contain the actual clause of the command in question, a map containing the types and properties of the TyphonML elements affected by this command, as well as the list of actual *Entities* that were affected. The actual Java types of these entities will be one of the concrete types created by the model-to-text transformation performed at the beginning of this process.

In order to parse the query and to instantiate the correct concrete Java classes, the engine uses the TyphonQL query parser to convert it to an abstract syntax tree (AST). Using this AST the engine then obtains the name of the affected entity and using reflection, it creates the relevant instance and populates any of the fields that have been affected by this DML command:

- Insert: entities with all fields that were given a value are included
- Delete: entities deleted, including their values before deletion are included
- Select: returned entities, including values requested by the command are included
- Update: updated entities, including the previous and current values of any updated field are included (through the *previousValue* object)

As these results can be arbitrarily large, the analytics expert is able to disable retrieval of these entities, if they know they do not need them for their analytics. This is done by setting the flag *resultSetNeeded* to False, for PreEvents that the expert does not wish this data to be captured for, as part of their authentication chain logic.

3.3.1 Handling of References to other Entities

Entities can have values that are other Entities; for example a *User* of an e-commence system can have an *Address*. The way this is handled is through the creation of proxies. Whenever a field has a value that is another Entity, a proxy instance of this Java class is created, with the *isProxy* field set to true and with only its identifier value set. As such, if the analytics developer requires to further navigate to this Entity as part of their analysis, they can use its identifier to query the polystore and obtain any further data required. This approach is chosen as it reflects what is returned by the polystore in such cases, and since resolving these proxies by the engine would have to perform the exact same query to the polystore regardless.

3.4 Change Capture

One of the limitations when offering analytics originating from DML commands is that only the data returned by the command itself is normally available. This section discusses how the analytics engine tackles the issue of change capture when insufficient data is returned by the polystore. Since insert statements already include all of the data required to create the *Entities* required, and since select statements do not alter any data, no additional changes are needed when those commands are executed.

3.4.1 Related Work

We have considered three alternatives for manage change capture: additional database queries, database triggers [22] and use of database logs (such as those provided by many popular databases like MySQL[6])

The simplest way to manage this issue is to create the necessary queries to the database whenever this data is needed, and before it is manipulated. Unless this is automated though, it will likely be a very complex process for analytics developers to perform themselves, as they would likely have to retrieve this data before their actual query is executed (and hence may have already altered the data). On the other hand, automating the process will incur additional overhead as data that may never be used will end up being collected alongside useful data, unless a very fine-grained customisation is provided on the user-level (such as being able to pick exactly what Entities and what Fields they want previous values for).

Another way to obtain this data is using database triggers. This approach is similar to querying the database and has the added benefit of being integrated with many database systems. On the other hand, the fact that the polystore aims at supporting a wide range of databases may be a limiting factor, as some such systems may either not offer or offer a substantially different way of exposing such triggers as presented in D5.1.

[6] https://dev.mysql.com/doc/refman/5.7/en/server-logs.html

The final way to obtain this data is using database logs. Systems such as Debezium [17] or the Qlik Data Integration Platform[7] read the database logs of supported databases and expose this data to be consumed through appropriate channels such as Kafka. The most significant limitation of such systems is that different database technologies offer very different logging to one another. For example, SQL databases often log enough data so that the previous values are available, but many NoSQL ones will either not log at all or will log much more limited data, hence not providing the system with enough data to re-create previous values. Finally, it is worth noting that commercial tools like HVR's CDC[8] or Oracle Database CDC[9] offer both trigger-based (synchronous) and log-based (asynchronous) change management.

We decided to offer an automated form of the first approach, to allow for customisation, generalisation and extensibility through code generation techniques. This approach can be extended to support any database system regardless of whether they offer triggers or logs and it can be customised to allow for more limited data to be captured based on user requirements.

3.4.2 Delete and Update Statements

When a delete TyphonQL command is issued and executed, the results only include the identifiers of the element(s) that have been deleted. Since the elements are now deleted, there is no way to query the database in order to retrieve the values they had before their deletion. As those values can be of use to the analytics developers, an *inverted select* (described below) is created and executed before the delete itself, capturing the values of those *Entities* just before deletion.

Similarly, when an update command is issued and executed, the updated values are available through the command itself but since they are then updated in the database, there is no way to query the database in order to retrieve these values before their change. As those previous values can be of use to the analytics experts, an *inverted select* (described below) is created and executed before the update itself, similarly to how this is done for deletion statements.

It is worth noting that since the polystore does not support the concept of cross-database atomic transactions, there is no way for us to guarantee temporal ordering. For example if the user issues a command to delete cheap products from the database (for example all products with a price less than 10) but between the retrieving of those products (through the inverted select) and the actual delete query being executed, one of those products has their price updated (in this case to greater than 10), then our previous values are not accurate. This is an inherent limitation of unifying different database instances and is documented in the user guides, so that the analytics experts are aware of this and can either chose to use this functionality knowing its limitations, or they can chose to turn it off and rely on the queries themselves for the analytics.

3.4.3 Inverted Selects

When more information is needed for providing a complete *Entity* by the analytics engine, and such information cannot be obtained during the execution of the analytics themselves, the concept of *inverted selects* is used. This concept uses the *where* clause in a delete

[7] https://www.qlik.com/us/data-streaming/data-streaming-cdc
[8] https://www.hvr-software.com/product/change-data-capture/
[9] https://docs.oracle.com/cd/B28359_01/server.111/b28313/cdc.htm

statement and the *where* and *set* clauses of an update statement and creates a *select* statement with those clauses instead. For example:

- delete User u where u.@id == #1
    - creates: from User u select u.@id, u.name, u.address, u.comments, u.paymentDetails, u.orders, u.reviews, u.basket
- update Address a where a.@id == #2 set { street: "street 18" }
    - creates: from Address a select a.@id, a.street

These select statements are then executed and their results are stored in the *invertedResultSet* of the *PostEvent* elements created by the polystore. As discussed in Section 3.3, the analytics developer will now have access to both the results from the command as well as any additional results from any inverted selects that were created and executed, for their analytics program. It is worth noting that referenced elements (for example the reviews a user has created) are retrieved by proxy (an element only containing the identifier in the database) in order to limit retrieving a potentially deeply nested structure. Furthermore, similarly to disabling the result set, the analytics developer is able to disabling retrieving the inverted result set as well, through the flag *invertedNeeded* in the PreEvent, if they do not need it in their analytics programs.

3.5 Requirements Coverage

In this section we present the requirements that the proposed architecture covers as those defined by the industrial partners. Table 1 summarises the *Use Case* requirements related to the data analytics & monitoring architecture along with their overall priority as it is presented in D1.1.

Table 1: Data analysis and monitoring use case requirements table

ID	Requirement	Overall Priority	Coverage
70	The polystore shall offer a mechanism for trigger definition and execution on update operations	SHALL	Supported
71	The polystore shall offer a mechanism for trigger definition and execution on data access operations	SHALL	Supported
72	The polystore shall offer a mechanism to define and retrieve clusters of objects with similar properties	SHALL	Supported
73	The polystore shall offer a mechanism to enable the identification of changes in stored trend displays	SHALL	Supported
74	The polystore shall offer a mechanism to hold historical searches	SHALL	Supported
75	The polystore may offer a mechanism to plugin processing components that transform data when its ingested	SHOULD	Supported
76	The polystore may offer a mechanism to plugin processing components that transform data when its retrieved	SHOULD	Not Supported

- **Requirements 70 & 71:** Both requirements are fully supported by the proposed architecture. This can be achieved either as an authorisation task if the trigger needs to be called before the execution of the query or as an analytics task if needs to be triggered afterwards.
- **Requirement 72:** This requirement is fully covered by the capabilities of the proposed architecture to define analytics tasks after the execution of each query. This is also facilitated by Apache Flink's operators for grouping items sharing same

properties, Flink's Complex Event Processing (CEP) libraries to group items based on patterns and more generic operators (e.g., map, process, apply, etc.) to define (or call) custom clustering algorithms.

- **Requirement 73:** Trends can be extracted by analysing relevant queries and monitored over time through Flink-based analytics components.
- **Requirement 74:** As all queries go through the analytics architecture, historical queries (searches) can be saved and later retrieved by an application if that is desirable.
- **Requirement 75:** This requirement can be achieved by using exploiting the authorisation task functionality. As these tasks are triggered before thee execution of a command, users are able to amend the data manipulation command and make the necessary transformations.
- **Requirement 76:** This requirement is not supported as there is not a mechanism available to amend the retrieved data returned by the Polystore. One can only interfere to amend the retrieval query but this will only impact which values will be retrieved but not the values themselves.

Table summarises the *Component* requirements for the data analytics & monitoring architecture as these described in section 10.4 of D.1.1.

Table 2: Data analysis and monitoring component requirements table

ID	Requirement	Overall Priority	Coverage
44	Distributed execution of real time analytics and monitoring facilities shall be supported.	SHALL	Supported
45	Execution of analytics and monitoring facilities shall be triggered by data access and update requests and events generated by TyphonQL.	SHALL	Supported
46	Fault tolerant execution of the analytics and monitoring facilities shall be supported.	SHALL	Supported
47	Publishing and subscribing to data access requests and events to a distributed message queue through a distributed messaging channel shall be enabled.	SHALL	Supported
48	Publishing and subscribing to data update events and events shall be enabled.	SHALL	Supported
49	Criteria based subscription to data events shall be enabled.	SHALL	Supported
50	The development of text mining pipelines for data events shall be simplified.	SHALL	Supported
51	Facilities for batching data events (e.g. per session) should be provided.	SHOULD	Partially Supported
52	Analytics facilities to prevent the fulfilment of data access/update requests[10] may be enabled.	MAY	Supported

- **Requirement 44:** The requirement is fulfilled by allowing developers to define analytics and monitoring tasks using method wrappers provided that can be executed over Apache Flink clusters.
- **Requirement 45 & 47, 48:** Appropriate events are generated and published automatically in a distributed message log (i.e., Apache Kafka [1]). The analytics

[10] Motivating scenario: In an e-commerce system, users may not be allowed to change their address more than three times within 12 hours (for fraud prevention reasons). This restriction can be implemented in a modular way in the context of a dedicated monitoring component, which can cause the API's e.g. User.setAddress() method to fail with an appropriate exception if this restriction is about to be violated.

tasks defined are automatically subscribed to the appropriate topic of the Kafka log and are triggered every time an access/update request is submitted.

- **Requirement 46:** This is achieved by implementing the architecture atop fault-tolerant infrastructure such as Apache Kafka and Apache Flink.
- **Requirement 49:** Analytics developers can subscribe and consume events stored in the POST Kafka queues and filter those fulfilling their criteria in the appropriate Flink "filter" operators.
- **Requirement 50:** Text mining pipelines are implemented atop Apach Flink and thus are compatible to run on the established Apache Flink infrastructure.
- **Requirement 51:** This can be achieved by exploiting the "slots" attribute in the PreEvent objects for adding information related to batching of data events (e.g., a session id).
- **Requirement 52:** This is supported through the authorisation tasks.

Finally, following are the cross-component requirements related to WP5.

ID	Requirement	Overall Priority	Coverage
53	The TyphonQL execution engine shall publish data access and update requests and events to the analytics and monitoring framework's distributed messaging channel [WP4].	SHALL	Supported
54	The generated polystore API shall publish data access and update requests and events to the analytics and monitoring framework's distributed messaging channel [WP2].	SHALL	Supported
55	The TyphonQL execution engine may support rejection of data access and update requests based on feedback from analytics facilities [WP4].	SHALL	Supported
56	The generated polystore API may support rejection of data access and update requests based on feedback from analytics facilities [WP2].	SHALL	Supported

4. DEPLOYMENT

In this section we discuss the deployment capabilities of the analytics architecture. This builds atop the Docker deployment presented in D5.3; a summary of it is presented here as a reminder to the reader and a few changes since the last version are also discussed. In Section 4.2 we discuss the newly-introduced Kubernetes deployment in detail. We suggest the use of the Docker deployment for development and testing of analytics and authorisation tasks, while the Kubernetes one for deployment in a distributed environment.

4.1 DOCKER

In this section, we present how the proposed analytics architecture is packaged and deployed using Docker's [23] containerisation technology. Apache Kafka messaging infrastructure requires Zookeeper [24] as an orchestrator, thus one should first initialise a container that starts up Zookeeper. Docker promotes the re-use of images that were developed to perform specific tasks. Thus, we instantiate all the necessary infrastructure for Kafka [1] and Zookeeper [24] using an already-developed image available on Docker Hub [25]. The wurstmeister/kafka docker image[11], used in our work, allows the deployment

[11] https://hub.docker.com/r/wurstmeister/kafka

of Kafka and Zookeper in a parameterized way using a docker-compose file. In addition, we merged the official Docker image of Apache Flink[12] into the existing docker-compose configuration script. The relevant parts of the docker-compose file are shown in Listing 3.

In lines 1-5, Zookeeper is configured to listen to port 2181. This is the port where Kafka will access to communicate and exchange all the necessary information with Zookeeper. In lines 7-22, the Kafka container is configured. What has changed since the deployment presented in D5.3 is the capability of Kafka queues to be accessible from outside the Docker deployment using this new configuration. More specifically, previously, one should deploy the analytics scenarios (both authorisation and analytics tasks) inside a new Docker container that has Java installed inside the same Docker network with the rest of the analytics infrastructure (i.e., Zookeeper and Kafka containers). Lines 24-33 configure a Flink job manager with published port 8081 (i.e. web browser based Flink user interface). Similarly, lines 34-43 configure a Flink task manager. In this new version the analytics infrastructure is now accessible from outside Docker by using the advertised listeners feature of Kafka (see line 13).

Analytics experts need to be able to test the scenarios they develop by running them, for example, from their local IDE instead of having them exported into runnable JARs that then need to be deployed in a new Docker container. In other words, new analytics scenarios can be tested by simply running their main method in the IDE, saving up time especially during debugging.

In the previous deliverable (i.e., D5.3) we focused on how analytics experts can deploy their scenarios using docker images. In Appendix A, we provide a detailed guide on how one can write and run analytics scenarios outside Docker using the new configuration described above.

Finally, in lines 44-45 we create a start a default authorisation task. In order for TyphonQL commands to be executed, they first need to be approved by the one or more authorisation tasks. However, it might be the case that users of the Polystore do not wish to use the authorisation mechanism. This is either because it is not part of their requirements at the moment or because they plan to implement the authorisation tasks in the future. Thus, an image that contains a default authorisation task that authorises the execution of all the TyphonQL queries is deployed when the Polystore starts. Users wishing to define custom authorisation tasks, need to stop this container and instead deploy their own custom authorisation tasks chain (see Section 3.2).

[12] https://hub.docker.com/_/flink

```yaml
zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - target: 2181
        published: 2181
  kafka:
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_LISTENERS: OUTSIDE://:29092, INSIDE://:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT, OUTSIDE:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
      KAFKA_ADVERTISED_LISTENERS: OUTSIDE://localhost:29092, INSIDE://:9092
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
    depends_on:
      - zookeeper
    ports:
      - target: 29092
        published: 29092
    build: .
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

  jobmanager:
    image: flink:latest
    environment:
      JOB_MANAGER_RPC_ADDRESS: jobmanager
    ports:
      - target: 8081
        published: 8081
    command: jobmanager
    expose:
      - 6123
  taskmanager:
    image: flink:latest
    environment:
      JOB_MANAGER_RPC_ADDRESS: jobmanager
    depends_on:
      - jobmanager
    command: taskmanager
    expose:
      - 6121
      - 6122
  authAll:
    image: zolotas4/typhon-analytics-auth-all
```

Listing 3: Docker-compose YAML file for Kafka/Zookeeper parametrisation and instantiation

4.2 KUBERNETES

The TYPHON analytics engine can also be deployed on a local or remote Kubernetes cluster as described in the following and in particular from the generation of TyphonDL deployment scripts to the evaluation of successful deployment. The requirement of the local and remote Kubernetes cluster is bound to the availability of the minikube and kubectl command-line tools. The installation of minikube[13] and kubectl[14] is described at the official Kubernetes documentation. Instructions for setting up a Kubernetes pool based on macOS version 10.15.4, Strimzi v0.17.0[15], and DigitalOcean is provided in APPENDIX B.

4.2.1 Generate deployment scripts using TyphonDL

The first step is to follow the guidelines on how to employ TyphonDL in deliverable D3.3 Section 3 or D3.4 (i.e. reporting on the final version of the "Hybrid Polystore Deployment Language").

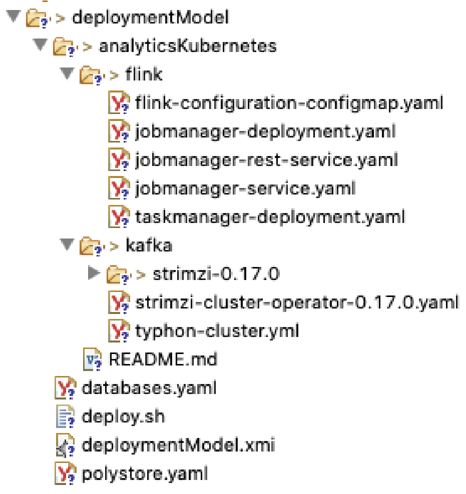
[13] https://kubernetes.io/docs/setup/learning-environment/minikube/#installation
[14] https://kubernetes.io/docs/tasks/tools/install-kubectl/
[15] An overview guide of Strimzi is available online at https://strimzi.io/docs/operators/master/overview.html.

The TyphonDL wizard for Kubernetes allows to configure the parameters depicted in Table 3. Most importantly, *Kafka URI* defines the DNS/IP address and port of the machine running the Kafka instance; *Flink jobmanager heap size* defines the overall heap memory of the Flink jobmanager instance; *Flink taskmanager memory process size* defines the memory size of Flink taskmanager instances; *Flink taskmanager replicas* defines the number of Flink taskmanager instances; *Kafka replicas* defines the number of Kafka broker instances; and *Kubeconfig* defines the path and file name of the Kubernetes cluster access file (i.e. provided by the cloud provider running a Kubernetes cluster).

Table 3: Typhon Analytics Kubernetes configuration parameters in TyphonDL

Parameter	**Default value**
Kafka URI	typhon-cluster-kafka-bootstrap:9092
Flink jobmanager heap size	1024m
Flink taskmanager memory process size	1024m
Loggerlevel rootlogger	INFO
Logging root target	file
Logglevel akka	INFO
Logglevel kafka	INFO
Logglevel Hadoop	INFO
Logglevel zookeeper	INFO
Logglevel flink	ERROR
Logging flink target	file
Flink jobmanager rest nodeport	automatic
Flink taskmanager replicas	2
Kafka replicas	1
Kafka version	2.4.0
Kafka storage claim	100Gi
Zookeeper storage claim	100Gi
Kubeconfig	myKubeConfig.kubeconfig

From the settings in Table 3, the TyphonDL plugin generates a folder with a set of Kubernetes YAML-based scripts as indicated in Figure 9. Most importantly, the latter contains the folders "flink" and "kafka" with respective Kubernetes deployment configurations for Flink and Kafka; as well as a script named "deploy.sh" which wires individual deployment steps together (cf. Section 4.2.2).



```
▼ > deploymentModel
  ▼ > analyticsKubernetes
    ▼ > flink
        flink-configuration-configmap.yaml
        jobmanager-deployment.yaml
        jobmanager-rest-service.yaml
        jobmanager-service.yaml
        taskmanager-deployment.yaml
    ▼ > kafka
      ▶ > strimzi-0.17.0
        strimzi-cluster-operator-0.17.0.yaml
        typhon-cluster.yml
      README.md
    databases.yaml
    deploy.sh
    deploymentModel.xmi
    polystore.yaml
```

Figure 9: Generated Typhon Analytics Kubernetes configuration files

4.2.2 Deploy Apache Kafka and Flink in remote Kubernetes cluster

Apache Kafka and Flink pods and services can be deployed to a remote Kubernetes cluster by executing the generated “deploy.sh” script depicted in Listing 4. This script employs the generated Kubernetes YAML-based scripts during deployment in a series of commands of which the most essential are indicated by Listing 4 and described in the following.

First, line 2 sets a variable named *kubeconfig* to the file system path containing the Kubernetes configuration file that has been retrieved from the cloud provider (cf. Section 4.2.3) and used throughout the script as extension of the *kubectl* command (e.g. *$kubeconfig* in line 4).

Second, line 11 applies the generated *databases.yaml* configuration, which contains Kubernetes database deployments and services such as a mongoDB for the Typhon polystore as well as a batch job inserting models into the created database. Third, line 20 executes a command that completes when the insertion of models into the deployed database is completed or 300 seconds have passed. Similarly, line 24 executes a command that completes when all deployed databases are ready, or 100 seconds have passed.

Fourth, line 27 applies the generated *polystore.yaml* configuration and in particular deploys the Typhon polystore service API and UI as well as a TyphonQL server Kubernetes service. Next, line 30 executes a command that completes in 300 seconds or when the Typhon API, UI, and QL Kubernetes deployments have completed.

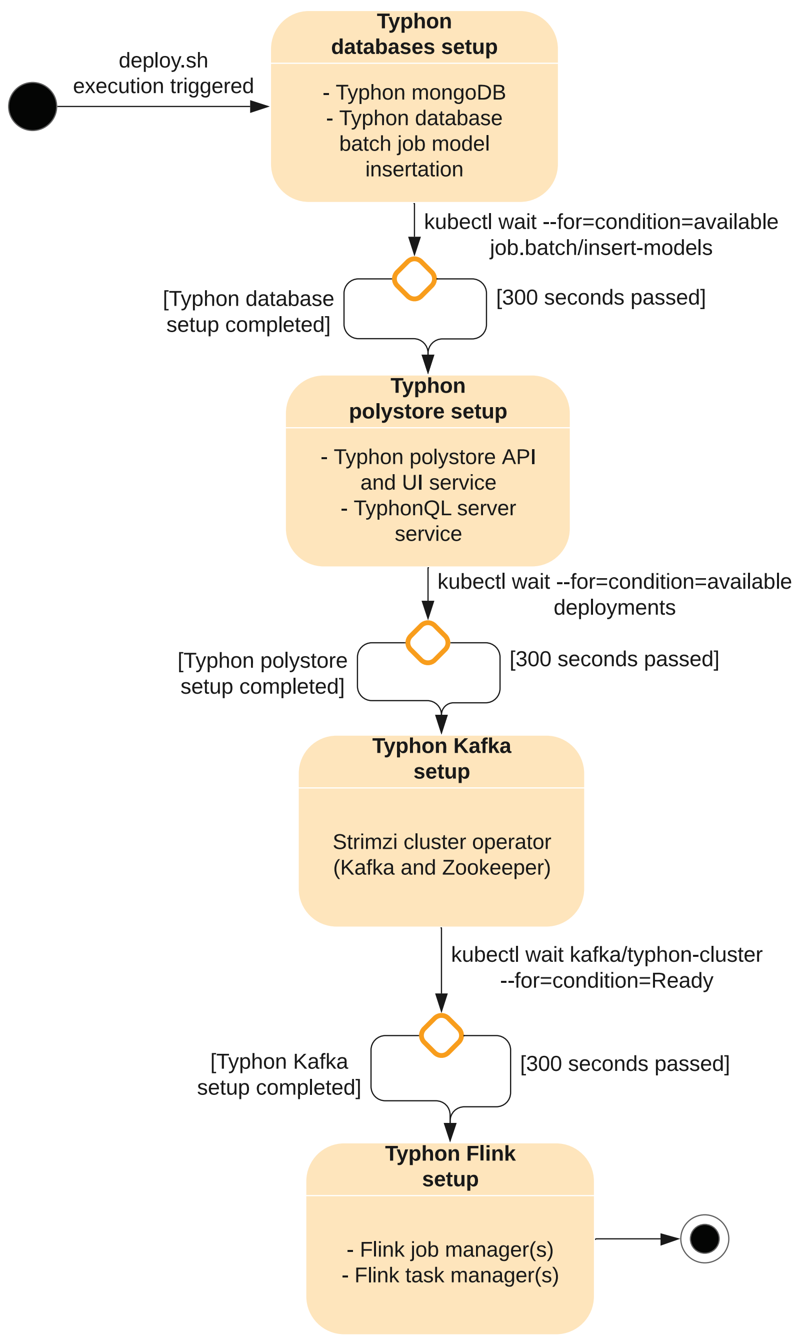
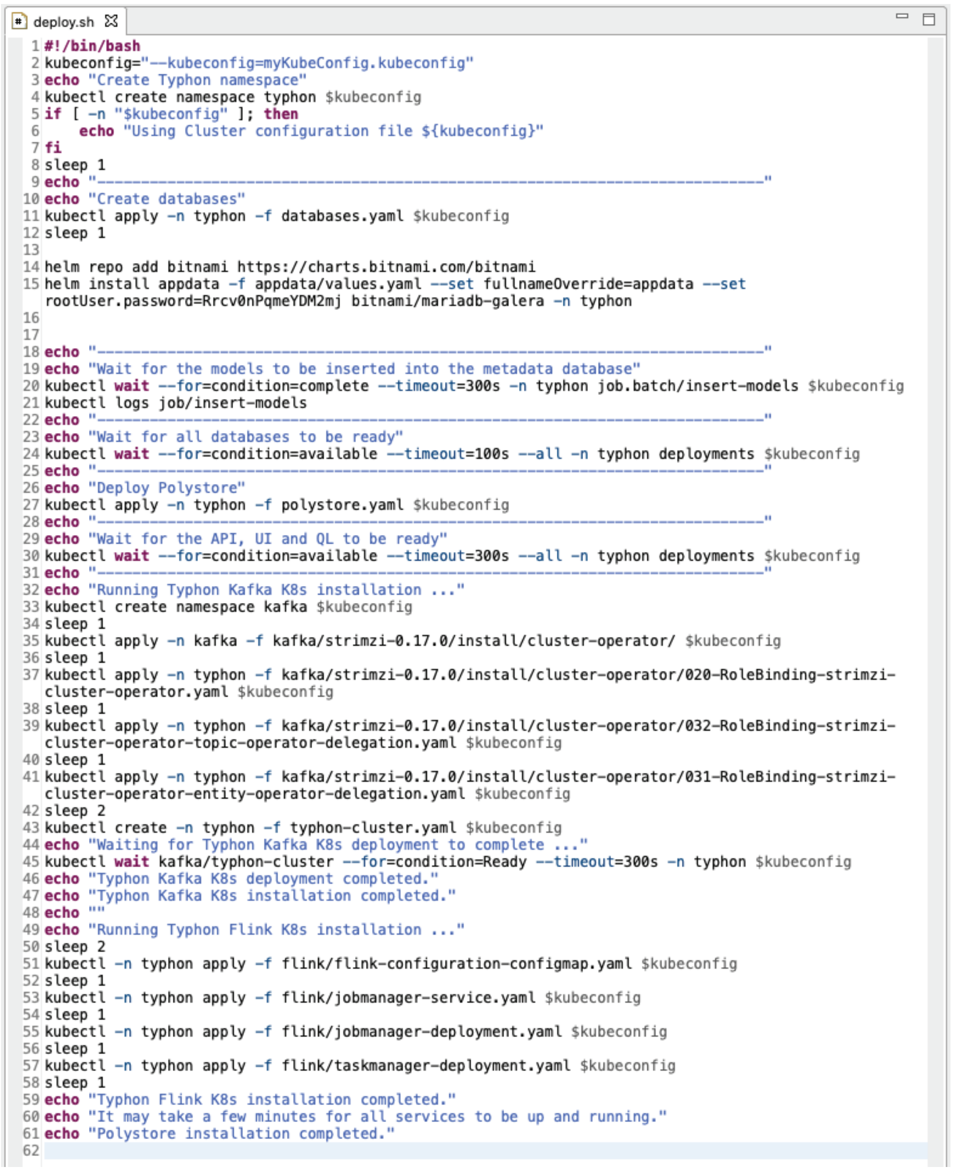


Figure 10: Typhon Analytics Kubernetes deployment state machine diagram

Fifth, lines 35-41 deploy Apache Kafka and in particular the Strimzi cluster operator (i.e. indicated by configuration files contained in the *cluster-operator* subdirectory of the Strimzi installation directory). More specifically, the official Strimzi deployment instructions[16] are closely followed. Sixth, line 43 indicates the application of the configuration file *typhon-cluster.yaml* which creates a Kubernetes cluster of type Kafka containing a number Kafka and Zookeeper instances (i.e. indicated by the *replicas* attribute and default replication number: 1) with persistent storage claims. Next, line 45 indicates a command that completes either then the Typhon cluster is ready, or 300 seconds have passed.

[16] https://strimzi.io/docs/operators/master/deploying.html#cluster-operator-str



deploy.sh

```
 1 #!/bin/bash
 2 kubeconfig="--kubeconfig=myKubeConfig.kubeconfig"
 3 echo "Create Typhon namespace"
 4 kubectl create namespace typhon $kubeconfig
 5 if [ -n "$kubeconfig" ]; then
 6     echo "Using Cluster configuration file ${kubeconfig}"
 7 fi
 8 sleep 1
 9 echo "-----------------------------------------------------------------------------"
10 echo "Create databases"
11 kubectl apply -n typhon -f databases.yaml $kubeconfig
12 sleep 1
13
14 helm repo add bitnami https://charts.bitnami.com/bitnami
15 helm install appdata -f appdata/values.yaml --set fullnameOverride=appdata --set
   rootUser.password=Rrcv0nPqmeYDM2mj bitnami/mariadb-galera -n typhon
16
17
18 echo "-----------------------------------------------------------------------------"
19 echo "Wait for the models to be inserted into the metadata database"
20 kubectl wait --for=condition=complete --timeout=300s -n typhon job.batch/insert-models $kubeconfig
21 kubectl logs job/insert-models
22 echo "-----------------------------------------------------------------------------"
23 echo "Wait for all databases to be ready"
24 kubectl wait --for=condition=available --timeout=100s --all -n typhon deployments $kubeconfig
25 echo "-----------------------------------------------------------------------------"
26 echo "Deploy Polystore"
27 kubectl apply -n typhon -f polystore.yaml $kubeconfig
28 echo "-----------------------------------------------------------------------------"
29 echo "Wait for the API, UI and QL to be ready"
30 kubectl wait --for=condition=available --timeout=300s --all -n typhon deployments $kubeconfig
31 echo "-----------------------------------------------------------------------------"
32 echo "Running Typhon Kafka K8s installation ..."
33 kubectl create namespace kafka $kubeconfig
34 sleep 1
35 kubectl apply -n kafka -f kafka/strimzi-0.17.0/install/cluster-operator/ $kubeconfig
36 sleep 1
37 kubectl apply -n typhon -f kafka/strimzi-0.17.0/install/cluster-operator/020-RoleBinding-strimzi-
   cluster-operator.yaml $kubeconfig
38 sleep 1
39 kubectl apply -n typhon -f kafka/strimzi-0.17.0/install/cluster-operator/032-RoleBinding-strimzi-
   cluster-operator-topic-operator-delegation.yaml $kubeconfig
40 sleep 1
41 kubectl apply -n typhon -f kafka/strimzi-0.17.0/install/cluster-operator/031-RoleBinding-strimzi-
   cluster-operator-entity-operator-delegation.yaml $kubeconfig
42 sleep 2
43 kubectl create -n typhon -f typhon-cluster.yaml $kubeconfig
44 echo "Waiting for Typhon Kafka K8s deployment to complete ..."
45 kubectl wait kafka/typhon-cluster --for=condition=Ready --timeout=300s -n typhon $kubeconfig
46 echo "Typhon Kafka K8s deployment completed."
47 echo "Typhon Kafka K8s installation completed."
48 echo ""
49 echo "Running Typhon Flink K8s installation ..."
50 sleep 2
51 kubectl -n typhon apply -f flink/flink-configuration-configmap.yaml $kubeconfig
52 sleep 1
53 kubectl -n typhon apply -f flink/jobmanager-service.yaml $kubeconfig
54 sleep 1
55 kubectl -n typhon apply -f flink/jobmanager-deployment.yaml $kubeconfig
56 sleep 1
57 kubectl -n typhon apply -f flink/taskmanager-deployment.yaml $kubeconfig
58 sleep 1
59 echo "Typhon Flink K8s installation completed."
60 echo "It may take a few minutes for all services to be up and running."
61 echo "Polystore installation completed."
62
```

Listing 4: Typhon Analytics Kubernetes deployment shell script (deploy.sh) generated by TyphonDL

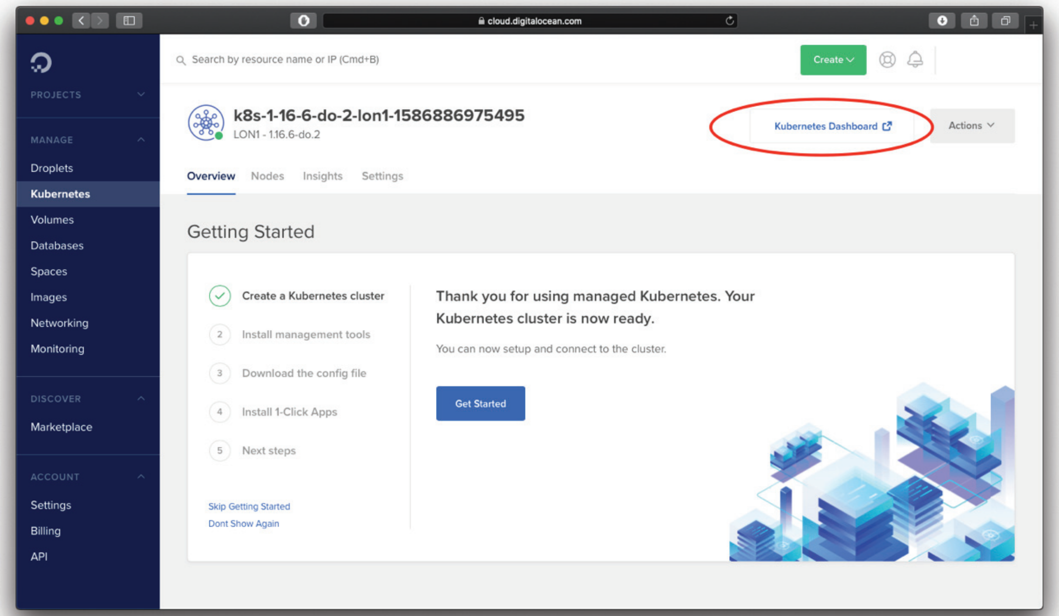
Finally, lines 51 to 57 deploy Apache Flink by applying a set of configuration files including *flink-configuration-configmap.yaml*, *jobmanager-service.yaml*, *jobmanager-deployment.yaml*, and *taskmanager-deployment.yaml*. More specifically, the file named *jobmanager-deployment.yaml* defines the Kubernetes deployment of Flink job managers (default replication number: 1) by pulling the Docker image *flink:latest* and executing the *jobmanager.sh* (i.e. part of the Apache Flink Docker image named *flink:latest*) command and setting up ports for RPC, blob, UI, and liveness probe (default: 6123, 6124, 8081, and 6123, respectively) as well as a volume containing further configuration files such as *flink-conf.yaml*. Further, the file named *jobmanager-service.yaml* defines the Flink job manager Kubernetes service with the same port numbers for RPC, blob, and UI as defined in the Kubernetes Apache Flink job manager deployment described beforehand. Moreover, the file name *taskmanager-deployment.yaml* defines the Kubernetes deployment of a number of Flink task managers (default replication number: 2) by pulling the Docker image *flink:latest*, executing the *taskmanager.sh* (i.e. part of the Apache Flink Docker image named *flink:latest*) command, setting up a RPC port (default: 6122), and defining an equally named volume containing the same configuration files as employed by the job manager deployment. Note that the completion of the Typhon Flink setup may take a few minutes.

From the machine intended to access the Typhon web UI, a proxy to the remote Kubernetes cluster is established by issuing the following command[17]:

```
$ kubectl proxy --port=8080 --kubeconfig=myKubeConfig.kubeconfig
```

4.2.3 Evaluate availability and status of deployments in remote Kubernetes cluster

The next step is to browse to https://cloud.digitalocean.com/kubernetes/, login to account (if required), and select the Kubernetes pool created in heading "Deploy remote Kubernetes pool" as illustrated in Figure 11. Note that although this instruction is based on the Digitalocean cloud provider, other providers will provide similar instructions as to accessing the Kubernetes dashboard.



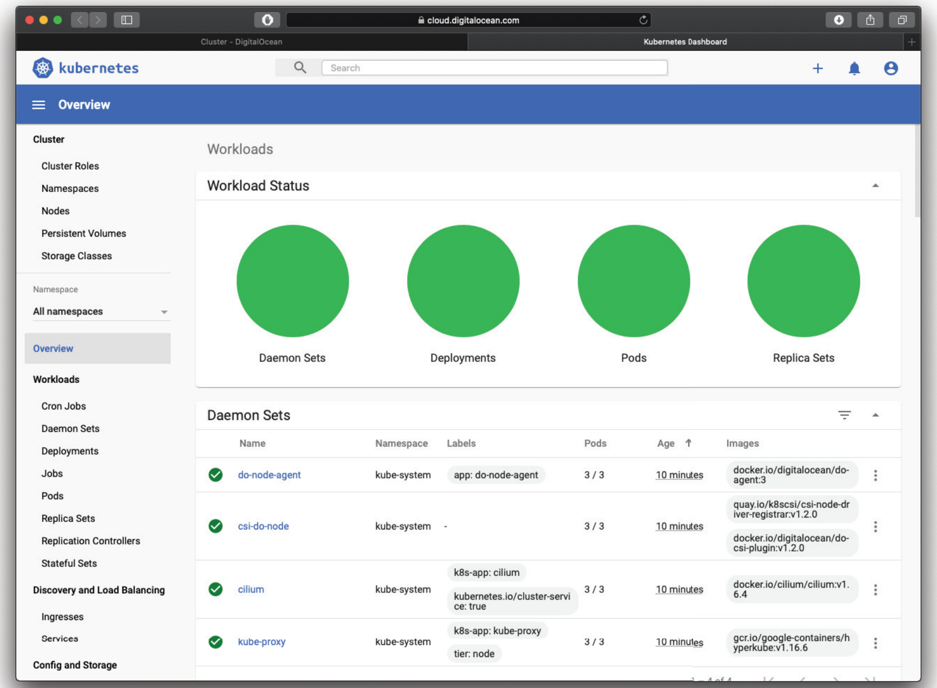
Thank you for using managed Kubernetes. Your Kubernetes cluster is now ready.

You can now setup and connect to the cluster.

Figure 11: Access Kubernetes cluster dashboard web UI.

A successful deployment of Apache Kafka and Flink will result in show the Kubernetes web UI dashboard Workload Overview as illustrated in Figure 12.

[17] Note that establishing a proxy into a Kubernetes cluster is executed in foreground by default (i.e. requiring the issuing terminal/bash process to remain in execution).



Daemon Sets

Name	Namespace	Labels	Pods	Age	Images
do-node-agent	kube-system	app: do-node-agent	3 / 3	10 minutes	docker.io/digitalocean/do-agent:3
csi-do-node	kube-system	-	3 / 3	10 minutes	quay.io/k8scsi/csi-node-driver-registrar:v1.2.0 docker.io/digitalocean/do-csi-plugin:v1.2.0
cilium	kube-system	k8s-app: cilium kubernetes.io/cluster-service: true	3 / 3	10 minutes	docker.io/cilium/cilium:v1.6.4
kube-proxy	kube-system	k8s-app: kube-proxy tier: node	3 / 3	10 minutes	gcr.io/google-containers/hyperkube:v1.16.6

Figure 12: Kubernetes web UI dashboard Workload Overview page.

In particular, the following named daemon sets, deployments, pods, replica sets, stateful sets, services, config maps, and persistent volume claims have been made available:

Deamon Sets:
- ✓ do-node-agent
- ✓ csi-do-node
- ✓ cilium
- ✓ kube-proxy

Deployments:
- ✓ flink-taskmanager
- ✓ flink-jobmanager
- ✓ typhon-cluster-entity-operator
- ✓ stimzi-cluster-operator
- ✓ typhonql-server-deployment
- ✓ relationaldatabase-deployment
- ✓ polystore-ui-deployment
- ✓ polystore-mongo-deployment
- ✓ documentdatabase-deployment

Pods:
- ✓ flink-taskmanager-[UUID1]
- ✓ flink-taskmanager-[UUID2]
- ✓ flink-jobmanager-[UUID3]
- ✓ typhon-cluser-entity-operator-[UUID4]
- ✓ typhon-cluster-kafka-0
- ✓ typhon-cluster-zookeeper-0

- ✓ stimzi-cluster-operator-[UUID5]
- ✓ polystore-mongo-deployment-[UUID6]
- ✓ polystore-ui-deployment-[UUID7]
- ✓ relationaldatabase-deployment-[UUID8]

Replica Sets:
- ✓ flink-taskmanager-[UUID9]
- ✓ flink-jobmanager-[UUID10]
- ✓ typhon-cluster-entity-operator-[UUID11]
- ✓ stimzi-cluster-operator-[UUID12]
- ✓ typhon-polystore-service-deployment-[UUID13]
- ✓ typhonql-server-deployment-[UUID14]
- ✓ relationaldatabase-deployment-[UUID15]
- ✓ polystore-ui-deployment-[UUID16]
- ✓ polystore-mongo-deployment-[UUID17]
- ✓ documentdatabase-deployment-[UUID18]

Stateful Sets:
- ✓ typhon-cluster-kafka
- ✓ typhon-cluster-zookeeper

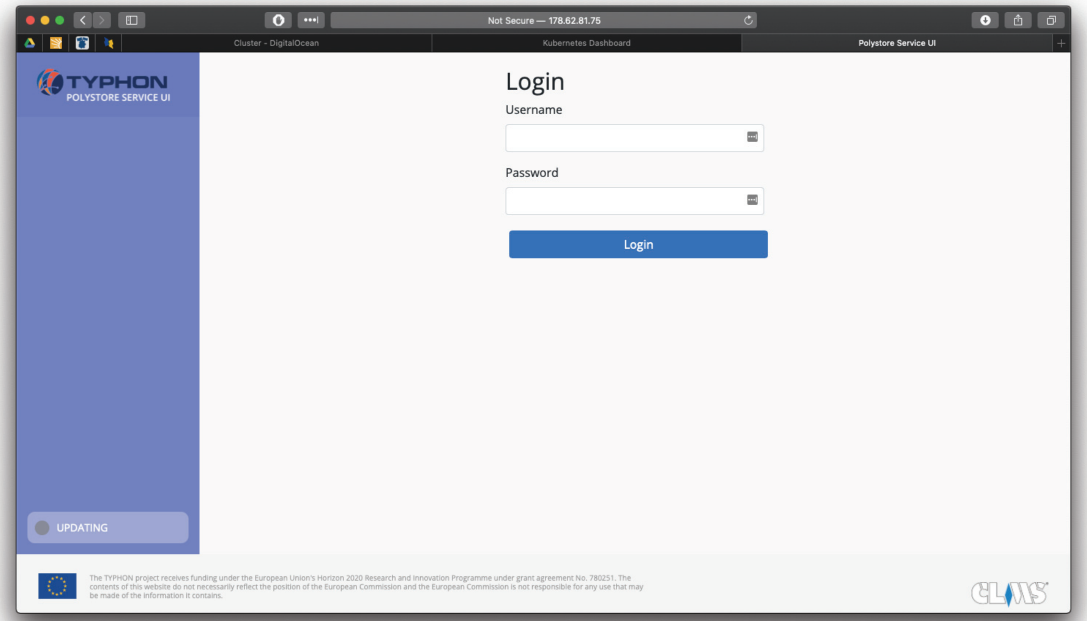
Services:
- ✓ flink-jobmanager
- ✓ typhon-cluster-kafka-external-bootstrap
- ✓ typhon-cluster-kafka-brokers
- ✓ typhon-cluster-kafka-bootstrap
- ✓ typhon-cluster-kafka-0
- ✓ typhon-cluster-zookeeper-client
- ✓ typhon-cluster-zookeeper-nodes
- ✓ documentdatabase
- ✓ typhon-polystore-service
- ✓ relationaldatabase

Config Maps (Kubernetes internal omitted):
- ✓ flink-config
- ✓ typhon-cluster-entity-topic-operator-config
- ✓ typhon-cluster-entity-user-operator-config
- ✓ typhon-cluster-kafka-config
- ✓ typhon-cluster-zookeeper-config

Persistent Volume Claims:
- ✓ data-0-typhon-cluster-kafka-0
- ✓ data-typhon-cluster-zookeeper-0

4.2.4 Access the Typhon Polystore Service web UI

The Typhon Polystore Service web UI can be accessed by browsing to http://HOST:8080 as depicted in Figure 13. The latter URL assumes “HOST” to be replaced by the IP address of the Kubernetes pool load balancer machine (if available; recommended) or the IP address of the Kubernetes pod running the Typhon Polystore web server.



The TYPHON project receives funding under the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 780251. The contents of this website do not necessarily reflect the position of the European Commission and the European Commission is not responsible for any use that may be made of the information it contains.

Figure 13: Typhon Polystore Service web UI login page.

5. EVALUATION

In the previous deliverable (D5.3) we evaluated the fitness of the proposed architecture for developing query-driven analytics scenarios. This was achieved by implementing Alpha Bank's scenarios using our polystore analytics engine. In this deliverable we evaluate the scalability of the analytics architecture using an e-shop simulator developed for this reason. Further scalability evaluation will be conducted by use-case partners during the second evaluation period of TYPHON.

5.1 E-SHOP SIMULATOR

5.1.1 Introduction

The evaluation of the proposed architecture requires ingestion of large volumes of data. In order to internally evaluate our work before the final evaluation by the use-case partners we developed data generators that produce large volumes of synthetic, but realistic, data. Beyond the fact that these generators are able to produce realistic input, they can be configured to generate big volumes of data allowing us to stress-test the proposed architecture.

In our previous work, we presented a generator that produces realistic *database events* for evaluating the fitness of our approach for implementing analytics scenarios for one of the industrial partners (i.e., Alpha Bank). This database event generator was consuming random data produced by a third-party web application (i.e., Mockaroo[18]) to generate database commands. The randomly generated data given to our database event generator are stored in CSV files. The generator developed is consuming these CSV files to generate database events, in the absence of real TYPHON database events at this stage. It is

[18] https://www.mockaroo.com/

important to highlight, that the proposed architecture *is not meant to produce analytics of interest by consuming data already stored in databases or text files (e.g., CSV files).* Proprietary tools (like Tableau [3]) or open-source (like Grafana [26]) are able to do so by using widgets that consume data directly from CSV files, database and many other data sources. By contrast, our architecture supports the extraction of analytics by consuming database events.

In this section we present a second generator that can produce unlimited amounts of synthetic data, helping with scalability testing. It simulates an e-shop application and is able to generate a configurable number of database events. In contrast with the previous one, events are produced without consuming pre-generated data stored in CSV files.

5.1.2 Implementation

As part of the evaluation of the proposed architecture we decided to develop a custom configurable application that simulates the database transactions happening in an e-commerce website. By having such a simulator, we are able to produce synthetic but as realistic as possible database events that can be used as input to test different simple and more complicated analytics scenarios, beyond those set by the project's use-case partners. We are also able to test the correctness of our proposed approach before providing the architecture to our use-case partners for testing. By producing hundreds of thousands of synthetic events we will also be able to stress test the approach and verify the claimed scalability of the architecture and its underlying frameworks (i.e., Apache Flink and Kafka). The e-commerce TyphonML model used in the simulator is shown in Listing 5.

```
entity Review{
      id : string[64]
      content : string[1024]
      product -> Product[1]
      comments :->
Comment."Comment.review"[0..*]
      user -> User[1]
}

entity Product {
      id : string[64]
      name : string[64]
      description : string[1024]
      category -> Category[1]
      reviews :->
Review."Review.product"[0..*]
}

entity Order {
      id : string[64]
      order_date : date
      totalAmount : int
      orderedProducts -> OrderedProduct[0..*]
      users -> User[1]
      paidWith -> CreditCard[1]
}

entity Category {
      id: string[64]
      name: string[32]
}

entity OrderedProduct {
      id : string[64]
      quantity : int
      product -> Product[1]
}

entity User {
      id : string[64]
      name : string[32]
      address :-> Address[1]
      comments :-> Comment[0..*]
      paymentsDetails :-> CreditCard[0..*]
      orders -> Order."Order.users"[0..*]
      reviews ->
Review."Review.product"[0..*]
      basket :-> Basket[1]
}



entity Address {
      id: string[64]
      street: string [256]
      country: string [32]
}

entity Basket {
      id : string[64]
      items :-> BasketProduct[0..*]
}

entity BasketProduct {
      id: string[64]
      quantity : int
      date_added: date
      product :-> Product[1]
}

entity Comment{
      id : string[64]
      content : string[1024]
      review -> Review[1]
      responses :-> Comment[0..*]
}

entity CreditCard{
      id : string[64]
      number : string[32]
      expiryDate : date
}
```

Listing 5: The E-Commerce TyphonML model

The e-shop simulator is based on the notion of "Agents". An agent simulates the behaviour of one type of shopper (i.e., a *User*) in an e-shop. For example, a "Browsing Agent" replicates the behaviour of a user who browses the catalogue of the website before placing an order. In order to be able replicate realistic scenarios where different customers use the e-shop in parallel, each agent implement the Java Runnable interface and as such multiple agents can operate in parallel. The behaviour of the agent should be defined in the implementation of the *run()* method of the Runnable interface. An example of an implementation of an agent is shown in Listing 6. Developers can re-use query generators (e.g., the SelectProductGenerator that creates select DML commands for the Product entity) or build their own. They can also use the *executeQuery(…)* method to execute a query against the polystore or the *createAndPublishPostEvent(…)/createAndPublishPreEvent(…)* to skip the execution of the command against the polystore and create directly a PostEvent/PreEvent object in the analytics queues.

```
public class BrowsingAgent extends Agent implements Runnable {

    @Override
    public void run() {

        final int MAX_NUM_OF_PRODUCTS_TO_BROWSE = 10;

        ExecuteQueries eq = new ExecuteQueries();
        ExecuteQueries.Utils utils = eq.new Utils();

        ...

        SelectProductGenerator spg = new
              SelectProductGenerator();
       Map<String, String> params =
             new HashMap<String, String>();

        Random r = new Random(seed);

        for (int i = 0; i < MAX_NUM_OF_PRODUCTS; i++) {
              params.put("seed", String.valueOf(seed));
              String query = spg.generateQuery(params);
              if (RunSimulator.goThroughPolystore) {
                    utils.executeQuery(query);
              } else {
                    utils.createAndPublishPostEvent(query);
              }
              this.randomSleep(1000, 5000);
        }
    }
  }
```

Listing 6: A "Browsing" Agent

At the beginning, the simulator produces a number of users with random details (e.g., name, Address, etc.). It also produces a number of random products for the e-shop. The number of users and products is configurable using the config.properties file shown in Listing 7. In case users and products were produced before and are already stored in the database, it is possible to skip the creation steps by setting the appropriate configuration flags (i.e., generate_users and generate_products). If these flags are set to *false* the simulator queries the database to retrieve the already created users and products. Also, developers are able to define what type of events (i.e., Pre or PostEvents) that the simultor will produce (see property *topic)*. Finally, in the configuration the number of the different types of agents that the simulator should instantiate (NB.: the total number of agents should match the total number of users as at the beginning each agent is assigned to one user entity).

```
goThroughPolystore = false
generate_users=true
num_of_users=600
generate_products=true
num_of_products=150
seed = 1892
num_of_buyer_agents = 0
num_of_buyer_reviewer_agents = 0
num_of_undecisive_agents = 0
num_of_browsing_agents=0
num_of_browsing_with_comment_agents=600
num_of_buyer_to_pre_agents =0
topic=POST
```

Listing 7: The config.properties file for the e-shop simulator

In order to be able to evaluate the scalability of the proposed architecture, the simulator offers the option to mock the actual execution of the database commands against the polystore and just publish the database event object directly to the analytics queues. This way, the overhead of having to wait for the execution of the actual command against the database in order to produce the Pre/PostEvent object is avoided allowing for the production of thousands of analytics events in the same amount of time. Of course, properties of the Event objects that require the execution of the command (i.e., ResultSet and InvertedResultSet) are left empty thus, the option of not going through the polystore is only used to evaluate scalability rather than the capabilities and features of the proposed architecture. The latter will be fully evaluated through the use-case evaluations conducted by the industrial partners of TYPHON over the 2nd evaluation period of the project.

5.1.3 Authorisation Chain Scalability Evaluation using the Simulator

One of the advantages of using the simulator is the fact that one can skip the execution of the commands against the polystore and be able to produce a big number of events in a short time. This is useful in order to be able to assess the scalability of the proposed architecture. In this section we present the evaluation of the authorisation chain.

In order to test the scalability of the authorisation chain, we produced an increasing number of events which were given as input into the PRE topic. More specifically, users (agents) were simulating the placement of orders in the e-shop. *Insert* commands were generated for the placed orders including details of the credit card used to pay the order. An example TyphonQL query is shown below:

"insert Order {id: "…", order_date: "…", totalAmount: "…", orderedProducts: "[…]", user:…, paidWith: ***CreditCard {id: "4782", number: "6007-2216-3740-9000", expiryDate: "2021-06-25T08:36:13.656"}***

The three authorisation tasks were applying different validation rules on the credit card used. The first task checks the existence of a credit card in the query, the second was checking if the credit card has expired and the third if the credit card number was valid.

The check condition method for all the three tasks was evaluating if the query arrived was an insert order query. The rest (e.g., queries for creation of users, products, OrderedProduct queries, etc.) were also evaluated by the "checkCondition(…)" method of each of the authorisation tasks but as the condition was not satisfied were ended up in the AUTH queue being approved.

As described in Section 3.2, if an event is rejected by one authorisation task, it is not passed to the following task(s) in the chain but is directed automatically to the AUTH queue as rejected. In the simulator the agents were producing orders that had always a credit card assigned to them, so they were approved by the first task. From those, half (50%) were having an expired credit card attached to them thus, they were rejected from the second task. Those passed successfully from the second task have a 50% chance of having an invalid credit card number. Following this pattern, we increased the variability as some of the events will be passing the whole chain, while some will be rejected earlier.

The chain was deployed in a cluster consisting of three machines; one acting as the master and the rest two as the workers[19]. In Flink clusters, the master node is responsible for orchestrating the process and the communication between the workers. In our experiment, the master was also hosting the relevant KAFKA topics (PRE and AUTH). We restricted the Flink deployment to allocate and use only 8GB of the available 64GB for each worker. During the execution of the analytics code in the Flink cluster we measured the CPU and memory consumption of the workers and master. We also kept track of the time needed for the cluster to calculate the results for the input given.

Table 4 summarises the configuration parameters for each of the five execution scenarios of increasing complexity.

Table 4: Summary of input to the authorisation chain scalability experiment

Users	Products	Total Events
600	150	92,557
1200	150	184,957
1800	150	277,357
2400	150	369,757
3000	150	462,157

Figure 14 shows the total execution time (in seconds) for processing all the event and posting the (rejected/approved) PreEvent in the AUTH queue. The graph shows linear scalability which confirms our expectation as the analytics architecture is built atop tools such as Apache Flink and does not add any bottleneck.

[19] All the machines were identical and had the following specifications: AMD Opteron(tm) Processor 4226 – 6-cores @ 2.7Ghz, 4x16GB DD3 1066MHz RAM

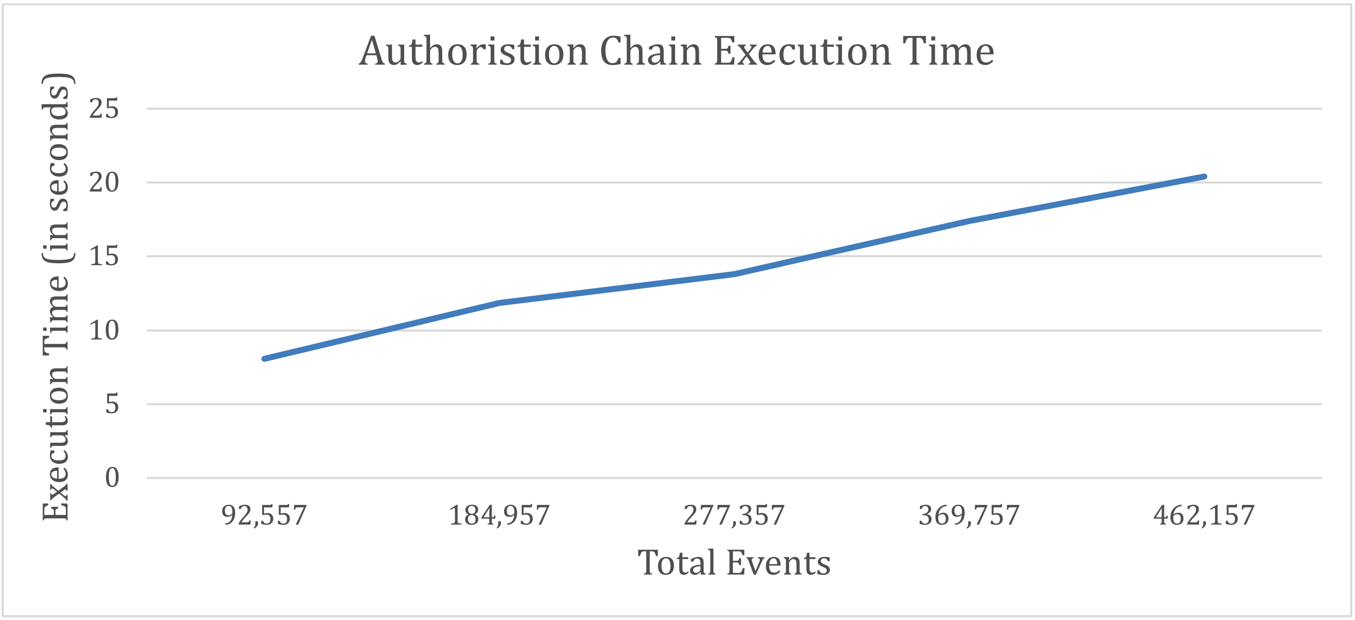


Figure 14: Total execution time for the authorisation chain experiment.

The average memory consumption and CPU utilisation for the workers is shown in Figure 15 and Figure 16. In this scenario, both workers requiring increasing amount of memory for each scenario from the operating system while the CPU utilisation is between 60-70%. The CPU utilisation and memory consumption is similar across the cluster's workers which shows even distribution of work.

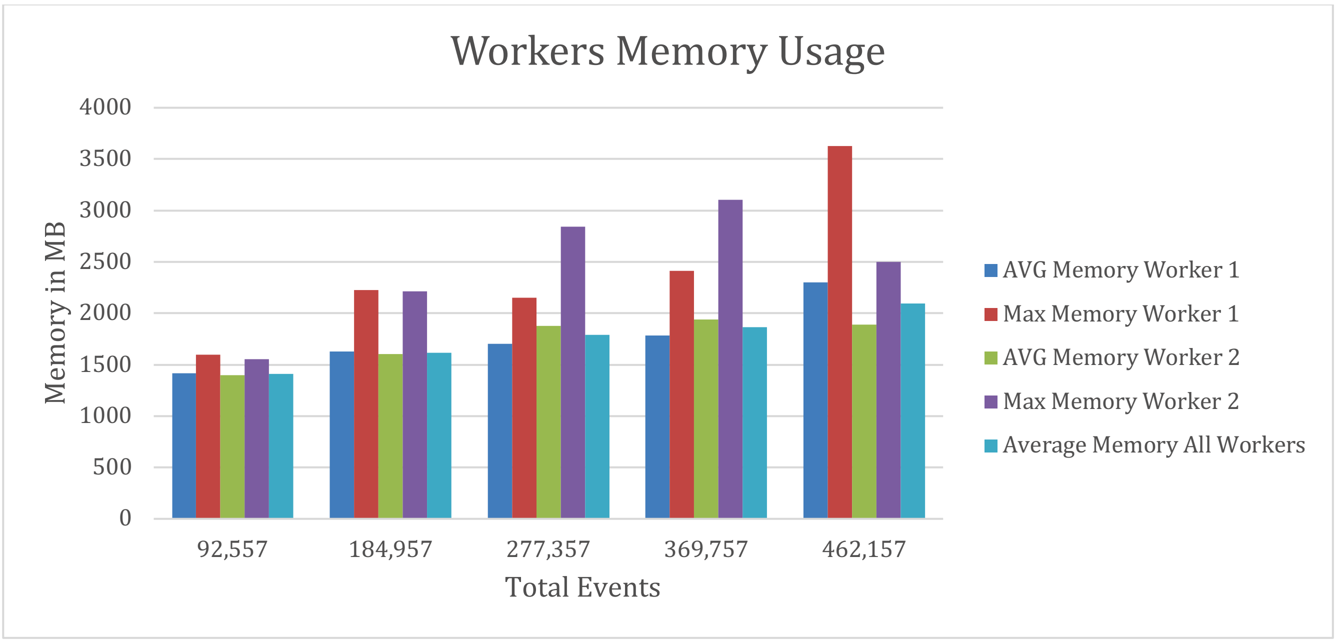


Figure 15: Workers' memory consumption in the authorisation chain scalability experiment

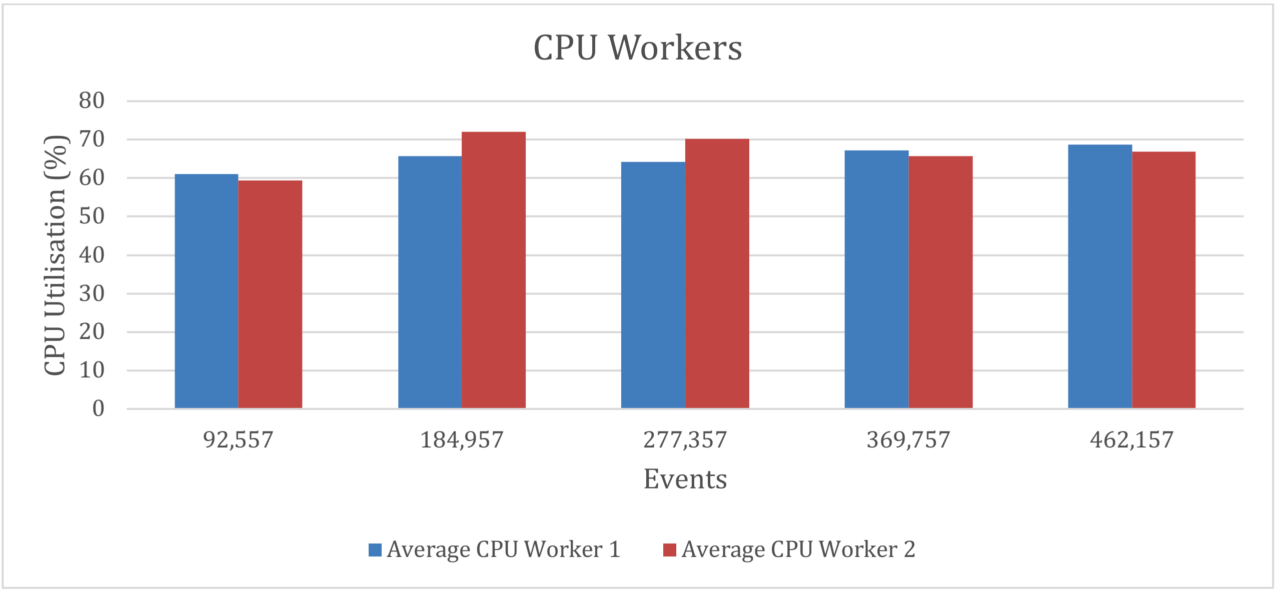


Figure 16: Workers' CPU utilisation for the authorisation chain scalability experiment

The master node's average memory increases steadily and averages between 450 and 650MB. The CPU is around 50% for all the experiments. This is justified by the fact that the master node in this experiment was hosting the KAFKA queue and more significantly the AUTH topic in which the workers were publishing the results. Thus, the CPU utilisaton is justified by having the master node writing these events in the AUTH queue.

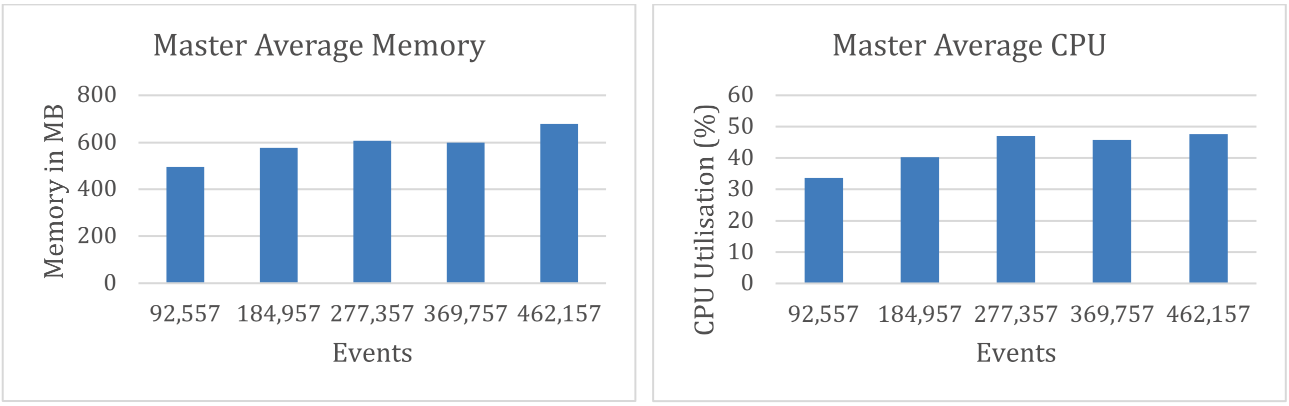


Figure 17: Master's average memory and CPU utilisation for the authorisation chain scalability experiment

5.1.4 Analytics Scalability Evaluation using the Simulator

In this section we present an experimental evaluation of the scalability of the analytics architecture using the said simulator.

We implemented a scenario in which a list of the top products that users browsed within a specific time window is produced. The simulator was instantiated with a varying number of users each of which was randomly navigating a number of products. Navigation of the catalogue has a result of generating one *Select* query each time a product page was visited. An example generated query is the following:

from Product p select p where p.@id = "…."

The implemented analytics scenario, consumes only those events (i.e., *select* events on the table *Product*) and calculates the top visited products in a specific time window (e.g., every 2-3 days). Such a scenario firstly, demonstrates the novelty of the proposed approach of extracting analytics through database events. Information like this is not normally stored in databases. Thus, calculating such analytics is not possible by querying the databases. Services such as Google Analytics[20] can pull this information by running analytics scripts at the application layer (e.g., cookies or scripts running at the client side/browser). This way requires adding analytics code to the e-shop application code that will track the user behaviour which leads in mixing business logic with analytics logic.

Secondly, this scenario includes a good variety of Flink operators in order to produce the desired results, thus we are confident that it includes operators that business analysts will uses to define most of their analytics scenarios. It includes mapping and filtering operators, timestamp assigners, time windowing, key grouping and aggregators.

Finally, it relies on using Flink's built-in operators for extracting analytics (i.e., aggregators on keyed groups). We believe that this is important as results that demonstrate non-linear scalability might not be due to the architecture not being scalable, but the algorithms used in the analytics scenarios not being scalable. For example, if in the scenario a clustering algorithm, that has non-polynomial complexity, is used, then the evaluation will reveal a total exponential scalability that might not be due to the analytics architecture, but due to the algorithm used.

As it might be the case that users in real deployments might exploit such an analytics scenario to promote their products (i.e., by visiting their product page repeatedly) and in order to test the *slots* feature introduced in Section 2, we were amending the PreEvent object linked to the PostEvent object that our simulator generated with the id of the user that requested the execution of the command. Such information can be taken for example from the query where the session user id is passed as a comment to the produced query.

In order to test the scalability of the architecture, we produced an increasing number of events which were given as input into the analytics architecture. The analytics code was deployed in some cluster cluster configuration as described in the evaluation of the authorisation chain (see Section 5.1.3). During the execution of the analytics code in the Flink cluster, we measured the CPU and memory consumption of the master and the workers. We also kept track of the time needed for the cluster to calculate the results for the input given. The cluster and the queues were reset before each run. Measurements of the passive memory of the system before deploying the cluster were taken in order to calculate the actual impact of deploying and running the architecture in the cluster for each scenario.

Table 5 summarises the configuration parameters for each of the five execution scenarios of increasing complexity.

[20] https://analytics.google.com/analytics/web/#/

Table 5: Summary of input to the analytics scenario scalability experiment

Users	Products	Related Events	Total Events
600	150	90,000	92,557
1200	150	180,000	184,957
1800	150	270,000	277,357
2400	150	360,000	369,757
3000	150	450,000	462,157

The simulator ran in five different configurations. In each of them a number of users (column "Users") and products (column "Products") were generated initially (along with their appropriate *insert* commands). As the number of the related to the analytics scenario events (i.e., the aforementioned "select" events) are affected by the total number of users, we increased in each run the number of users. The column "Related Events" shows the total number of events that the analytics engine consumed to produce the results for the top-browsed scenario. The column "Total Events" shows the total number of events in the Kafka POST queue and includes other commands necessary to construct the simulated scenario (e.g., insert commands for generating mock users, products, etc). The latter, also passed into the analytics scenario but were filtered in the first operator.

The time needed for our architecture to produce the results for the five simulated scenarios is shown in Figure 18. The graph shows linear scalability which confirms our expectation as the analytics architecture is built atop scalable tools and does not add any bottleneck.

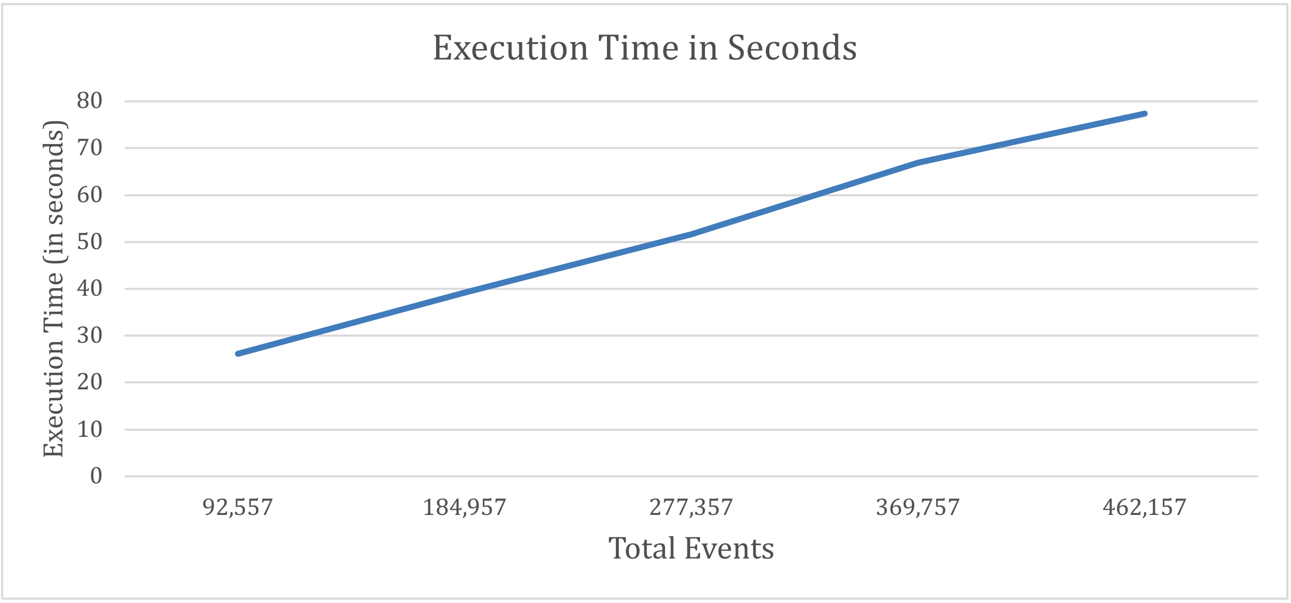


Figure 18: Execution time for the five simulations in the analytics scalability experiment

The average workers' average and max memory consumption for each worker for the five simulated scenarios are shown in Figure 19 while the average CPU utilisation is shown in Figure 20. The workers are using above 80% of the available processing power on average across the five different scalability scenarios. Also, the JVM is claiming all the necessary memory (especially in the last 4 of the five scenarios) but is not running out of memory which is explained by the Java garbage collector replacing unnecessary memory when needed. The load balance is equally split among the workers both in terms of CPU utilisation and memory usage which demonstrates that the workload is shared equally in the cluster.

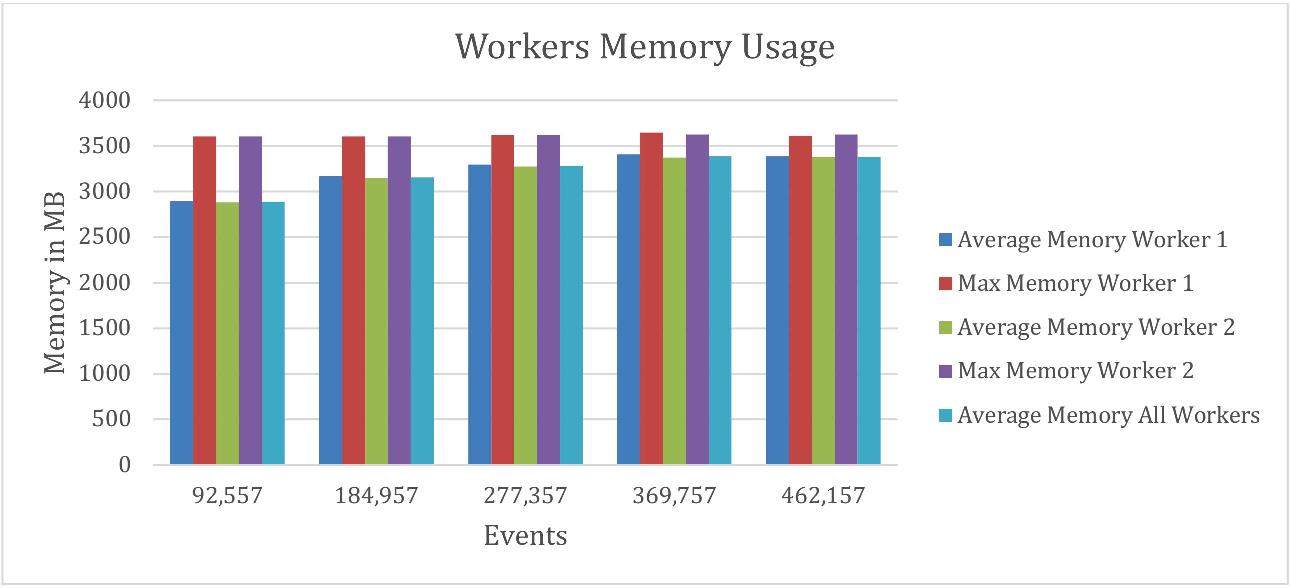


Figure 19: Workers' memory consumption in the analytics scalability experiment

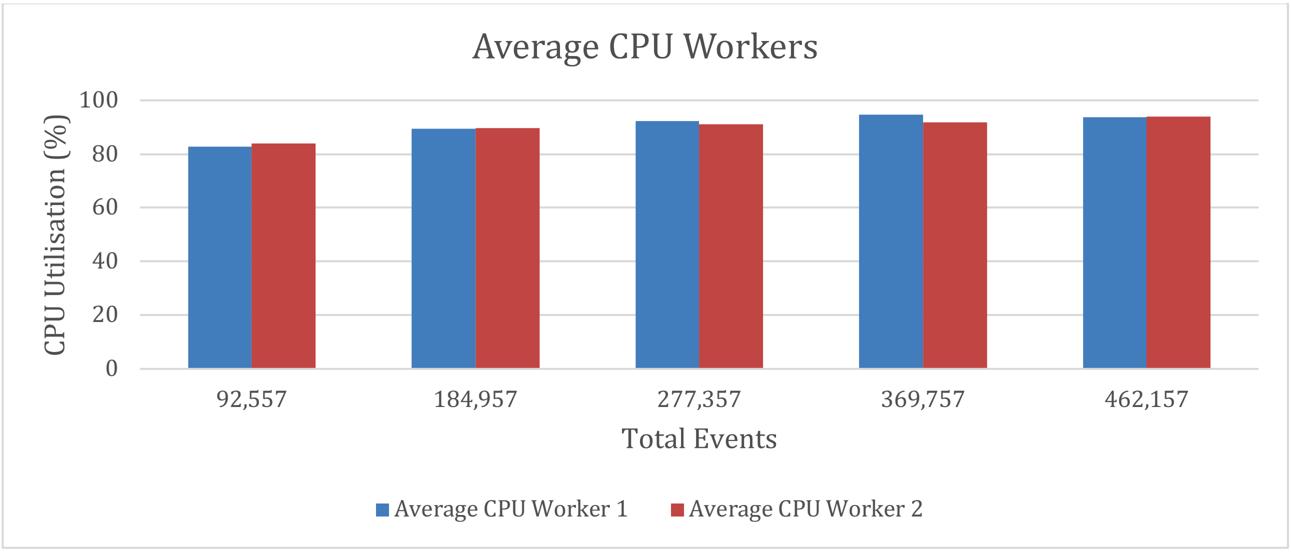


Figure 20: Workers' CPU use in the analytics scalability experiment

Finally, the CPU utilisation and memory consumption for the master node are given in Figure 21. Both remain quite low as in this experiment the workers are only reading from the POST queue hosted in the master and thus the master is not required to perform any writes to the KAFKA queue.

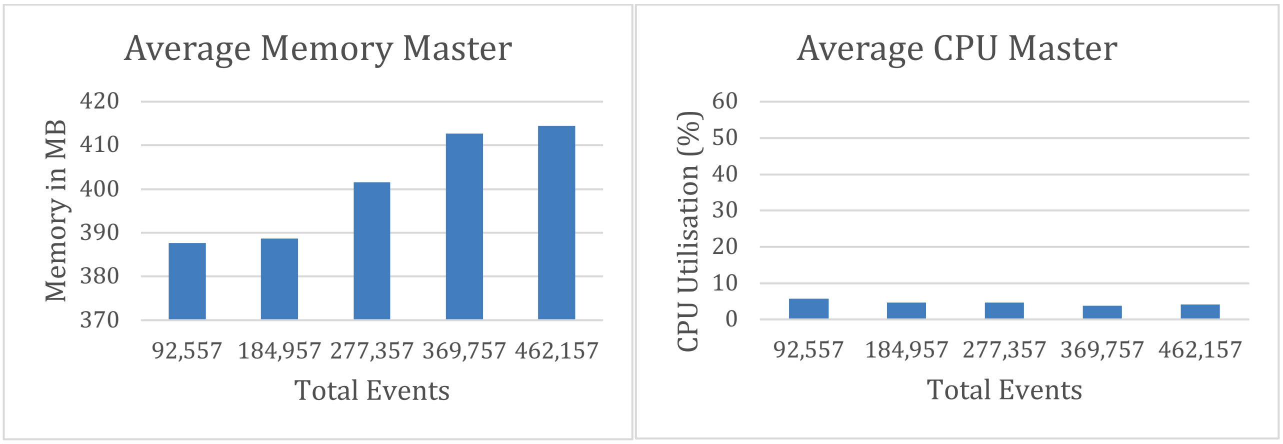


Figure 21: Average master's memory and CPU utilisation for the analytics scenario scalability experiment

6. Conclusions And Future Work

In this document the final event publishing and monitoring architecture and the data event metamodel that describes the structure of the events stored in the analytics messaging queues were presented. Beyond the post-execution analytics architecture, in this document we discuss in detail the pre-execution authorisation mechanism. The deployment facilities based on Docker and Kubernetes are also presented. Finally, the scalability of both the authorisation and the analytics components of the proposed architecture is evaluated.

In the future, it would be of interest to explore if authorisation tasks can be re-arranged automatically in the chain. Tasks that reject a higher proportion of events or require less time to execute would be useful to be positioned earlier in the chain. Machine learning algorithms can be used to identify the most efficient chains based on different features among those described (i.e., execution time and rejection rate).

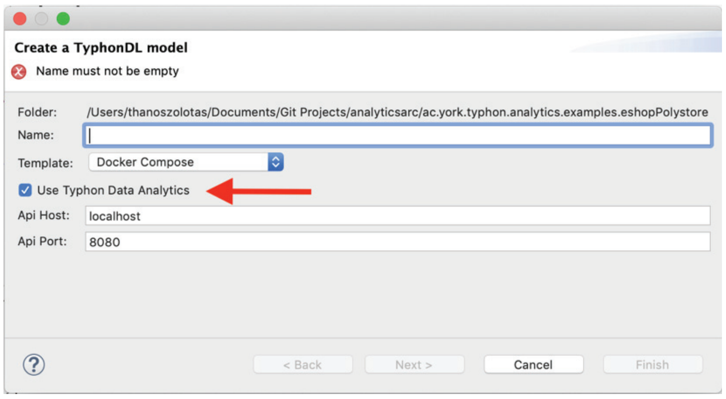
Finally, applying limits to the number of returned results stored in the resultSet and invertedResultSet is of interest. Beyond using approaches similar to LIMIT in relational databases, another approach would be that of returning a desired subset of the results using different criteria (e.g., entities created within a specific time period).

APPENDIX A

ANALYTICS HOW-TO GUIDE

Prerequisites

i) This guide assumes that you have already installed all the necessary tools to create and run a Polystore (e.g., TyphonML, TyphonDL, etc.).
ii) You need to make sure that you have those updated (from their respective Eclipse update sites and by doing a docker-compose pull) to their latest version.
iii) Start by creating the polystore as described in the appropriate guide. Make sure that in the step of the DL wizard you have checked the "Use Typhon Data Analytics" option, as shown in the image below.
iv) Run the Polystore.


Create a TyphonDL model
Name must not be empty

Folder: /Users/thanoszolotas/Documents/Git Projects/analyticsarc/ac.york.typhon.analytics.examples.eshopPolystore
Name:
Template: Docker Compose
☑ Use Typhon Data Analytics
Api Host: localhost
Api Port: 8080

Figure 22: TyphonDL wizard that enables analytics

Work with the Analytics Component

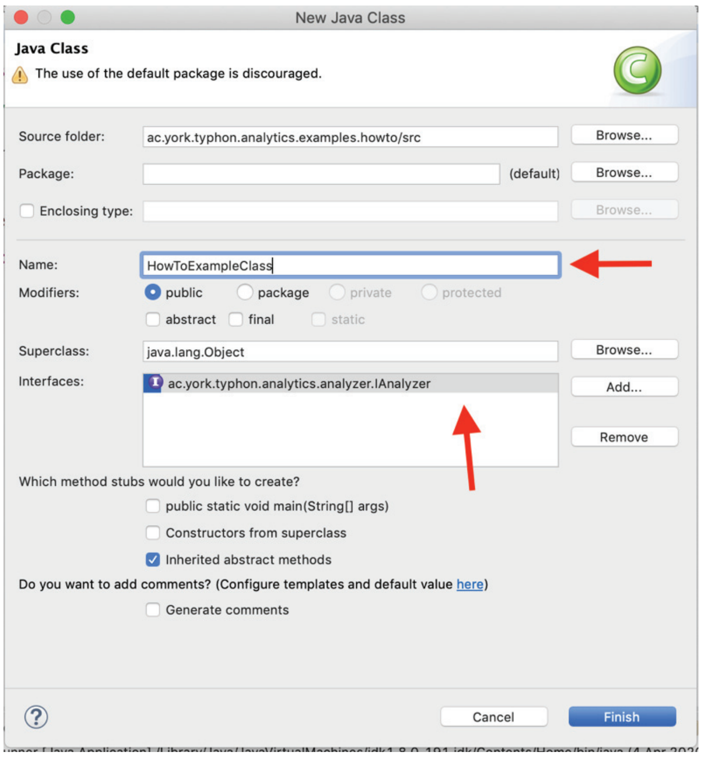
i) Download (if you do not have already done) Eclipse with Java support from here: https://www.eclipse.org/downloads/packages/release/2020-03/r/eclipse-ide-java-developers
ii) Download the analytics zip file from here:
https://drive.google.com/file/d/15tyOF9yKnVsl0JxbbB6eQUvONA1rH04d/view?usp=sharing
iii) Unzip and import the two projects (ac.york.typhon.analytics.examples.howto and ac.york.typhon.analytics) into Eclipse by going to File → Import → Existing Projects into Workspace
iv) The ac.york.typhon.analytics.examples.howto project is an example project that has a simple analytics scenario (TestAnalyticScenario class) and a runner (AnalyticsRunner class) in it.
v) If you navigate to the pom.xml file you will see that it has a dependency to the ac.york.typhon.analytics project.
vi) You can either use this project to test analytic scenarios or you can create another Maven project that has the same dependency (to the ac.york.typhon.analytics)
vii) The ac.york.typhon.analytics project includes the analytics infrastructure. You do not need to do anything with it. It should just be included as a dependency when you create a new Analytics project, as described above.

Please note: The generated docker compose defines port 29092 for external (outside Docker) access to the Kafka queue and port 9092 for internal (inside Docker) access. As this guide describes how to write analytics in your local IDE, the configuration is set to access port 29092. If you want to export the jar and run it inside Docker, then you need to open the "resources/typhonAnalyticsConfig.properties" file and set the port in line 12 to 9092.

Write Analytics

i) Create a new maven project that has a dependency on the ac.york.typhon.analytics project. Of course you can instead use the example project (ac.york.typhon.analytics.examples.howto) you have imported.

ii) Create a new class (right click on the src folder → New → Class) that implements the "IAnalyzer" interface. If you use the example project you will see that such a class already exists (named "TestAnalyticScenario").



New Java Class

Java Class

⚠ The use of the default package is discouraged.

Source folder: ac.york.typhon.analytics.examples.howto/src

Package: (default)

☐ Enclosing type:

Name: HowToExampleClass

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces: ac.york.typhon.analytics.analyzer.IAnalyzer

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

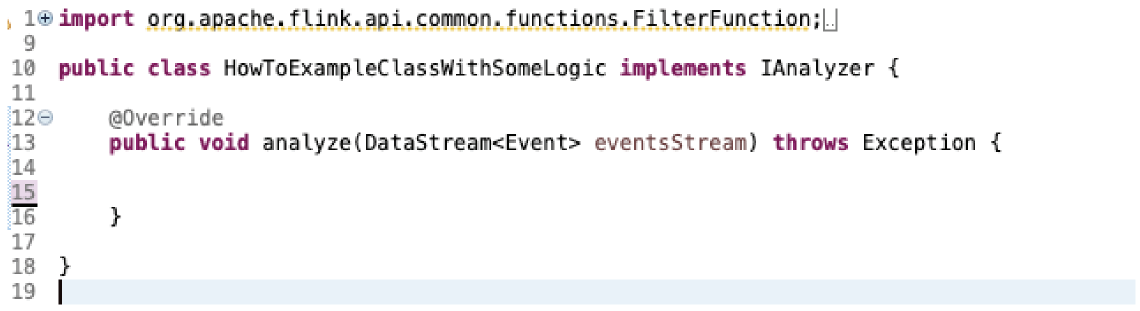
☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 23: Create a new Analytics Scenario

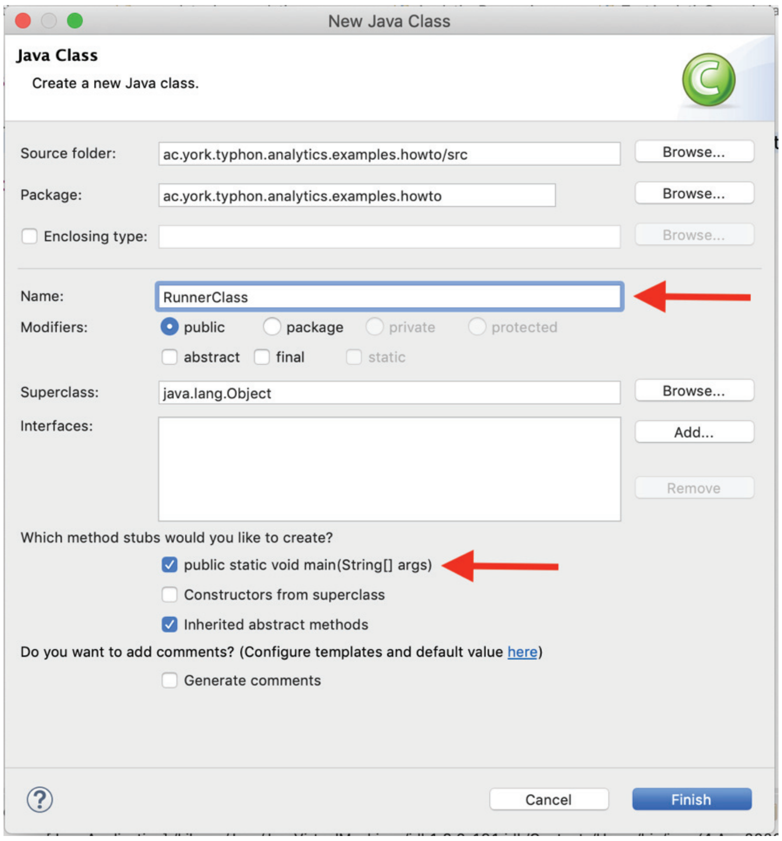
iii) The new class will include the method you need to implement called "analyze (DataStream<Event> eventsStream). The eventsStream parameter are the PostEvent objects arriving to the POST queue of the analytics architecture. You need to write Flink code to consume them and produce analytics of interest (more on this later).



```
import org.apache.flink.api.common.functions.FilterFunction;

public class HowToExampleClassWithSomeLogic implements IAnalyzer {

    @Override
    public void analyze(DataStream<Event> eventsStream) throws Exception {


    }

}
```

Figure 24: The "analyze" method that needs to be implemented

iv) In order to run the analytics code you need to create a main class which calls the classes that include analytics. Create a new class (right click on the src folder → New → Class), give it a name (e.g., RunnerClass) and include a main method in it (If you use the example project this is the AnalyticsRunner class.). You need to create such a class only once for each analytic scenario.



New Java Class

Java Class

Create a new Java class.

Source folder: ac.york.typhon.analytics.examples.howto/src

Package: ac.york.typhon.analytics.examples.howto

Enclosing type:

Name: RunnerClass

Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[✓ public static void main(String[] args)]

Constructors from superclass

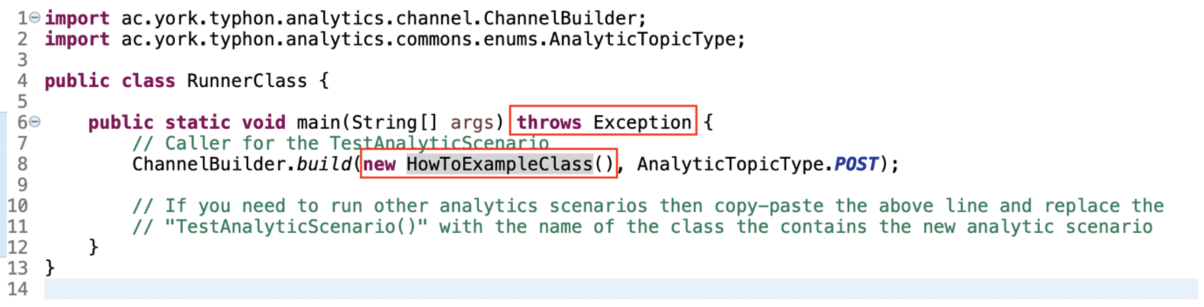
[✓ Inherited abstract methods]

Do you want to add comments? (Configure templates and default value here)

Generate comments

Figure 25: A runner class for the analytics scenario

a. You need to make a call to the class that includes an analytics scenario using the ChannelBuilder.build(…) method as shown in the code below. This method takes as a first parameter the name of the class that includes analytics code (e.g., HowToExampleClass) and as a second parameter the name of the Kafka topic from which events should consumed. This should be AnalyticTopicType.POST always when writing analytics for Typhon Post Events.
b. Remember to declare that your main class throws an exception (or surround the ChannelBuilder methods with try…catch statements)



```java
import ac.york.typhon.analytics.channel.ChannelBuilder;
import ac.york.typhon.analytics.commons.enums.AnalyticTopicType;

public class RunnerClass {

    public static void main(String[] args) throws Exception {
        // Caller for the TestAnalyticScenario
        ChannelBuilder.build(new HowToExampleClass(), AnalyticTopicType.POST);

        // If you need to run other analytics scenarios then copy-paste the above line and replace the
        // "TestAnalyticScenario()" with the name of the class the contains the new analytic scenario
    }
}
```

Figure 26: The runner class implementation

v) Run this main method as a Java Application. Your analytics code inside the HowToExampleClass will start consuming PostEvents as these arrive in the Polystore. As the analyze method's body is empty, this will do nothing. More on how to write analytics code is described in the next section.

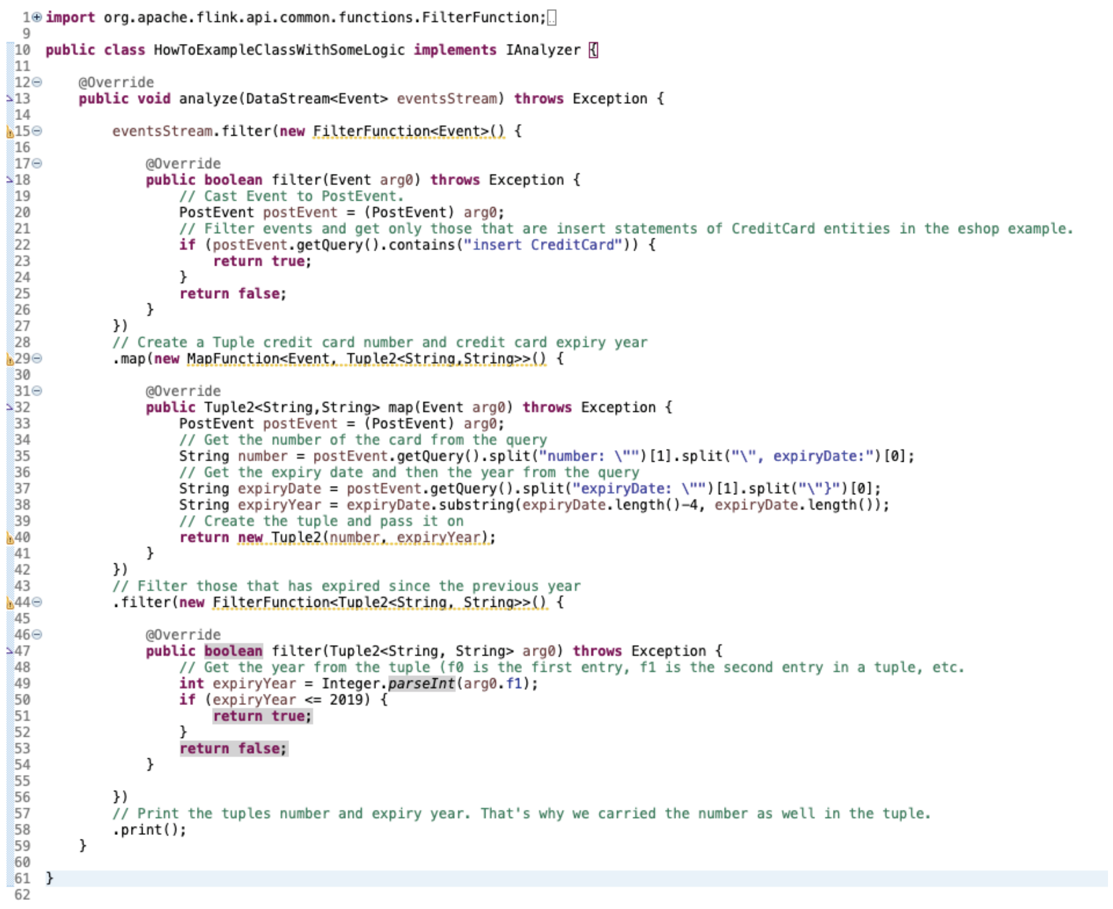
IMPORTANT!!! Post events in Typhon are created every time a TyphonQL query is executed. Thus, your code will produce results, if and only if you start using the polystore and execute some TyphonQL queries.

Writing Analytics Code with Flink

Flink is a distributed execution framework. By using its **operators** Flink can easily distribute you code without requiring user's input/configuration. The goal of this guide is not to train people on writing Flink code. There are plenty of resources on this online. The basic idea is that Flink works with streams (in the context of the analytics component). Steams as the name suggests, provide continuous real-time input to your programs. In the analytics component, the stream of events is the "eventsStream" parameter is the analyse method. This is configured to automatically consume all the events coming to the POST queue.

To consume streams using Flink, one should use Flink Operators. A comprehensive list of all the available Flink operators is available here: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/

This should be the starting point of anyone trying to use Flink as they contain a brief description and an example of how to make them work. You will find yourself mostly having to use the filter and map function (the first filters events based on a condition, the second is used to transform objects to other forms). Experiment with these 2 first and then you can proceed to more complicated operators. Below is a simple example that consumes Typhon PostEvents and produces at the end a list of the credit card numbers that have expired. You can find the code into the "HowToExampleClassWithSomeLogic" class of the "ac.york.typhon.analytics.examples.howto" project. The example is based on an ECommerce polystore example.



```java
import org.apache.flink.api.common.functions.FilterFunction;

public class HowToExampleClassWithSomeLogic implements IAnalyzer {

    @Override
    public void analyze(DataStream<Event> eventsStream) throws Exception {

        eventsStream.filter(new FilterFunction<Event>() {

            @Override
            public boolean filter(Event arg0) throws Exception {
                // Cast Event to PostEvent.
                PostEvent postEvent = (PostEvent) arg0;
                // Filter events and get only those that are insert statements of CreditCard entities in the eshop example.
                if (postEvent.getQuery().contains("insert CreditCard")) {
                    return true;
                }
                return false;
            }
        })
        // Create a Tuple credit card number and credit card expiry year
        .map(new MapFunction<Event, Tuple2<String,String>>() {

            @Override
            public Tuple2<String,String> map(Event arg0) throws Exception {
                PostEvent postEvent = (PostEvent) arg0;
                // Get the number of the card from the query
                String number = postEvent.getQuery().split("number: \"")[1].split("\", expiryDate:")[0];
                // Get the expiry date and then the year from the query
                String expiryDate = postEvent.getQuery().split("expiryDate: \"")[1].split("\"}")[0];
                String expiryYear = expiryDate.substring(expiryDate.length()-4, expiryDate.length());
                // Create the tuple and pass it on
                return new Tuple2(number, expiryYear);
            }
        })
        // Filter those that has expired since the previous year
        .filter(new FilterFunction<Tuple2<String, String>>() {

            @Override
            public boolean filter(Tuple2<String, String> arg0) throws Exception {
                // Get the year from the tuple (f0 is the first entry, f1 is the second entry in a tuple, etc.
                int expiryYear = Integer.parseInt(arg0.f1);
                if (expiryYear <= 2019) {
                    return true;
                }
                return false;
            }

        })
        // Print the tuples number and expiry year. That's why we carried the number as well in the tuple.
        .print();
    }

}
```

Figure 27: An example analytics scenario

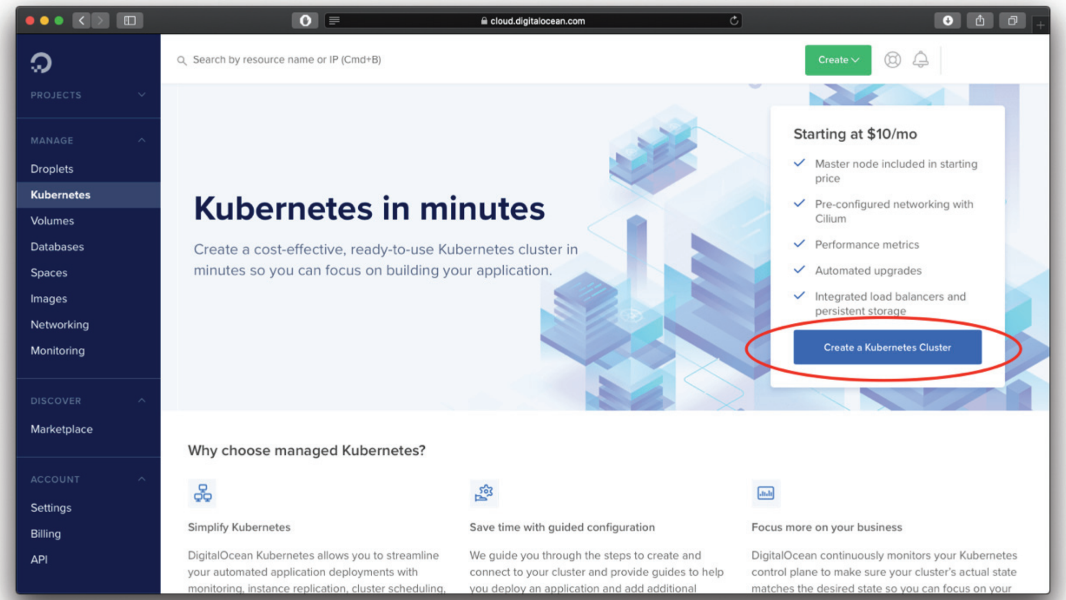
APPENDIX B

Installation of kubectl command-line tool

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl
-s                                https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/darwin/amd64/kubectl"
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
```

Deploy remote Kubernetes pool

Browse to https://cloud.digitalocean.com/kubernetes/, login to account (if required), and select "Create a Kubernetes Cluster" as illustrated in Figure 28 and Figure 29.



Kubernetes in minutes

Create a cost-effective, ready-to-use Kubernetes cluster in minutes so you can focus on building your application.

Starting at $10/mo

- ✓ Master node included in starting price
- ✓ Pre-configured networking with Cilium
- ✓ Performance metrics
- ✓ Automated upgrades
- ✓ Integrated load balancers and persistent storage

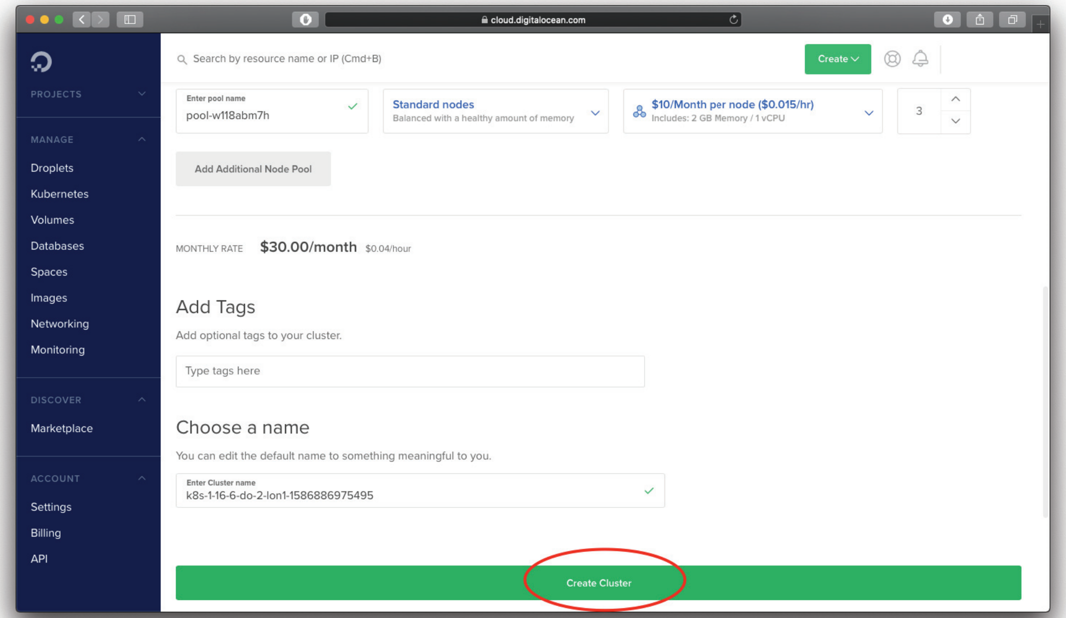
Why choose managed Kubernetes?

Simplify Kubernetes
DigitalOcean Kubernetes allows you to streamline your automated application deployments with monitoring, instance replication, cluster scheduling,

Save time with guided configuration
We guide you through the steps to create and connect to your cluster and provide guides to help you deploy an application and add additional

Focus more on your business
DigitalOcean continuously monitors your Kubernetes control plane to make sure your cluster's actual state matches the desired state so you can focus on your

Figure 28: Deploy remote Kubernetes pool (step 1 of 2).



Enter pool name: pool-w118abm7h
Standard nodes — Balanced with a healthy amount of memory
$10/Month per node ($0.015/hr) — Includes: 2 GB Memory / 1 vCPU
3

MONTHLY RATE $30.00/month $0.04/hour

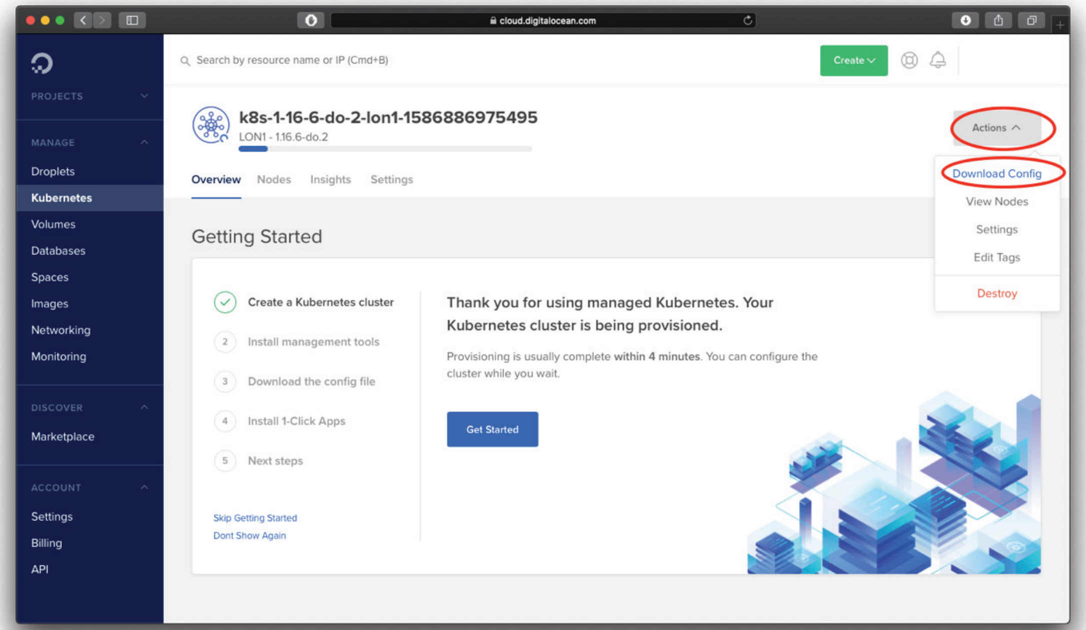
Add Tags
Add optional tags to your cluster.
Type tags here

Choose a name
You can edit the default name to something meaningful to you.
Enter Cluster name: k8s-1-16-6-do-2-lon1-1586886975495

Figure 29: Deploy remote Kubernetes pool (step 2 of 2).

Download cloud provider Kubernetes configuration

Browse to https://cloud.digitalocean.com/kubernetes/clusters/, login (if required), and download the configuration of (an existing Kubernetes cluster) by selecting "Actions" and then "Download Config" as illustrated in Figure 30.



Thank you for using managed Kubernetes. Your Kubernetes cluster is being provisioned.

Provisioning is usually complete **within 4 minutes**. You can configure the cluster while you wait.

Figure 30: Download cloud provider Kubernetes cluster YAML-based configuration file.

BIBLIOGRAPHY

[1] J. Kreps, N. Narkhede and J. Rao, “Kafka: A distributed messaging system for log processing,” in *Proceedings of the NetDB*, 2011.

[2] The Apache Software Foundation, “Powerd by Flink,” 2019. [Online]. Available: https://flink.apache.org/poweredby.html. [Accessed 28 June 2019].

[3] TABLEAU SOFTWARE, LLC, “Business Intelligence and Analytics Software,” [Online]. Available: https://www.tableau.com/. [Accessed May 2020].

[4] E. Friedman and K. Tzoumas, Introduction to Apache Flink: stream processing for real time and beyond, O'Reilly Media, Inc., 2016.

[5] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao and V. Y. Ye, “Building LinkedIn's Real-time Activity Data Pipeline,” *IEEE Data Engineering,* vol. 35, no. 2, pp. 33-45, 2012.

[6] S. Boschi and G. Santomaggio, RabbitMQ cookbook, Packt Publishing Ltd, 2013.

[7] The Apache Software Foundation, “Flexible & Powerful Open Source Multi-Protocol Messaging,” 2019. [Online]. Available: https://activemq.apache.org/. [Accessed 2019 June 20].

[8] P. Dobbelaere and K. S. Esmaili, “Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2017.

[9] The Apache Software Foundation, “What is Apache Flink?,” 2019. [Online]. Available: https://flink.apache.org/flink-architecture.html. [Accessed 20 June 2019].

[10] The Apache Software Foundation, “Apache Spark,” 2019. [Online]. Available: https://spark.apache.org/. [Accessed 20 June 2019].

[11] The Apache Software Foundation, “Apache Storm,” 2015. [Online]. Available: https://storm.apache.org/. [Accessed 20 June 2019].

[12] T. Chen and X. Huang, “Fight Peak Data Traffic On 11.11: The Secrets of Alibaba Stream Computing,” 27 12 2017. [Online]. Available: https://102.alibaba.com/detail?id=35. [Accessed 28 June 2019].

[13] S. Rooney, P. Urbanetz, C. Giblin, D. Bauer, F. Froese, L. Garcés-Erice and S. Tomić, “Kafka: the Database Inverted, but Not Garbled or Compromised,” in *IEEE International Conference on Big Data (Big Data)*, 2019.

[14] Confluent.io, “Kafka Connect,” 2020 June 30. [Online]. Available: https://docs.confluent.io/current/connect/index.html. [Accessed 2020 June 30].

[15] ZenDesk, “Maxwell's Daemon,” [Online]. Available: https://maxwells-daemon.io/. [Accessed 1 July 2020].

[16] Oracle Corporation, “Real-time access to realtime Information, Oracle White Paper,” Redwood Shores, California, 2015.

[17] Debezium Community, “Debezium,” 2020. [Online]. Available: https://debezium.io/. [Accessed 23 June 2020].

[18] Confluent, Inc, “Confluent: Apache Kafka & Event Streaming Platform for Enterprise,” 2020. [Online]. Available: https://www.confluent.io/. [Accessed 01 July 2020].

[19] Confluent Inc, 2020. [Online]. Available: https://assets.confluent.io/m/40dd744b4c79d1e8/original/20190820-DS-Confluent_Platform.pdf?_ga=2.184200038.1585514032.1593583811-120261339.1587742459&_gac=1.50595419.1593584512.CjwKCAjwxev3BRBBEiwAiB_PWI7usRtLc9G241yJyqbUzL8ID2YguhzzosqBExdOENVr9cHCJCsBhB. [Accessed 2020 July 01].

[20] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. Pollack, “The Epsilon Generation Language,” in *European Conference on Model Driven Architecture-Foundations and Applications*, 2008.

[21] G. Shlyuger, “Big Data Analytics approach for security data processing,” 2018.

[22] J. Widom and S. Ceri, Active database systems: Triggers and rules for advanced database processing, Morgan Kaufmann, 1996.

[23] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing,* pp. 81--84, 2014.

[24] P. Hunt, M. Konar, F. Junqueira and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems.,” in *USENIX annual technical conference*, Boston, MA, USA, 2010.

[25] Docker Inc., “Docker Hub,” 2019. [Online]. Available: https://hub.docker.com/. [Accessed 18 June 2019].

[26] Grafana Labs, “Grafana: The open-source Observability Platform,” [Online]. Available: https://grafana.com/. [Accessed May 2020].

[27] Docker Inc., “Docker Compose,” 2019. [Online]. Available: https://docs.docker.com/compose/. [Accessed 18 June 2019].

[28] YAML.org, “YAML: YAML Ain't Markup Language,” 2019. [Online]. Available: https://yaml.org/. [Accessed 18 June 2019].

[29] J. Baier, Getting Started with Kubernetes, Packt Publishing Ltd, 2015.

[30] The Apache Software Foundation, “Welcome to Apache Zookeeper,” 2019. [Online]. Available: https://zookeeper.apache.org/. [Accessed 20 June 2019].

[31] D. Goldin, S. Srinivasa and V. Srikanti, “Active databases as information systems,” in *International Database Engineering and Applications Symposium*, 2004.