

Seminararbeit
im Studiengang
Master Informatik

Dokumentation der praktischen Umsetzung des Streaming Systems Praktikum

GitHub-Repository: <https://github.com/larimei/Streaming-Systems>

Referent : Prof. Dr. Bernhard Hollunder

Vorgelegt am : 28.01.2024

Vorgelegt von : Lara Meister

Matrikelnummer: 274416

Jonathan Reißer

Matrikelnummer: 275736

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Quellcodeverzeichnis	III
Abbildungsverzeichnis	V
1 Einleitung	1
1.1 Aufgabenstellung	1
1.2 Verwendete Technologien	2
2 Event-Source-Anwendung	3
2.1 Aufgabe 1: Event Sourcing	3
2.2 Aufgabe 2: Erweiterte Event Sourcing-Funktionalität	8
2.3 Aufgabe 3: Integration von JMS für Event Store	9
2.4 Aufgabe 4: Nutzung von Apache Kafka im Event Store	11
3 Verkehrsüberwachung	13
3.1 Aufgabe 5: Datengenerierung	13
3.2 Aufgabe 6: Skalierung	16
3.3 Aufgabe 7: Apache Beam	17
3.4 Aufgabe 8: EPL und Esper	20
4 Aufgabe 10: Vergleich der Technologien	23
4.1 Bewertung der Technologien	23
4.2 Persönliche Einschätzung	31
Eidesstattliche Erklärung	33

Quellcodeverzeichnis

Codeauschnitt 2.1: Vector Klasse	4
Codeauschnitt 2.2: Aufgabe 1 Integration Test	5
Codeauschnitt 2.3: Aufgabe 1 Unit Test	6
Codeauschnitt 2.4: Moving Item Event Interface	9
Codeauschnitt 2.5: Aufgabe 3 Modularisierte Main-Methode	10
Codeauschnitt 2.6: Aufgabe 4 Printausgabe	11
Codeauschnitt 3.1: Aufgabe 5 Unit Test Conversion	14
Codeauschnitt 3.2: Aufgabe 5 Unit Test Filtering	15
Codeauschnitt 3.3: Aufgabe 7 Pipeline Generierung	17
Codeauschnitt 3.4: Aufgabe 7 PAssert Unit Test	19
Codeauschnitt 3.5: Aufgabe 8 Esper Traffic Jam Test	21

Abbildungsverzeichnis

Abbildung 1:	Output Aufgabe 1	7
Abbildung 2:	Output Aufgabe 2	8
Abbildung 3:	Output Aufgabe 3	10
Abbildung 4:	Output Aufgabe 4	11
Abbildung 5:	Offset Explorer Aufgabe 4	12
Abbildung 6:	Output Aufgabe 5	15
Abbildung 7:	Offset Explorer Aufgabe 6	16
Abbildung 8:	Output Vergleich Aufgabe 7 und Aufgabe 5	18
Abbildung 9:	Output Aufgabe 8	22

1 Einleitung

1.1 Aufgabenstellung

Für die Erstellung der ersten Anwendung liegt der Fokus auf der Entwicklung einer Event Sourcing-Anwendung. Ziel ist es, eine robuste und skalierbare Plattform zu erstellen, die Positions- und Zustandsänderungen von Objekten effizient verwalten kann. Die Anwendung soll Objekte, die das 'MovingItem'-Interface implementieren, durch verschiedene Befehle erstellen, löschen, die Position ändern und den Wert aktualisieren können. Hierbei wird auf Event Sourcing-Prinzipien zurückgegriffen, bei denen Ereignisse als fundamentale Elemente des Systems gelten. Es wird ein Domänenmodell entworfen, das aus Events wie 'MovingItem-CreatedEvent' und 'MovingItemValueChangedEvent' besteht, um die Anwendung zu führen und zu validieren.

Die Anwendung soll eine klare Trennung zwischen der Write Side, die Befehle verarbeitet und Ereignisse erzeugt, und der Read Side, die Query-Schnittstellen bereitstellt, gewährleisten. Der Event Store übernimmt die dauerhafte Speicherung von Ereignissen.

In der zweiten Anwendung, steht die Entwicklung eines Testdatengenerators im Kontext der Verkehrsüberwachung im Fokus. Dieser Generator erstellt Datensätze, die verschiedene Messwerte von Sensoren zu unterschiedlichen Zeitpunkten simulieren. Die Konfigurierbarkeit des Generators ermöglicht es, die Anzahl der Sensoren, die Anzahl der Messwerte, die Taktung und weitere Parameter anzupassen.

Ziel ist es, eine Apache Kafka-basierte Plattform zu entwickeln, die diese Datensätze in ein Topic mit einer Partition einstellt. Eine darauf aufbauende Anwendung wird implementiert, um die Durchschnittsgeschwindigkeit für jeden Sensor in zeitlichen Fenstern zu ermitteln. Diese Berechnungen ermöglichen die Analyse des zeitlichen Verlaufs der Durchschnittsgeschwindigkeit an einzelnen Messstellen sowie die Ermittlung von Durchschnittsgeschwindigkeiten auf Streckenabschnitten. Die Integration von Datenverarbeitungstechnologien wie Apache Beam und Complex Event Processing (CEP) unterstützt dabei eine umfassende Analyse der generierten Datenströme.

1.2 Verwendete Technologien

Die Event Sourcing-Anwendung wurde vollständig in Kotlin entwickelt, wodurch viele Vorteile der Programmiersprache genutzt werden konnten. Kotlin, als eine auf der JVM basierende Sprache, ermöglicht eine nahtlose Integration in bestehende Java-Umgebungen. Die Entscheidung für Kotlin resultierte aus seiner kompakten Syntax, seinem Schutz vor Nullpointer-Ausnahmen und der verbesserten Lesbarkeit des Codes im Vergleich zu Java.

Der Einsatz von Apache Kafka bildete das Rückgrat für die Datenverarbeitung in der Verkehrsüberwachungsanwendung. Durch Kafka wurden die generierten Datensätze in ein zentrales Topic eingestellt, um eine robuste und skalierbare Datenverteilung zu gewährleisten.

Für die Verarbeitung der kontinuierlichen Datenströme wurde Apache Beam eingesetzt. Dabei erlaubte die Anwendung von Kotlin als Programmiersprache in Verbindung mit Beam, eine saubere und deklarative Definition von Datenverarbeitungs-Pipelines. Dies erleichterte nicht nur die Implementierung, sondern ermöglichte auch eine bessere Wartbarkeit der Codebasis.

Complex Event Processing (CEP) wurde durch die Esper-Bibliothek realisiert. Kotlin ermöglichte auch hier eine klare und präzise Beschreibung von Ereignismustern und -abfragen.

Insgesamt wurden diese Technologien in Kombination mit Kotlin ausgewählt, um eine effiziente, skalierbare und leicht wartbare Lösung für die gestellten Praktikumsaufgaben zu schaffen.

2 Event-Source-Anwendung

2.1 Aufgabe 1: Event Sourcing

Im Folgenden wird die Entwicklung einer Kotlin-Anwendung für Event Sourcing zur Verwaltung von Positions- und Zustandsänderungen von Objekten beschrieben. Die Anwendung soll Kommandos für die Erstellung, Löschung und Änderung von Objekten verarbeiten. Dabei ist eine klare Trennung zwischen der Write-Side und der Read-Side zu gewährleisten.

Im Programm werden mehrere Bewegungsereignisse für simulierte Items generiert und durch die Anwendung verschiedener Komponenten verarbeitet. Zunächst werden Items erstellt und zufällig bewegt, indem das CommandImpl-Objekt verwendet wird, das wiederum auf CommandHandler, EventStoreImpl und einer Ereigniswarteschlange basiert. Nachdem mehrere Bewegungen durchgeführt wurden, projiziert der ProjectionHandler diese Ereignisse auf ein Query-Modell, das dann vom QueryHandler genutzt wird, um die gesammelten Daten zu durchsuchen und auszugeben, wobei Details zu einzelnen Items und deren Bewegungen dargestellt werden.

Im Projekt ist die Klassenstruktur gemäß der in der Vorlesung vorgestellten Architektur des Bibliotheksbeispiels organisiert. Dabei sind die Klassen in verschiedene Packages aufgeteilt, um eine klare Trennung der Verantwortlichkeiten zu gewährleisten. Das Package event beinhaltet Klassen, die sich mit der Verarbeitung von Ereignissen beschäftigen, was für das Event Sourcing entscheidend ist. Im projection Package sind Klassen angesiedelt, die sich um die Projektion dieser Ereignisse auf unser Query-Modell kümmern. Das read Package umfasst Klassen, die für das Lesen und Abrufen von Daten zuständig sind, während das write Package die Command-Klassen beinhaltet, welche die Geschäftslogik und Schreiboperationen ausführen. Zusätzlich befinden sich die Main-Klasse und die Vector-Klasse auf der obersten Ebene, um einen zentralen Einstiegspunkt und grundlegende Funktionalitäten für das gesamte System bereitzustellen. Diese Strukturierung gewährleistet eine klare und effiziente Organisation des Codes, die den Prinzipien der in der Vorlesung besprochenen Architektur entspricht.

Für die Darstellung räumlicher Positionen wurde eine spezielle Vector-Klasse implementiert. Die Vector-Klasse bietet Typsicherheit und Klarheit, indem sie explizit dreidimensionale Vektoren repräsentiert, im Gegensatz zu einem allgemeinen Integer-Array. Sie ermöglicht die Definition spezifischer Vektoroperationen wie Addition. Durch Kapselung und Unveränderlichkeit der privaten Felder wird die Sicherheit und Integrität der Vektordaten gewährleistet. Zudem

verbessert die Möglichkeit, Methoden wie `toString` und `equals` anzupassen, die Lesbarkeit und Funktionalität des Codes im Vergleich zu einem einfachen Array.

```
1 class Vector(private val x: Int = 0, private val y: Int = 0, private val z: Int = 0) {  
2     fun add(other: Vector): Vector {  
3         val newX = this.x + other.x  
4         val newY = this.y + other.y  
5         val newZ = this.z + other.z  
6         return Vector(newX, newY, newZ)  
7     }  
8  
9     override fun toString(): String {  
10        return "($x, $y, $z)"  
11    }  
12  
13    override fun equals(other: Any?): Boolean {  
14        return other is Vector && other.x == x && other.y == y && other.z == z  
15    }  
16  
17    override fun hashCode(): Int {  
18        var result = x  
19        result = 31 * result + y  
20        result = 31 * result + z  
21        return result  
22    }  
23  
24    fun copy(): Vector {  
25        return Vector(x, y, z)  
26    }  
27 }
```

Codeauschnitt 2.1: Vector Klasse

Ein wesentlicher Fokus lag auf der Kapselung durch Interfaces, was zur Strukturierung des Codes beitrug und die Wartbarkeit sowie Erweiterbarkeit des Systems verbesserte. Die Verwendung von Interfaces unterstützte zudem das Prinzip der Dependency Injection, was eine flexible und effiziente Verwaltung von Abhängigkeiten innerhalb der Anwendung ermöglichte. Zudem ermöglicht Dependency Injection eine saubere Trennung zwischen der Testumgebung und der eigentlichen Geschäftslogik, indem Abhängigkeiten wie `EventStore`, `CommandHandler` und `ProjectionHandler` bei Bedarf durch Mocks oder Stubs ersetzt werden können. Dies gewährleistet, dass die Tests für `SystemIntegrationTest` und `CommandImplTest` sich auf das spezifische Verhalten und die Interaktionen der Komponenten konzentrieren können, ohne von externen Systemen oder komplexen Initialisierungsprozeduren abhängig zu sein.

Ein wesentlicher Bestandteil der Strategie war die Implementierung von Integrationstests sowie Unit-Tests. Diese waren entscheidend, um die Funktionalität der einzelnen Komponenten und deren Zusammenspiel zu überprüfen. Durch diese systematische Teststrategie konnte die Korrektheit und Vollständigkeit der Lösung sichergestellt werden.

Im Test „moveItem reflects in QueryModel after projection“ des SystemIntegrationTest-Pakets wird überprüft, ob ein Bewegungsereignis (MoveItem) korrekt im Query-Modell reflektiert wird, nachdem es durch den ProjectionHandler verarbeitet wurde. Der Test stellt sicher, dass nach dem Ausführen eines Bewegungsbefehls die neue Position und die Anzahl der Bewegungen des Objekts im Query-Modell aktualisiert und korrekt sind.

```
1 class SystemIntegrationTest {
2
3     @Test
4     fun 'moveItem reflects in QueryModel after projection'() {
5
6         val eventQueue = LinkedBlockingQueue<MovingItemEvent>()
7         val eventStore = EventStoreImpl(eventQueue)
8         val domainItems = mutableMapOf<String, MovingItemImpl>()
9         val commandHandler = CommandHandler(eventStore, domainItems)
10        val commandImpl = CommandImpl(commandHandler)
11        val queryModel = mutableMapOf<String, MovingItemDTO>()
12        val projectionHandler = ProjectionHandler(eventStore, queryModel)
13        val itemId = "item1"
14        commandImpl.createItem(itemId)
15
16        val moveVector = Vector(1, 1, 1)
17        commandImpl.moveItem(itemId, moveVector)
18        projectionHandler.projectEvents()
19
20        val dto = queryModel[itemId]
21        assertNotNull(dto)
22        assertEquals(moveVector, dto?.location)
23        assertEquals(1, dto?.numberOfMoves)
24    }
25 }
```

Codeauschnitt 2.2: Aufgabe 1 Integration Test

Der Test „moveItem should save MovingItemMovedEvent to event store“ im CommandImplTest-Paket prüft, ob das Ausführen eines Bewegungsbefehls (MoveItem) durch CommandImpl ein ItemMovedEvent im EventStore speichert. Hier wird sichergestellt, dass das Event korrekt erstellt wird und die Informationen wie Item-ID und Bewegungsvektor korrekt im EventStore gespeichert sind.

```
1 class CommandImplTest {
2
3     private lateinit var eventStore: EventStore
4     private lateinit var commandHandler: CommandHandler
5     private lateinit var commandImpl: CommandImpl
6     private val domainItems = mutableMapOf<String, MovingItemImpl>()
7     private val eventQueue = LinkedBlockingQueue<MovingItemEvent>()
8
9     @BeforeEach
10    fun setUp() {
11        eventStore = EventStoreImpl(eventQueue)
12        commandHandler = CommandHandler(eventStore, domainItems)
13        commandImpl = CommandImpl(commandHandler)
14    }
15
16    @Test
17    fun 'moveItem should save MovingItemMovedEvent to event store'() {
18
19        val itemId = "5"
20        val moveVector = Vector(1, 1, 1)
21
22        commandImpl.createItem(itemId)
23        commandImpl.moveItem(itemId, moveVector)
24
25        val event = eventStore.getAllEvents().last()
26        assertTrue(event is ItemMovedEvent)
27        assertEquals(itemId, (event as ItemMovedEvent).id)
28        assertEquals(moveVector, event.vector)
29    }
30 }
```

Codeauschnitt 2.3: Aufgabe 1 Unit Test

Im Bereich des Domänenmodells wurden einfache Überprüfungen im Command-Bereich durchgeführt, um die Integrität der Daten zu gewährleisten. Des Weiteren wurden Immutables verwendet, um die Unveränderlichkeit bestimmter Datenstrukturen sicherzustellen, was wiederum die Zuverlässigkeit und Vorhersehbarkeit der Anwendung erhöhte.

Abschließend wurden die Vorteile von Kotlin im Vergleich zu Java genutzt. Kotlin bietet mehrere fortschrittliche Funktionen wie Data-Klassen, verbesserte Listenverwaltung und höhere Ordnungsfunktionen, die die Entwicklung effizienter und angenehmer gestalteten. Diese Eigenschaften von Kotlin trugen maßgeblich zur Leistungsfähigkeit und Erweiterbarkeit der Lösung bei.

Insgesamt führte diese Kombination aus sorgfältiger Planung, der Anwendung fortschrittlicher Programmierkonzepte und einer systematischen Teststrategie zu einer robusten, skalierbaren und gut funktionierenden Lösung.

```
Item: itemAtPos, Location: (4, 3, 2), Moves: 1, Value: 0
Item: 1, Location: (3, 2, 4), Moves: 1, Value: 0
Item: 2, Location: (8, 7, 10), Moves: 4, Value: 0
Item: 3, Location: (7, 2, 2), Moves: 2, Value: 0
Item: 4, Location: (7, 2, 9), Moves: 3, Value: 0
Item: 5, Location: (3, 5, 12), Moves: 4, Value: 0
Item: 6, Location: (2, 10, 4), Moves: 3, Value: 0
Specific item details: MovingItemDTO(name=3, location=(7, 2, 2), numberOfMoves=2, value=0)
These items are at position (4, 3, 2):
Item: itemAtPos, Location: (4, 3, 2), Moves: 1, Value: 0
```

Abbildung 1: Output Aufgabe 1

2.2 Aufgabe 2: Erweiterte Event Sourcing-Funktionalität

Die Anwendung soll um Funktionen erweitert werden, die eine automatische Löschung von Objekten nach einer festgelegten Anzahl von Bewegungen ermöglichen. Außerdem ist es wichtig, bei Positionsänderungen auf Kollisionen zu prüfen.

Eine zentrale Frage war, ob bei einer Positionsbewegung ein Vektor mit dem Wert 0 ein Bewegungsevent ausgeführt wird oder ob dies ignoriert wird, da eine Bewegung um einen Nullvektor praktisch keine Bewegung darstellt. Die Positionsbewegung um 0 wurde dennoch miteinbezogen, da es dennoch wichtig sein könnte, wenn eine Bewegung hätte stattfinden sollen.

Ein wesentlicher Punkt war die Entscheidung zwischen der Anpassung des Domänenmodells und der Erweiterung der Query-Schnittstelle. Gemäß den Prinzipien von CQRS und DDD empfahl sich die Anpassung des Domänenmodells, um die Logik zur automatischen Löschung und Kollisionserkennung zu integrieren. CQRS sieht die Trennung von Befehlen und Abfragen vor, und die beschriebene Funktionalität deutete auf eine Zustandsänderung hin, was typischerweise durch Befehle abgebildet wird. Die Erweiterung des CommandHandlers, der die Verarbeitung von Befehlen steuert, erwies sich als geeignete Stelle, um das Domänenmodell zu erweitern.

Die Verwendung der Vektorklasse erwies sich erneut als vorteilhaft, insbesondere für einen präzisen Vergleich und den Zugriff auf die Add-Funktion zum Hinzufügen von Vektoren.

Die Verwendung von High-Order Functions ermöglichte eine elegante und funktionale Verarbeitung von Listen und Collections. Im Kontext der Anwendung wurden sie speziell für die Filterung von Daten verwendet, was die Lesbarkeit des Codes verbesserte.

```
Move should be executed for the 20th time - Item will be deleted instead
Item does not exist
Already one item at this position - will be deleted
Already one item at this position - will be deleted
Item: 1, Location: (1, 5, 2), Moves: 2, Value: 0
Item: 2, Location: (2, 2, 4), Moves: 1, Value: 0
Item: 3, Location: (6, 3, 10), Moves: 4, Value: 0
Item: 4, Location: (7, 1, 3), Moves: 3, Value: 0
Item: 5, Location: (9, 6, 4), Moves: 4, Value: 0
Item: 6, Location: (6, 13, 11), Moves: 4, Value: 0
Item: 10, Location: (8, 8, 8), Moves: 1, Value: 0
Specific item details: MovingItemDTO(name=3, location=(6, 3, 10), numberOfMoves=4, value=0)
```

Abbildung 2: Output Aufgabe 2

2.3 Aufgabe 3: Integration von JMS für Event Store

Der Event Store soll auf JMS (Java Message Service) umgestellt werden. Der Command Handler fungiert als JMS-Nachrichtenproduzent und eine separate Projektion sollte die erzeugten JMS-Ereignisse konsumieren und das Query Model aktualisieren.

Ein zentraler Aspekt der Lösung war die Verwendung der Active MQ Connection Factory, welche als Singleton implementiert wurde. Diese Entscheidung ermöglichte eine effiziente und wiederverwendbare Verbindungsaufnahme zu Active MQ, was für die Stabilität und Leistung der Anwendung entscheidend war.

Für die Übertragung von JSON-Daten wurde die Bibliothek Jackson eingesetzt. Jackson ist bekannt für seine effiziente und flexible Verarbeitung von JSON, was die Handhabung von Datenübertragungen und -umwandlungen innerhalb der Anwendung erheblich vereinfachte und optimierte.

Das Interface der erstellten Events hat zusätzlich ein Attribut timestamp erhalten, welches vor dem Versenden des Produzenten initialisiert wird. Beim Konsumenten wird die Zeit berechnet, welche das Event vom Produzent zum Konsument gebraucht hat. Alle berechneten Zeiten werden dann in einer globalen mutable List eingetragen, um die durchschnittliche Zeit aller Ereignisse am Ende des Programms auszugeben.

```
1 @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "
   event_type")
2 @JsonSubTypes(
3     JsonSubTypes.Type(value = ItemCreatedEvent::class, name = "ItemCreatedEvent"),
4     JsonSubTypes.Type(value = ItemDeletedEvent::class, name = "ItemDeletedEvent"),
5     JsonSubTypes.Type(value = ItemMovedEvent::class, name = "ItemMovedEvent"),
6     JsonSubTypes.Type(value = ItemValueChangedEvent::class, name = "ItemValueChangedEvent"),
7 )
8 interface MovingItemEvent {
9     val id: String
10    val timestamp: Long
11 }
```

Codeauschnitt 2.4: Moving Item Event Interface

Ein weiterer wichtiger Punkt war die Verwendung der bisherigen Eventstore-Schnittstelle. Diese bereits etablierte Schnittstelle zu nutzen, half dabei, die Kontinuität und Kompatibilität mit bestehenden Systemkomponenten zu gewährleisten, während gleichzeitig neue Funktionen integriert wurden.

Um die Konfiguration der Anwendung zu vereinfachen und zu zentralisieren, wurden globale App-Konfigurationskonstanten eingeführt. Diese globalen Konstanten trugen dazu bei, die Wartbarkeit des Codes zu verbessern und die Konfigurationseinstellungen leichter zugänglich und änderbar zu machen.

Schließlich wurde eine Modularisierung der Main-Funktion vorgenommen. Aufgrund der zunehmenden Komplexität und der Vielzahl an Funktionalitäten, die in der Main-Funktion implementiert waren, wurde diese in kleinere, übersichtlichere Module aufgeteilt. Diese Modularisierung verbesserte nicht nur die Lesbarkeit des Codes, sondern erleichterte auch das Testen und die Wartung der einzelnen Funktionsbereiche.

```

1 fun main() {
2     val connectionProducer = ActiveMQConnectionFactory.instance.createConnection().apply {
3         start() }
4     val connectionConsumer = ActiveMQConnectionFactory.instance.createConnection().apply {
5         start() }
6     try {
7         val commandImpl = initializeCommandSide(connectionProducer)
8         val (queryHandler, queryModel) = initializeQuerySide()
9         val projectionHandler = ProjectionHandler(queryModel)
10        startConsumer(connectionConsumer, projectionHandler)
11
12        processItems(commandImpl, ITEM_COUNT, MAX_MOVES)
13
14        printQueryResults(queryHandler)
15    } finally {
16        connectionConsumer.close()
17        connectionProducer.close()
18    }
19 }

```

Codeauschnitt 2.5: Aufgabe 3 Modularisierte Main-Methode

Zusammenfassend lässt sich sagen, dass die Kombination aus einer effizienten Verbindungshandhabung, einer leistungsstarken JSON-Verarbeitung, dem geschickten Einsatz bestehender Schnittstellen, einer zentralisierten Konfigurationsverwaltung und einer klaren Modularisierung zu einer leistungsstarken und gut strukturierten Lösung führte.

```

Event ItemMovedEvent(id=7, timestamp=1706305402876, vector=(2, 0, 3)) needed 9 ms to get to the consumer
Event ItemMovedEvent(id=7, timestamp=1706305402877, vector=(3, 0, 4)) needed 8 ms to get to the consumer
Event ItemMovedEvent(id=7, timestamp=1706305402878, vector=(3, 1, 3)) needed 7 ms to get to the consumer
Event ItemMovedEvent(id=7, timestamp=1706305402878, vector=(2, 3, 1)) needed 7 ms to get to the consumer
Event ItemMovedEvent(id=7, timestamp=1706305402879, vector=(0, 1, 3)) needed 7 ms to get to the consumer
Event ItemDeletedEvent(id=7, timestamp=1706305402881) needed 6 ms to get to the consumer
Event ItemCreatedEvent(id=8, timestamp=1706305402886, position=(0, 0, 0), value=0) needed 2 ms to get to the consumer
Event ItemMovedEvent(id=8, timestamp=1706305402887, vector=(8, 8, 8)) needed 1 ms to get to the consumer
Already one item at this postion - will be deleted
Event ItemCreatedEvent(id=9, timestamp=1706305402888, position=(0, 0, 0), value=0) needed 1 ms to get to the consumer
Event ItemDeletedEvent(id=8, timestamp=1706305402888) needed 1 ms to get to the consumer
Event ItemMovedEvent(id=9, timestamp=1706305402889, vector=(8, 8, 8)) needed 1 ms to get to the consumer
Already one item at this postion - will be deleted
Event ItemCreatedEvent(id=10, timestamp=1706305402890, position=(0, 0, 0), value=0) needed 2 ms to get to the consumer
Event ItemDeletedEvent(id=9, timestamp=1706305402892) needed 0 ms to get to the consumer
Item: 1, Location: (9, 4, 8), Moves: 4, Value: 0
Item: 2, Location: (7, 7, 6), Moves: 2, Value: 0
Item: 3, Location: (4, 2, 3), Moves: 1, Value: 0
Item: 4, Location: (0, 0, 0), Moves: 0, Value: 0
Item: 5, Location: (10, 10, 14), Moves: 5, Value: 0
Item: 6, Location: (18, 15, 20), Moves: 8, Value: 0
Item: 10, Location: (0, 0, 0), Moves: 0, Value: 0
Specific item details: MovingItemDTO(name=3, location=(4, 2, 3), numberOfMoves=1, value=0)
Event ItemMovedEvent(id=10, timestamp=1706305402892, vector=(8, 8, 8)) needed 1 ms to get to the consumer
Average time until reaching consumer 26.74 ms

```

Abbildung 3: Output Aufgabe 3

2.4 Aufgabe 4: Nutzung von Apache Kafka im Event Store

Apache Kafka wurde als persistenter Speicher für Ereignisse im Event Store eingeführt. Das Domänenmodell wird dynamisch basierend auf den im Event Store gespeicherten Ereignissen aufgebaut. Eine Umstellung von der zuvor verwendeten Map zur Speicherung der Objekte wurde durchgeführt.

Die Windows-spezifischen Instruktionen zur Installation von Apache Kafka erwiesen sich als nicht optimal und somit wurde mit einem Docker Image ein Container für Apache Kafka und Zookeeper erstellt. Dieser Ansatz ermöglichte eine konsistente und isolierte Umgebung, unabhängig von den spezifischen Eigenheiten des Windows-Betriebssystems.

Während der Implementierung stellte sich die Verwendung von Offset-Explorer als leistungstarkes Werkzeug heraus, um Kafka-Topics zu analysieren und zu überwachen. Dieses Tool ermöglichte einen visuellen Einblick in die Topics und half bei der Fehlersuche und Überwachung des Datenflusses.

Die Verwendung von Kafka als Datenbank wurde kritisch betrachtet, da Kafka primär eine Event-Streaming-Plattform ist. Insbesondere wurde die Ressourcenintensität von `loadNamesOfMovingItems()` in Frage gestellt, und die Eignung von Kafka für diesen Anwendungsfall wurde diskutiert. Nach längerer Überlegung und Recherche wurden jedoch die Vorteile, die Kafka auch als Datenbank bietet, deutlicher, insbesondere die Echtzeitverarbeitung und Skalierbarkeit.

```
1 private fun printQueryResults(queryHandler: QueryHandler) {  
2     queryHandler.getMovingItems().forEach { dto ->  
3         println("Item: ${dto.name}, Location: ${dto.location}, Moves: ${dto.numberOfMoves},  
4             Value: ${dto.value}")  
5     }  
6     try {  
7         println("Specific item details: ${queryHandler.getMovingItemByName("3")}")  
8     } catch (e: NoSuchElementException) {  
9         println("Error: ${e.message}")  
10    }  
11 }
```

Codeauschnitt 2.6: Aufgabe 4 Printausgabe

```
Item: 1, Location: (22, 21, 14), Moves: 9, Value: 0  
Item: 3, Location: (24, 19, 22), Moves: 10, Value: 0  
Item: 5, Location: (15, 19, 12), Moves: 8, Value: 0  
Item: 6, Location: (10, 7, 1), Moves: 3, Value: 0  
Item: 2, Location: (14, 12, 10), Moves: 8, Value: 0  
Item: 10, Location: (8, 8, 8), Moves: 1, Value: 0  
Specific item details: MovingItemDTO(name=3, location=(24, 19, 22), numberOfMoves=10, value=0)
```

Abbildung 4: Output Aufgabe 4

Zur Überprüfung der Lösung wurden am Ende die Consumer, Producer und veröffentlichten Nachrichten in Offset analysiert und nachvollzogen.

Partition	Offset	Key	Value	Timestamp
0	0	Streaming	["eventType":"ItemCreatedEvent","id":"1","timestamp":1706306408000,"position":{"x":0,"y":0,"z":0},"value":0]	2024-01-26 23:00:28.371
0	1	Streaming	["eventType":"ItemMovedEvent","id":"1","timestamp":1706306428600,"position":{"x":3,"y":2,"z":2},"value":0]	2024-01-26 23:00:28.605
0	2	Streaming	["eventType":"ItemCreatedEvent","id":"2","timestamp":1706306428656,"position":{"x":0,"y":0,"z":0},"value":0]	2024-01-26 23:00:28.656
0	3	Streaming	["eventType":"ItemMovedEvent","id":"2","timestamp":1706306428701,"position":{"x":0,"y":1,"z":3}]	2024-01-26 23:00:28.701
0	4	Streaming	["eventType":"ItemMovedEvent","id":"2","timestamp":1706306428744,"position":{"x":1,"y":1,"z":0}]	2024-01-26 23:00:28.744
0	5	Streaming	["eventType":"ItemMovedEvent","id":"2","timestamp":1706306428784,"position":{"x":2,"y":4,"z":2}]	2024-01-26 23:00:28.784
0	6	Streaming	["eventType":"ItemMovedEvent","id":"2","timestamp":1706306428828,"position":{"x":2,"y":4,"z":4}]	2024-01-26 23:00:28.828
0	7	Streaming	["eventType":"ItemMovedEvent","id":"2","timestamp":1706306428866,"position":{"x":3,"y":3,"z":0}]	2024-01-26 23:00:28.866
0	8	Streaming	["eventType":"ItemCreatedEvent","id":"3","timestamp":1706306428942,"position":{"x":0,"y":0,"z":0},"value":0]	2024-01-26 23:00:28.942
0	9	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306428999,"position":{"x":3,"y":2,"z":3}]	2024-01-26 23:00:28.999
0	10	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429038,"position":{"x":4,"y":3,"z":1}]	2024-01-26 23:00:29.038
0	11	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429076,"position":{"x":0,"y":3,"z":0}]	2024-01-26 23:00:29.076
0	12	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429121,"position":{"x":4,"y":0,"z":1}]	2024-01-26 23:00:29.121
0	13	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429166,"position":{"x":1,"y":3,"z":3}]	2024-01-26 23:00:29.166
0	14	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429208,"position":{"x":3,"y":2,"z":0}]	2024-01-26 23:00:29.208
0	15	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429249,"position":{"x":1,"y":1,"z":3}]	2024-01-26 23:00:29.250
0	16	Streaming	["eventType":"ItemMovedEvent","id":"3","timestamp":1706306429306,"position":{"x":3,"y":0,"z":4}]	2024-01-26 23:00:29.307
0	17	Streaming	["eventType":"ItemCreatedEvent","id":"4","timestamp":1706306429411,"position":{"x":0,"y":0,"z":0},"value":0]	2024-01-26 23:00:29.411
0	18	Streaming	["eventType":"ItemMovedEvent","id":"4","timestamp":1706306429449,"position":{"x":2,"y":1,"z":1}]	2024-01-26 23:00:29.449
0	19	Streaming	["eventType":"ItemMovedEvent","id":"4","timestamp":1706306429516,"position":{"x":0,"y":4,"z":1}]	2024-01-26 23:00:29.521
0	20	Streaming	["eventType":"ItemMovedEvent","id":"4","timestamp":1706306429552,"position":{"x":0,"y":2,"z":2}]	2024-01-26 23:00:29.552
0	21	Streaming	["eventType":"ItemCreatedEvent","id":"5","timestamp":1706306429587,"position":{"x":0,"y":0,"z":0},"value":0]	2024-01-26 23:00:29.587
0	22	Streaming	["eventType":"ItemMovedEvent","id":"5","timestamp":1706306429628,"position":{"x":2,"y":2,"z":3}]	2024-01-26 23:00:29.628

Abbildung 5: Offset Explorer Aufgabe 4

3 Verkehrsüberwachung

3.1 Aufgabe 5: Datengenerierung

Ein Testdatengenerator wurde entwickelt, der Datensätze für die Verkehrsüberwachung mit variablen Parametern erstellt. Die Datensätze werden in einen Apache Kafka Topic eingestellt.

Im Mittelpunkt stand ein vollständig anpassbarer Generator, der durch eine globale Konfiguration gesteuert wurde. Diese Flexibilität ermöglichte es, die Generierung von Daten präzise zu steuern und anzupassen, was für die spezifischen Anforderungen der Aufgabe essentiell war. Ein Schlüsselement hierbei war die Möglichkeit Werte mit bestimmten Wahrscheinlichkeiten zu generieren, beispielsweise keinen Wert mit einer Chance von 1 Prozent und negative Werte mit einer Chance von 5 Prozent.

Die Kapselung der Funktionalität erfolgte durch eine dedizierte Generator-Klasse. Diese Kapselung trug zur Klarheit und Wartbarkeit des Codes bei, indem sie eine klare Trennung der Generierungsfunktionalität vom Rest der Anwendung ermöglichte.

Ein weiteres wichtiges Merkmal der Lösung war die Aggregation nur gültiger Daten. Dies bedeutet, dass nur Daten, die bestimmte Kriterien erfüllten, für die weitere Verarbeitung und Analyse berücksichtigt wurden. Diese Vorgehensweise verbesserte die Qualität und Relevanz der verarbeiteten Daten.

Die Anwendung funktionaler Programmierprinzipien spielte ebenfalls eine wichtige Rolle. Diese Prinzipien unterstützten eine klare, modulare und effiziente Gestaltung der Codebasis, was besonders in der Echtzeit-Datenverarbeitung mit Zeitfenstern von Vorteil war.

Für das Zeitmanagement innerhalb der Anwendung wurde auf Klassen wie `Instant` und `Duration` zurückgegriffen. Diese Klassen aus der Java-Standardbibliothek ermöglichten eine präzise und intuitive Handhabung von Zeit- und Dauerangaben, was für die Echtzeit-Datenverarbeitung und das Zeitfenster-Management unerlässlich war.

Abschließend war die Fehlerbehandlung bei der JSON-Verarbeitung ein kritischer Aspekt. Es wurde sichergestellt, dass Fehler effektiv erkannt und behandelt wurden, um die Stabilität und Zuverlässigkeit der Anwendung bei der Verarbeitung von Datenströmen zu gewährleisten.

Um die Zuverlässigkeit und Korrektheit der generierten Daten zu gewährleisten, wurden Unit-Tests durchgeführt. Diese Tests überprüften die Funktionalität des Generators unter verschied-

denen Bedingungen und stellten sicher, dass die generierten Daten den gestellten Anforderungen entsprachen.

Im ersten Test der `KafkaConsumerTest`-Klasse wird eine private Methode `processSensorData` getestet, um sicherzustellen, dass Geschwindigkeitsdaten von Metern pro Sekunde in Kilometer pro Stunde umgerechnet und korrekt in einer Map gespeichert werden. Der zweite Test überprüft die gleiche Methode, fokussiert sich aber darauf, zu bestätigen, dass negative Geschwindigkeitswerte angemessen behandelt werden, indem sichergestellt wird, dass alle gespeicherten Geschwindigkeiten im Ergebnis größer oder gleich Null sind.

```

1 class KafkaConsumerTest {
2     @Test
3     fun 'test private method processSensorData for speed conversion'() {
4         val kafkaConsumer = KafkaConsumer()
5         val processSensorDataMethod = KafkaConsumer::class.declaredMemberFunctions
6             .firstOrNull { it.name == "processSensorData" }
7             ?: throw NoSuchMethodException("Method processSensorData not found")
8
9         processSensorDataMethod.isAccessible = true
10
11         val originalSpeedsMs = listOf(20.4, 33.5, 40.0)
12
13         val expectedSpeedsKmh = originalSpeedsMs.map { round(it * ConsumerConfig.KM_FACTOR*
14             10) / 10.0 }
15
16         val sensorData = SensorData("2024-01-01T10:01:29.551Z", 1, originalSpeedsMs)
17         processSensorDataMethod.call(kafkaConsumer, sensorData, System.currentTimeMillis())
18
19         val sensorSpeeds = kafkaConsumer::class.memberProperties
20             .firstOrNull { it.name == "sensorSpeeds" }
21             ?.also { it.isAccessible = true }
22             ?.call(kafkaConsumer) as? Map<Int, List<Double>>
23
24         assertNotNull(sensorSpeeds)
25         assertTrue(sensorSpeeds?.containsKey(1) ?: false)
26         assertEquals(expectedSpeedsKmh, sensorSpeeds?.get(1))
27     }
28 }

```

Codeauschnitt 3.1: Aufgabe 5 Unit Test Conversion

```

1 class KafkaConsumerTest {
2     @Test
3     fun 'test processSensorData with negative speeds'() {
4         val kafkaConsumer = KafkaConsumer()
5         val processSensorDataMethod = KafkaConsumer::class.declaredMemberFunctions
6             .firstOrNull { it.name == "processSensorData" }
7             ?: throw NoSuchMethodException("Method processSensorData not found")
8
9         processSensorDataMethod.isAccessible = true
10
11         val sensorData = SensorData("2023-09-23T10:01:29.551Z", 1, listOf(-10.0, 20.0, 30.0))
12         processSensorDataMethod.call(kafkaConsumer, sensorData, System.currentTimeMillis())
13
14         val sensorSpeeds = kafkaConsumer::class.memberProperties
15             .firstOrNull { it.name == "sensorSpeeds" }
16             ?.also { it.isAccessible = true }
17             ?.call(kafkaConsumer) as? Map<Int, List<Double>>
18
19         assertTrue(sensorSpeeds?.get(1)?.all { it >= 0 } ?: false)
20     }
21 }

```

Codeauschnitt 3.2: Aufgabe 5 Unit Test Filtering

Insgesamt führte diese Kombination aus anpassbarer Datengenerierung, sorgfältiger Kapselung, umfassenden Tests, Datenaggregation, funktionalen Programmierprinzipien, effizientem Zeitmanagement und robustem Fehlerhandling zu einer leistungsfähigen und zuverlässigen Lösung für die gestellten Aufgaben.

```

-----
30s passed 1. data is available
Sensor 3: average speed = 103.2 km/h
With those speeds: [94.3, 67.3, 73.4, 54.4, 78.1, 69.8, 63.4, 169.6, 152.3, 39.6, 170.6, 139.0, 169.6]
Sensor 2: average speed = 91.4 km/h
With those speeds: [49.0, 82.4, 59.8, 97.9, 58.3, 38.5, 148.0, 119.2, 106.2, 94.0, 85.3, 158.8]
Sensor 1: average speed = 102.3 km/h
With those speeds: [160.2, 117.7, 131.4, 62.6, 87.8, 59.0, 165.6, 103.7, 45.0, 42.1, 98.6, 153.4]
-----
30s passed 2. data is available
Sensor 3: average speed = 106.9 km/h
With those speeds: [42.1, 99.4, 141.5, 147.2, 154.1, 47.5, 135.7, 38.2, 175.7, 124.9, 60.5, 177.1, 126.7, 85.0, 48.6]
Sensor 2: average speed = 104.8 km/h
With those speeds: [40.0, 126.4, 90.0, 130.0, 112.3, 112.3, 126.4, 46.1, 140.8, 122.0, 124.6, 138.2, 91.4, 67.0]
Sensor 1: average speed = 116.8 km/h
With those speeds: [133.9, 106.9, 108.0, 113.0, 167.8, 117.4, 126.7, 130.3, 47.5]

```

Abbildung 6: Output Aufgabe 5

3.2 Aufgabe 6: Skalierung

Der Testdatengenerator erstellt Datensätze nun in mehrere Partitionen des Kafka Topics.

Die wichtigste Feststellung war, dass die Konsolenauswertung auch nach der Änderung korrekt funktionierte. Es gab keine Notwendigkeit, signifikante Modifikationen an der bestehenden Lösung vorzunehmen, da das Programm die Daten, die über verschiedene Partitionen verteilt waren, effektiv verarbeiten konnte.

Ein wesentlicher Aspekt der Aufgabe war die dynamische Zuweisung des Topic-Keys, was bedeutet, dass die Zuweisung der Daten zu verschiedenen Partitionen nicht statisch, sondern basierend auf bestimmten Kriterien oder Algorithmen erfolgte. Diese Flexibilität in der Zuweisung der Topic-Keys trug dazu bei, die Effizienz der Datenverteilung zu erhöhen und eine gleichmäßigere Lastverteilung über die Partitionen zu gewährleisten.

Bezüglich der Leistungsaspekte wurde festgestellt, dass keine signifikanten Performance-Unterschiede zwischen den Lösungen von Aufgabe 5 und der aktuellen Aufgabe auftraten. Dies deutet darauf hin, dass die Änderungen an der Datenverteilungslogik und die Einführung von dynamischen Topic-Keys keinen negativen Einfluss auf die Gesamtleistung des Systems hatten.

Zusammenfassend lässt sich sagen, dass die Erweiterung des Testdatengenerators um die Fähigkeit, Daten in verschiedene Partitionen eines Topics einzustellen, erfolgreich implementiert wurde, ohne die Funktionalität oder Performance der bestehenden Konsolenauswertung negativ zu beeinflussen. Die Lösung erwies sich als effizient im Umgang mit der neuen Anforderung der Datenpartitionierung.

Partition	Offset	Key	Value	Timestamp
2	139	0	("timestamp":"2024-01-26T22:30:32.146492300Z",...	2024-01-26 23:30:32.641
0	196	1	("timestamp":"2024-01-26T22:30:33.960381900Z",...	2024-01-26 23:30:33.961
2	140	2	("timestamp":"2024-01-26T22:30:35.243497400Z",...	2024-01-26 23:30:35.243
2	141	3	("timestamp":"2024-01-26T22:30:37.467347800Z",...	2024-01-26 23:30:37.167
1	143	4	("timestamp":"2024-01-26T22:30:38.255993900Z",...	2024-01-26 23:30:38.255
0	197	5	("timestamp":"2024-01-26T22:30:39.961040400Z",...	2024-01-26 23:30:39.961
1	144	6	("timestamp":"2024-01-26T22:30:41.915044200Z",...	2024-01-26 23:30:41.615
0	198	7	("timestamp":"2024-01-26T22:30:42.786713600Z",...	2024-01-26 23:30:42.786
0	199	8	("timestamp":"2024-01-26T22:30:44.050592600Z",...	2024-01-26 23:30:44.050
2	142	9	("timestamp":"2024-01-26T22:30:45.176198800Z",...	2024-01-26 23:30:45.175
1	145	10	("timestamp":"2024-01-26T22:30:46.502547200Z",...	2024-01-26 23:30:46.502
0	200	11	("timestamp":"2024-01-26T22:30:48.392457500Z",...	2024-01-26 23:30:48.392
1	146	12	("timestamp":"2024-01-26T22:30:49.814832700Z",...	2024-01-26 23:30:49.814
1	147	13	("timestamp":"2024-01-26T22:30:51.232462100Z",...	2024-01-26 23:30:51.232
1	148	14	("timestamp":"2024-01-26T22:30:52.489393Z",se...	2024-01-26 23:30:52.489
0	201	15	("timestamp":"2024-01-26T22:30:53.714464200Z",...	2024-01-26 23:30:53.715
2	143	16	("timestamp":"2024-01-26T22:30:54.802494Z",se...	2024-01-26 23:30:54.802
0	202	17	("timestamp":"2024-01-26T22:30:56.434501900Z",...	2024-01-26 23:30:56.434
1	149	18	("timestamp":"2024-01-26T22:30:58.271520200Z",...	2024-01-26 23:30:58.271
1	150	19	("timestamp":"2024-01-26T22:30:59.694337400Z",...	2024-01-26 23:30:59.694
1	151	20	("timestamp":"2024-01-26T22:31:01.662593900Z",...	2024-01-26 23:31:01.662
0	203	21	("timestamp":"2024-01-26T22:31:03.524569600Z",...	2024-01-26 23:31:03.524
0	204	22	("timestamp":"2024-01-26T22:31:05.364347Z",se...	2024-01-26 23:31:05.364

Abbildung 7: Offset Explorer Aufgabe 6

3.3 Aufgabe 7: Apache Beam

Die Anwendung wurde erweitert, um Apache Beam zur Konsumierung von Kafka-Topic-Daten zu verwenden und Berechnungen zur Durchschnittsgeschwindigkeit pro Sensor in Zeitfenstern durchzuführen.

Die Implementierung erfolgte durch den Aufbau einer Apache Beam-Pipeline, die speziell für die Anforderungen der Verkehrsüberwachung konzipiert wurde. Dabei wurde insbesondere auf die Integration von Transformationsklassen und die Beibehaltung von den Konstanten in den Konfigurationsdateien geachtet.

Die Pipeline wurde in die Consumer Klasse integriert. Dies ermöglicht eine nahtlose Interaktion zwischen der Datenquelle (Kafka) und der Verarbeitungseinheit (Apache Beam-Pipeline), wodurch unnötige Komplexität vermieden wird und die genaue Struktur aus Aufgabe 5 weiterhin bestehen bleibt.

```
1 val options = PipelineOptionsFactory.create()
2     val pipeline: Pipeline = Pipeline.create(options)
3
4     val kafkaRead = KafkaIO.read<String, String>().withBootstrapServers("localhost:9092")
5         .withKeyDeserializer(StringDeserializer::class.java).withValueDeserializer(
6             StringDeserializer::class.java)
7         .withTopic(AppConfig.TOPIC)
8
9     val kafkaRecords = pipeline.apply("Read from Kafka", kafkaRead)
10
11     val sensorDataRecords = kafkaRecords.apply(
12         "JSON",
13         ParDo.of(JSOToSensorData())
14     ).setCoder(SerializableCoder.of(SensorData::class.java))
```

Codeauschnitt 3.3: Aufgabe 7 Pipeline Generierung

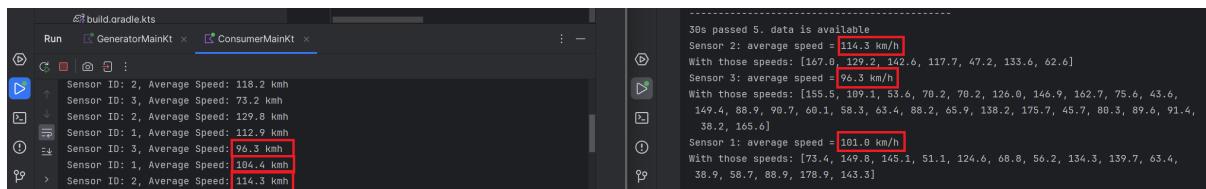
Es war wichtig, dass die Klasse SensorData aus den vorherigen Aufgaben übernommen wurde, um eine übersichtliche Lösung zu gewährleisten. Allerdings führte dies bei einigen Transformationen zu Problemen. Wenn keine standardisierten Typen verwendet wurden, benötigt PCollection einen speziellen Coder für die Serialisierung.

Für gute Lesbarkeit wurden alle Klassen, die für die Transformationen benötigt wurden, modularisiert. Die Pipeline beginnt damit, Nachrichten aus einem Kafka-Thema zu lesen, wobei der KafkaIO-Connector verwendet wird. Anschließend transformiert sie die Rohdaten im JSON-Format mithilfe einer ParDo-Funktion (JSOToSensorData) und der Bibliothek Jackson, um sie in sinnvolle SensorData-Objekte zu konvertieren.

Die Ausgabe der Kafka-Consumer-Pipeline wird im Terminal angezeigt, und die berechneten Durchschnittsgeschwindigkeiten werden mit den Ergebnissen aus Aufgabe 5 verglichen. Beide Consumer wurden nahezu gleichzeitig gestartet, und die Zielsetzung bestand darin, sicherzustellen, dass beide Consumer konsistente und vergleichbare Durchschnittswerte für die

Geschwindigkeiten berechnen.

Die Ausgaben beider Consumer können visuell überprüft werden, um sicherzustellen, dass die ermittelten Durchschnittsgeschwindigkeiten für die jeweiligen Sensor-IDs konsistent sind, wie in der folgenden Abbildung zu sehen ist. Aufgrund von der nicht exakt gleichen Startzeit der Consumer weicht der Durchschnitt leicht voneinander ab. Es ist aber zu erkennen, dass die Werte sehr nah aneinander liegen.



```
build.gradle.kts
Run GeneratorMainKt ConsumerMainKt

Sensor ID: 2, Average Speed: 118.2 kmh
Sensor ID: 3, Average Speed: 73.2 kmh
Sensor ID: 2, Average Speed: 129.8 kmh
Sensor ID: 1, Average Speed: 112.9 kmh
Sensor ID: 3, Average Speed: 96.3 kmh
Sensor ID: 1, Average Speed: 104.4 kmh
Sensor ID: 2, Average Speed: 114.3 kmh

30s passed 5. data is available
Sensor 2: average speed = 114.3 km/h
With those speeds: [167.0, 129.2, 142.6, 117.7, 47.2, 133.6, 62.6]
Sensor 3: average speed = 96.3 km/h
With those speeds: [155.5, 109.1, 53.6, 70.2, 70.2, 126.0, 146.9, 162.7, 75.6, 43.6, 149.4, 88.9, 90.7, 60.1, 58.3, 63.4, 88.2, 65.9, 138.2, 175.7, 45.7, 80.3, 89.6, 91.4, 38.2, 165.6]
Sensor 1: average speed = 101.0 km/h
With those speeds: [73.4, 149.8, 145.1, 51.1, 124.6, 68.8, 56.2, 134.3, 139.7, 63.4, 38.9, 58.7, 88.9, 178.9, 143.3]
```

Abbildung 8: Output Vergleich Aufgabe 7 und Aufgabe 5

Um die Filterung der negativen Werte und der Sensordaten, die keine Geschwindigkeiten enthalte, wurde noch ein Unit-Test implementiert.

Apache Beam bietet mit PAssert eine nützliche Testbibliothek, die speziell für das Überprüfen von Daten in Pipelines entwickelt wurde. PAssert ermöglicht es, klare Aussagen darüber zu machen, was in Pipelines erwartet wird. Es handelt sich um eine Assertion-Bibliothek, die speziell für das Testen von Daten in parallelen und verteilten Datenverarbeitungsszenarien entwickelt wurde.

```
1 class KafkaConsumerTest {
2     @Test
3     fun testFilterNegativeAndZeroSpeeds() {
4         val options = PipelineOptionsFactory.create()
5         val pipeline = TestPipeline.create(options)
6
7         val testInput = listOf(
8             SensorData("",1, listOf(10.0, -1.0, 0.0, 30.0)),
9             SensorData("",2, listOf(20.0, 30.0)),
10            SensorData("",3, listOf(-5.0, 0.0, 15.0))
11        )
12
13        val inputPCollection: PCollection<SensorData> = pipeline
14            .apply("Erstelle Testdaten", Create.of(testInput))
15            .setCoder(SerializableCoder.of(SensorData::class.java))
16
17        val filteredSensorData = inputPCollection.apply("Filter Negative and Zero Speeds",
18            ParDo.of(object : DoFn<SensorData, SensorData>() {
19                @ProcessElement
20                fun processElement(@Element sensorData: SensorData, output: OutputReceiver<
21                    SensorData>) {
22                    val filteredSpeeds = sensorData.speeds.filter { it > 0 }
23                    output.output(SensorData("",sensorData.sensorId, filteredSpeeds))
24                }
25            })))
26
27        PAssert.that(filteredSensorData).containsInAnyOrder(
28            SensorData("",1, listOf(10.0, 30.0)),
29            SensorData("",2, listOf(20.0, 30.0)),
30            SensorData("",3, listOf(15.0))
31        )
32
33        pipeline.run().waitUntilFinish()
34    }
35 }
```

Codeauschnitt 3.4: Aufgabe 7 PAssert Unit Test

3.4 Aufgabe 8: EPL und Esper

In dieser Aufgabe wurde die Implementierung von Complex Event Processing (CEP) mit Esper behandelt. Es wurden EPL-Anfragen erstellt, um Daten zu bereinigen und die Durchschnittsgeschwindigkeit zu berechnen. Zudem werden komplexe Ereignisse identifiziert, die mögliche Staus anzeigen könnten.

Ein zentraler Punkt in dieser Aufgabe war die Verwendung von Eventtypen als Data Classes in Kotlin. Durch diese Entscheidung konnte der Code deutlich reduziert werden, da Kotlin Data Classes automatisch Getter- und Setter-Methoden sowie andere nützliche Funktionen generieren. Diese Verringerung des Boilerplate-Codes führte zu einer schlankeren und klareren Codebasis.

Zudem wurde der bereits in der vorherigen Aufgabe angepasste Generator erneut verwendet und leicht modifiziert.

Die Konfigurierbarkeit des Systems wurde durch die Verwendung globaler Konfigurationen, wie zum Beispiel für den Generator oder für die Schwellenwerte für Stauereignisse und Zeitfenster, weiter verbessert. Diese globale Konfigurierbarkeit erlaubte eine einfache Anpassung an unterschiedliche Szenarien oder Anforderungen.

Ein weiterer wichtiger Aspekt war die Modularisierung des Codes. Durch die Aufteilung des Systems in kleinere, überschaubare Module wurde die Übersichtlichkeit und Wartbarkeit des Codes erhöht. Diese Strukturierung erleichterte auch das Testen der einzelnen Komponenten, da jede in einem isolierten Kontext betrachtet und getestet werden konnte.

Schließlich wurden spezifische Unit-Tests implementiert, um die Funktionalität der Stauerkennung zu überprüfen. Diese Tests spielten eine entscheidende Rolle bei der Sicherstellung, dass die Stauerkennung korrekt und zuverlässig funktionierte. Die Fähigkeit, spezifische Funktionen durch Unit-Tests zu validieren, ist ein wichtiger Bestandteil der Softwarequalitätssicherung.

Die TrafficJamWarningTest-Klasse in Kotlin führt einen Test durch, um sicherzustellen, dass der TrafficJamWarningEventListener korrekt ausgelöst wird, wenn Sensordaten, die einen bestimmten Schwellenwert zwischen zwei berechneten Durchschnitts- überschreiten, verarbeitet werden. Im Test werden zunächst normale Verkehrsdaten und anschließend Verkehrsdaten, die über diesem Schwellenwert liegen, an das System gesendet, woraufhin überprüft wird, ob der Listener wie vorgesehen nur einmalig reagiert.

```
1 class TrafficJamWarningTest {
2
3     @Test
4     fun testTrafficJamWarningEventListenerTriggered() {
5         val runtime = setupEsperRuntime()
6
7         val cleanDeployment = compileAndDeploy(runtime, cleanEPL())
8         setupListener(runtime, cleanDeployment, "CleanEvents", CleanSensorEventListener())
9
10        val valueConversionDeployment = compileAndDeploy(runtime, valueConversionEPL())
11        setupListener(runtime, valueConversionDeployment, "ValueConversionCalculation",
12            ValueConversionListener())
13
14        val avgSpeedDeployment = compileAndDeploy(runtime, avgSpeedEPL())
15        setupListener(runtime, avgSpeedDeployment, "AvgSpeedCalculation",
16            AvgSpeedEventListener())
17
18        val trafficJamWarningListenerMock = Mockito.mock(TrafficJamWarningEventListener::class
19            .java)
20        val trafficJamWarningDeployment = compileAndDeploy(runtime, trafficJamWarningEPL())
21        setupListener(
22            runtime,
23            trafficJamWarningDeployment,
24            "TrafficJamWarningCalculation",
25            trafficJamWarningListenerMock
26        )
27
28        runtime.eventService.sendEventBean(ValueConversionEvent(1, 70.0), "
29            ValueConversionEvent")
30        runtime.eventService.sendEventBean(ValueConversionEvent(2, 70.0), "
31            ValueConversionEvent")
32        Thread.sleep(11000)
33
34        runtime.eventService.sendEventBean(ValueConversionEvent(1, 30.0), "
35            ValueConversionEvent")
36        runtime.eventService.sendEventBean(ValueConversionEvent(2, 60.0), "
37            ValueConversionEvent")
38        Thread.sleep(11000)
39
40        verify(trafficJamWarningListenerMock, times(1)).update(any(), any(), any(), any())
41    }
42 }
```

Codeauschnitt 3.5: Aufgabe 8 Esper Traffic Jam Test

```
Durchschnittsgeschwindigkeit: Sensor 1 - 129.38571428571422 km/h  
Durchschnittsgeschwindigkeit: Sensor 2 - 134.63333333333333 km/h  
Durchschnittsgeschwindigkeit: Sensor 3 - 83.17499999999997 km/h  
Stauwarnung: Sensor 3 - 51.458333333333336
```

Abbildung 9: Output Aufgabe 8

Zusammenfassend lässt sich sagen, dass durch die Nutzung von Kotlin Data Classes, die Wiederverwendung und Anpassung bestehender Komponenten, die erweiterte Konfigurierbarkeit, die klare Modularisierung und die gründlichen Unit-Tests eine leistungsstarke, flexible und gut wartbare Lösung geschaffen wurde.

4 Aufgabe 10: Vergleich der Technologien

Im folgenden Kapitel wird eine ausführliche Bewertung der eingesetzten Technologien (Java Messaging System, Apache Kafka, Apache Beam, Complex Event Processing mit EPL/Esper) vorgenommen. Dabei werden verschiedene Kriterien wie Datenrepräsentation, Programmiermodelle und Ausfallsicherheit berücksichtigt.

4.1 Bewertung der Technologien

1. Repräsentation von Ereignissen

Java Messaging System (JMS):

JMS nutzt Message-Objekte, die verschiedene Formate wie Map, Byte, Stream, Text und Object Messages unterstützen. Diese Objekte können als JSON String serialisiert und als Text Message verschickt werden, was eine hohe Flexibilität in der Datenrepräsentation bietet. Die vorhandenen Messagedatentypen erben von einer gemeinsamen Parent Message Klasse und können weiter vererbt werden. Beim Empfang der Nachrichten müssen diese jedoch entsprechend deserialisiert werden. Metadaten wie Modus, Lebensdauer oder Priorität können ebenfalls übermittelt werden, was die Übertragung von zusätzlichen Informationen ermöglicht.

Apache Kafka:

Kafka verwendet ein Nachrichtenformat, das aus Schlüssel, Wert, Timestamp und optionalen Metadaten in Headern besteht. Nachrichten können als JSON serialisiert werden, was eine hohe Flexibilität bietet, da nahezu alle Datenstrukturen als JSON repräsentiert werden können. Allerdings gibt es in Kafka keine direkte Unterstützung für Vererbungsstrukturen bei Nachrichtentypen.

Apache Beam:

Beam abstrahiert Ereignisse als Elemente in PCollections und unterstützt benutzerdefinierte Datentypen, was eine hohe Flexibilität in der Datenstruktur ermöglicht. Die Verwendung von Java-Klassen ermöglicht zudem die Nutzung von Vererbungsstrukturen, was bei Transformationen nützlich sein kann. Jedoch werden in Apache Beam keine Metainformationen über Ereignisse mitgeführt.

Complex Event Processing (CEP) mit EPL / Esper:

In CEP mit EPL/Esper werden komplexe Ereignisse als spezielle Data Classes repräsentiert. Diese können flexibel gestaltet werden, um verschiedene Datentypen zu enthalten. Es ist auch möglich, Events von anderen Events abzuleiten, was eine gewisse Flexibilität in der Ereignisstruktur bietet. Allerdings werden in diesem System keine Metainformationen zu den Ereignissen mitgeführt.

2. Erzeugung und Übermittlung von Ereignissen/Nachrichten**Java Messaging System (JMS):**

In JMS erzeugen Produzenten Nachrichten, die dann über eine Connection Factory an einen Broker gesendet werden. Dies erfordert das Erstellen einer Session über die Verbindung, um Nachrichten an ein spezifisches Topic oder eine Queue zu senden. Nach dem Senden der Nachricht muss die Session geschlossen werden. JMS unterstützt kein Batching, also das gebündelte Senden von Nachrichten.

Apache Kafka:

In Kafka senden Produzenten Nachrichten an einen oder mehrere Kafka-Broker. Die Nachrichten werden in Topics organisiert, und der Broker sorgt für die Verteilung. Die Verbindung zum Kafka-Cluster und das Auswählen des Topics sind erforderlich. Kafka unterstützt das Batching von Nachrichten, was die Effizienz der Datenübertragung verbessern kann.

Apache Beam:

Beam verwendet Pipelines zur Übermittlung von Ereignissen. Daten werden über diverse Quellen in die Pipeline eingespeist, wobei Transformationen innerhalb der Pipeline durchgeführt werden. Beam unterstützt auch Batching und ermöglicht es, eine große Anzahl von Ereignissen in verschiedenen Fenstern zu verarbeiten.

Complex Event Processing (CEP) mit EPL / Esper:

In CEP mit EPL/Esper werden Ereignisse über das Laufzeitsystem oder über EPL generiert. Ereignisse können auch in Reaktion auf bestimmte Muster von Ereignissen erzeugt werden. Da es keinen zentralen Broker gibt, werden Ereignisse als Stream gesendet und können gebündelt übertragen werden.

3. Konsumentenanzwendung

Java Messaging System (JMS):

In JMS hören Verbraucher auf Queues oder Topics, indem sie eine Session über eine Connection erstellen und dem Consumer eine Listener-Funktion als Callback zuweisen. Die Nachrichten werden deserialisiert und entsprechend verarbeitet, sobald sie eintreffen.

Apache Kafka:

Kafka-Consumer abonnieren Topics und verbinden sich über die Kafka Consumer API mit dem Cluster. Der Broker sendet Nachrichten gemäß der Partitionen an die Consumer. Kafka setzt auf ein Pull-basiertes System, wobei die Consumer die Nachrichten abrufen.

Apache Beam:

Beam verarbeitet Daten in Echtzeit oder als Batch und bietet flexible Ausgabemöglichkeiten. Die Datenflusssteuerung in der Pipeline wird durch den Beam Runner geregelt, und es gibt spezielle I/O-Transformationen für verschiedene Datenquellen wie Kafka.

Complex Event Processing (CEP) mit EPL / Esper:

CEP mit EPL/Esper verwendet Ereignislistener, die auf spezifische Ereignismuster reagieren und diese dann entsprechend weiterverarbeiten.

4. Dauerhafte Speicherung

Java Messaging System (JMS):

JMS selbst bietet keine persistente Speicherung von Ereignissen. Eine externe Persistenz, wie eine Datenbank oder eine Datei, ist notwendig, um Nachrichten bis zur erfolgreichen Auslieferung zu speichern.

Apache Kafka:

Kafka speichert Nachrichten dauerhaft in Partitionen, wobei die Aufbewahrungsdauer entweder zeit- oder größenbasiert festgelegt werden kann.

Apache Beam:

Beam ist kein Datenspeichersystem, sondern ein Datenverarbeitungsmodell. Bei der Integration mit Systemen wie Kafka können jedoch die Datenspeicherfunktionen dieser Systeme genutzt werden.

Complex Event Processing (CEP) mit EPL / Esper:

EPL/Esper fokussiert sich auf Echtzeitverarbeitung und bietet keine eingebaute Persistenz. Daten müssen außerhalb des Systems gespeichert werden.

5. Auslieferungs- und Verarbeitungsgarantien

Java Messaging System (JMS):

ActiveMQ bietet Transaktionsunterstützung, was bedeutet, dass Nachrichten innerhalb einer Transaktion gesendet und empfangen werden können. Dies bietet eine stärkere Garantie für die Nachrichtenverarbeitung, da entweder alle Nachrichten einer Transaktion erfolgreich verarbeitet oder im Fehlerfall alle zurückgerollt werden.

JMS bietet verschiedene Auslieferungsgarantien:

At-Most-Once: Eine Nachricht wird höchstens einmal gesendet. Bei Fehlern erfolgt keine erneute Sendung. At-Least-Once: Nachrichten werden mindestens einmal zugestellt, können aber im Fehlerfall dupliziert werden. Exactly-Once: Jede Nachricht wird genau einmal zugestellt. Dies ist die strengste Form der Zustellung.

Apache Kafka:

Kafka bietet ähnliche Auslieferungsgarantien wie JMS und legt einen besonderen Schwerpunkt auf die Exactly-Once-Zustellung.

Apache Beam:

Beam garantiert im Streaming-Modus die einmalige Verarbeitung jeder Nachricht durch Checkpointing und Wiederherstellungsmechanismen.

Complex Event Processing (CEP) mit EPL / Esper:

CEP mit EPL/Esper bietet keine spezifischen Garantien für die Auslieferung oder Verarbeitung von Nachrichten. Der Fokus liegt auf der schnellen Echtzeitverarbeitung.

6. Sicherheitsmaßnahmen**Java Messaging System (JMS):**

ActiveMQ, eine populäre Implementierung von JMS, unterstützt verschiedene Authentifizierungsmechanismen und ermöglicht detaillierte Zugriffskontrollen. Dies beinhaltet, welche Benutzer oder Gruppen Nachrichten senden oder empfangen dürfen. TLS/SSL wird für verschlüsselte Verbindungen unterstützt.

Apache Kafka:

Kafka bietet SSL/TLS zur Datenübertragungsverschlüsselung und unterstützt Authentifizierungsmechanismen über Benutzername/Passwort oder Zertifikate.

Apache Beam:

Beam selbst hat keine eigenen Sicherheitsmaßnahmen und ist abhängig von der gewählten Ausführungsumgebung. Entwickler müssen selbst entscheiden, welche Umgebung und Sicherheitsmaßnahmen angemessen sind.

Complex Event Processing (CEP) mit EPL / Esper:

CEP mit EPL/Esper bietet keine spezifischen Sicherheitsmaßnahmen.

7. Programmiermodelle

Java Messaging System (JMS):

JMS bietet keine speziellen Funktionen für die Aggregation oder Gruppierung von Ereignissen. Entwickler müssen eigene Lösungen für komplexere Verarbeitungslogiken implementieren.

Apache Kafka:

Kafka's Streams-Processor-API bietet Verarbeitungsfunktionen wie Aggregationen, Transformationen und Windowing. KSQL ermöglicht SQL-ähnliche Abfragen. Die API ist stark in der Verarbeitung von Datenströmen mit Echtzeitbezug.

Apache Beam:

Beam bietet eine deklarative Pipeline-Definition mit vielfältigen Transformationsmöglichkeiten. Es unterstützt parallele Berechnungen und ermöglicht Windowing auf Basis der Ereigniszeit.

Complex Event Processing (CEP) mit EPL / Esper:

EPL/Esper ist spezialisiert auf die Verarbeitung komplexer Ereignismuster in Echtzeit. Es ermöglicht die Definition spezifischer Muster, Aggregation und zeitbasierte Operationen, ist jedoch in seiner Typsicherheit eingeschränkt, da die Syntax string-basiert ist.

8. Skalierungsmöglichkeiten

Java Messaging System (JMS):

JMS ermöglicht Clustering für horizontale Skalierung, was bedeutet, dass mehrere Broker laufen können, um die Last zu verteilen. Es bietet jedoch darüber hinaus keine speziellen Skalierungsfunktionen.

Apache Kafka:

Kafka zeichnet sich durch hohe Skalierbarkeit aus. Es unterstützt die Partitionierung von Topics, was die Lastverteilung erleichtert, und bietet Replikation zur Verbesserung der Verfügbarkeit. Mehrere Broker können eingesetzt werden, um die Kapazität des Clusters zu erhöhen, und es gibt flexible Konfigurationsmöglichkeiten, um auf unterschiedliche Anforderungen zu reagieren.

Apache Beam:

Die Skalierung bei Apache Beam hängt von der verwendeten Ausführungs-Engine ab, wie zum Beispiel Google Dataflow. Beam erlaubt die Konfiguration der Parallelität von Transformationen, was eine effizientere Datenverarbeitung ermöglicht.

Complex Event Processing (CEP) mit EPL / Esper:

CEP/EPL verarbeitet Datenströme in Echtzeit und ist effizient bei hohen Durchsatzraten. Skalierbarkeit kann durch Verteilung der Ereignisverarbeitung auf verschiedene Knoten erreicht werden, jedoch bietet es ansonsten keine spezifischen Skalierungsmöglichkeiten.

9. Ausfallsicherheit**Java Messaging System (JMS):**

JMS selbst bietet keine Ausfallsicherheit, aber es gibt Methoden, um dies durch verschiedene Cluster und Failover-Mechanismen zu erreichen.

Apache Kafka:

Kafka gewährleistet hohe Verfügbarkeit durch Replikation und Leader-Election. Im Falle eines Ausfalls wird die Replikation zur Verfügung gestellt. Zookeeper wird zur Verwaltung verwendet, was die Verfügbarkeit und Konsistenz erhöht.

Apache Beam:

Die Ausfallsicherheit bei Apache Beam hängt von der Ausführungsumgebung ab und beinhaltet oft fehlertolerante Mechanismen wie automatisches Checkpointing. Im Fehlerfall kann die Umgebung den Pipeline-Zustand wiederherstellen.

Complex Event Processing (CEP) mit EPL / Esper: CEP/EPL bietet keine spezielle Ausfallsicherheit, da der Fokus auf Echtzeitverarbeitung liegt.

10. Typsicherheit

Java Messaging System (JMS):

JMS unterstützt verschiedene Datentypen, die auf Konsumentenseite entsprechen müssen. Bei der Verwendung von Object Messages gibt es jedoch keine Typsicherheit. Fehler werden durch Exceptions erkannt, die zur Laufzeit zum Absturz führen können.

Apache Kafka:

Kafka ist nicht streng typisiert; Nachrichten werden als Byte-Arrays gesendet. Die Typsicherheit liegt in der Verantwortung des Entwicklers, wobei Schema Registry häufig für Typsicherheit verwendet wird.

Apache Beam:

Beam bietet Typsicherheit durch SDKs in verschiedenen Sprachen. Es ist statisch typisiert und erkennt viele Typfehler während der Laufzeit.

Complex Event Processing (CEP) mit EPL / Esper:

EPL/Esper unterstützt Typsicherheit in der Ereignisabfrage, aber die Definition als String kann syntaktisch unsicher sein. Es kann vorkommen, dass Konsumenten Ereignistypen erhalten, die sie nicht erwarten.

11. Unterstützte Programmiersprachen

Java Messaging System (JMS): Es gibt die Möglichkeit JMS mit Java zu implementieren. Wir haben uns dafür entschieden mit Kotlin zu arbeiten, denn JVM-basierte wie auch Scala oder Clojure sind ebenso möglich. Für andere Programmiersprachen gibt es aber ebenso Clients, um mit JMS zu arbeiten

Apache Kafka:

Kafka unterstützt Java und bietet Clients in zahlreichen anderen Sprachen, einschließlich Scala und Kotlin.

Apache Beam:

Beam kann in Java, Python und Kotlin verwendet werden.

Complex Event Processing (CEP) mit EPL / Esper:

EPL/Esper ist auf Java und andere JVM-basierte Sprachen ausgerichtet.

4.2 Persönliche Einschätzung

Die ausgewählten Technologien wurden auf verschiedenen Ebenen im vorherigen Kapitel untersucht und evaluiert.

Unserer Meinung nach macht JMS genau das, was es machen soll. Es war sehr verständlich und straight-forward, hat aber nicht sehr viel mehr Potenzial. Die Art und Weise wie JMS funktioniert wird in Zukunft sicherlich noch relevant bleiben, aber JMS als einzelnes Tool eher weniger.

Kafka als Message Broker hat sich als zuverlässig erwiesen, aber die Windows-Kompatibilität war leider etwas enttäuschend, denn man hat keine gute Anleitung gefunden. Wir fanden es gut, diese Technologie kennen zu lernen, da sie in vielen Firmen eingesetzt wird und von Bedeutung ist. Weiterhin denken wir, dass die Technologie auch in Zukunft weiterhin verwendet werden wird auch wenn vielleicht nur die Kerngedanken dahinter in Form einer weiterentwickelten Technologie fortlebt.

Apache Beam bietet eine flexible und skalierbare Plattform, aber der Einstieg erforderte Zeit, insbesondere wenn man nicht mit Datenstromverarbeitung vertraut ist. Das Pipelining war dennoch gut verständlich, aber kleine Fehler wie das Setzen des Coders haben Probleme verursacht. Wir denken, dass es für die Zukunft weiterhin ein verständliches Tool und weiterhin eine gute Alternative bleiben wird, wenn Streamingdaten verarbeitet werden soll.

Die Dokumentation von Esper war leider sehr veraltet und kaum hilfreich. Es gab zahlreiche Probleme, da Events anfangs nach dem ersten Event kein weiteres Event in das Laufzeitsystem erstellt wurde und es den Anschein hatte, dass es in der Schleife hängen geblieben ist und das Programm nicht weiterlief ohne Hinweise auf Fehlermeldungen. Weiterhin ist die Formulierung der Statements in Strings nicht gerade einfach, um Fehler in der Syntax zu erkennen, welche dann erst zur Kompilierzeit zu Fehlern führt. Für unsere Komplexität der Events fanden wir die Verwendung von Esper und EPL nicht empfehlenswert. Trotzdem denken wir, dass Esper ein wichtiges Werkzeug in Bereichen sein kann, in denen schnelle Datenanalyse und Reaktion erforderlich sind.

Versicherung über redliches wissenschaftliches Arbeiten

Hiermit versichern wir, dass wir die vorliegende Arbeit selbstständig verfasst und erstellt haben. Wir versichern, dass wir nur zugelassene Hilfsmittel und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Ferner versichern wir, dass wir alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gemäß gängiger wissenschaftlicher Zitierregeln korrekt zitiert und als solche gekennzeichnet haben. Darüber hinaus versichern wir, dass alle verwendeten Hilfsmittel, wie KI-basierte Chatbots (bspw. ChatGPT), Übersetzungs- (bspw. DeepL), Paraphrasier- (bspw. Quillbot) oder Programmier-Applikationen (bspw. Github Copilot) vollumfänglich deklariert und ihre Verwendung an den entsprechenden Stellen angegeben und gekennzeichnet haben.

Wir sind uns bewusst, dass die Nutzung maschinell generierter Texte keine Garantie für die Qualität von Inhalten und Text gewährleistet. Wir versichern, dass wir uns textgenerierender KI-Tools lediglich als Hilfsmittel bedient haben und in der vorliegenden Arbeit unser gestalterischer Einfluss überwiegt. Wir verantworten die Übernahme jeglicher von uns verwendeter maschinell generierter Textpassagen vollumfänglich selbst. Auch versichern wir, die „Satzung der Hochschule Furtwangen (HFU) zur Sicherung guter wissenschaftlicher Praxis“ vom 27. Oktober 2022 zur Kenntnis genommen zu haben und uns an den dortigen Ausführungen zu orientieren.

Uns ist bewusst, dass unsere Arbeit auf die Benutzung nicht zugelassener Hilfsmittel oder Plagiate überprüft werden kann. Auch haben wir zur Kenntnis genommen, dass ein Verstoß gegen § 10 bzw. § 11 Absatz 4 und 5 der Allgemeinen Teile der HFU-SPOen zu einer Bewertung der betroffenen Arbeit mit der Note 5 oder mit «nicht ausreichend» und/oder zum Ausschluss von der Erbringung aller weiteren Prüfungsleistungen führen kann.

Furtwangen, 28.01.2024

Lara Meister, Jonathan Reißer