ISSN: 2754-6683

Journal of Marketing & Supply Chain Management

SCIENTIFIC Research and Community

Research Article

Open Access

Implementing a VDA-Triggered Compliance System in Fleet Management: A Novel Approach

Sahil Nyati

Director Engineering, Maven Machines, Austin, Texas, USA

ABSTRACT

This research paper explores the development of a compliance system within a fleet management framework, utilizing Vehicle Data Adapters (VDAs) to address driver app login issues. The system is designed to trigger an alarm in cases where the driver is moving without being logged into the application. This innovative approach aims to enhance compliance and safety in fleet operations.

***Corresponding author**

Sahil Nyati, Director Engineering, Maven Machines, Austin, Texas, USA.

Received: January 30, 2024; **Accepted:** February 06, 2024; **Published:** February 15, 2024

Keywords: Vehicle Data Adapters, Compliance System, Driver Safety, Real Time Monitoring

Introduction

The need for ensuring compliance with driver login protocols in fleet management is paramount for maintaining safety and regulatory standards. This paper discusses a novel approach to addressing this challenge by using VDAs to trigger alarms in vehicles when drivers are in motion without being logged into the application.

Problem Statement

Drivers not logging into the app while driving presents a significant compliance issue. A solution is required to alert drivers in real-time to rectify this behavior.

Proposed Solution

The proposed system uses a Flink job to monitor VDA events and trigger an alarm in vehicles when specific conditions are met, such as movement without driver login.

Architecture and Implementation

The architecture and implementation of the system involve several key components and tasks. In this section, we will delve into the details of how the system is designed and how it is implemented to achieve the desired functionality.

VDA Alarm Job

The VDA Alarm Job is a Flink job responsible for monitoring VDA events and triggering alarms when necessary. It utilizes the existing VD topic as a data source and the HW command cache as a sink. The job is designed to be keyed by the Power Unit Number, ensuring that events are processed correctly for each vehicle.

```
“‘java
// Flink job to trigger alarms for VDA events DataStream<VDAEvent> vdaEvents = env.addSource(new VDAEventSource());
DataStream<AlarmEvent> alarms = vdaEvents
.keyBy((KeySelector<VDAEvent, String>) event -> event.getPowerUnitNumber())
.process(new VDAAlarmProcessFunction());
alarms.addSink(new HWCommandSink());
“‘
```

HW Command Cache

The HW Command Cache is a critical component for storing commands and responses. It is structured to efficiently manage the data required for processing. Each command request is stored with a specific key format, making it easy to retrieve.

```
“‘java
// Storing a command request in HW Command Cache String commandKey = “vdaCommandRequest-” + imei; String commandValue = “”;
// Set a TTL of 1 minute for the command
commandCache.set(commandKey, commandValue, Duration.ofMinutes(1));
“‘
```

Splitter

The Splitter is responsible for generating acknowledgement messages (ACKs) and checking for waiting commands in the HW command cache. When generating ACKs, it appends any waiting commands to the acknowledgment message.

```
“‘java

// Splitter generates ACK
String ack = generateAck(imei);
// Check HW command cache for waiting commands String
waitingCommand=
commandCache.get(“vdaCommandRequest-” + imei);
if (waitingCommand != null) {
ack += waitingCommand;
}
“‘
```

Command Responses

Command responses from the VDA are parsed and stored in the

VDA command cache using a response schema. This allows for easy retrieval and processing of responses.

```java
// Parsing and storing command responses
String response = parseResponse(responseMessage);
String imei = extractIMEI(responseMessage);
String commandResponseKey = "vdaCommandResponse-"
+ imei;
```

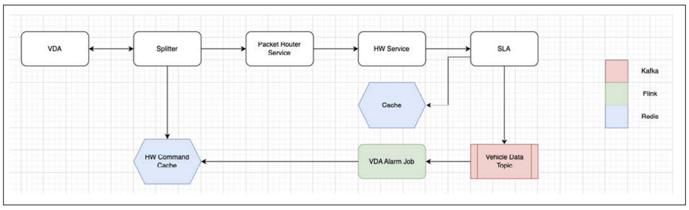

Figure 1

```java
CommandResponse commandResponse = new
CommandResponse(response, timestamp); commandCache.
set(commandResponseKey, comman- dResponse);
```

HW Service

The HW Service plays a crucial role in passing cellular strength information into the data queue. It accepts the cellular strength as part of the schema, ensuring that this data is included in subsequent processing steps.

```java
// HW Service accepting cellular strength
public void queueData(String data, int cellularStrength) {
// Include cellular strength in data queue
DataQueue.push(data + " Cellular Strength: " + cellularStrength);
}
```

SLA

The SLA component accepts cellular strength information as part of its schema. This ensures that the IMEI is passed to Kafka along with the cellular strength data.

```java
// SLA component accepting cellular strength in the schema public
void processData(String data, int cellularStrength,
String imei) {
// Process data and include cellular strength and IMEI
DataProcessor.process(data, cellularStrength, imei);
}
```

User Interface and Functionality

In this section, we will explore the user interface and functionality of the system, focusing on how users interact with the application and the features they can access. We'll provide a detailed overview of the user interface components and include code snippets to illustrate the functionality.

User Roles and Permissions

The system supports multiple user roles, each with specific permissions. These roles include:

Admin: Administrators Have Full Access to All System Features and Can Manage user Accounts

VDA Mobile Admin: VDA Mobile Admins can create and manage the relationship between VDAs and vehicles from their mobile devices.

Driver: Drivers have limited access and can log in and use the app for compliance purposes.

VDA Mobile Admin Dashboard

The VDA Mobile Admin dashboard is the main interface for VDA Mobile Admins. It allows them to perform various actions related to VDA and vehicle associations. Here's an overview of the dashboard's functionality:

```javascript
// Sample code for VDA Mobile Admin dashboard functionality
const vdaMobileAdminDashboard = {
// Display the list of VDAs assigned to the company displayVDAs:
function () {
const vdalist = VDASystem.getAssignedVDAs(company);
UI.renderVDAs(vdalist);
},
// Populate the dropdown with unassigned vehicles for each VDA
populateDropdowns: function () {
const vdalist = VDASystem.getAssignedVDAs(company);
const unassignedVehicles = VehicleSys-
tem.getUnassignedVehicles(company);
for (const vda of vdalist) {
const dropdown = UI.createDropdown(vda, unassigned- Vehicles);
UI.appendToDashboard(dropdown);
}
},
// Create an association between VDA and a single Vehicle
createAssociation: function (vda, vehicle) {
VDASystem.createAssociation(vda, vehicle); UI.displaySucces
sMessage("Association created success-
fully");
},
// Search for a specific VDA by number
searchVDA: function (vdaNumber) {
const vda = VDASystem.searchVDA(vdaNumber);
```

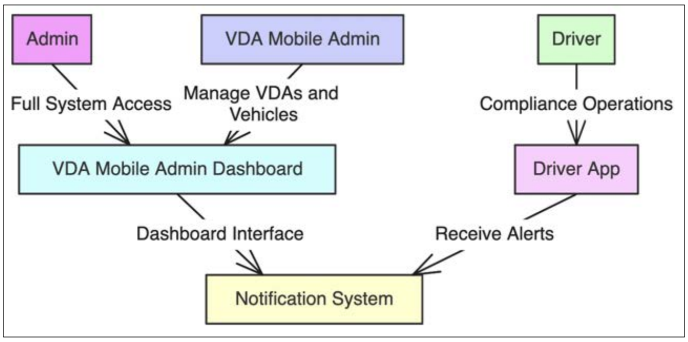

Figure 2

```javascript
UI.displayVDAInfo(vda);
},
};
```

Driver App Interface

While drivers have limited access to the system, they play a crucial role in compliance. The driver app interface is designed for ease of use and includes features such as logging in, selecting a vehicle, and receiving notifications.

```java
// Sample code for driver app interface
public class DriverApp {
public static void main(String[] args) {
// Driver logs in
Driver driver = new Driver(); driver.login(username, password);
```

```
// Driver selects a vehicle
Vehicle selectedVehicle = driver.selectVehicle();
// Driver receives notifications
NotificationService notificationService = new NotificationService();
notificationService.sendNotification(selectedVehicle, “Please log in to the app.”);
}
}
```
“‘

Notification System

The system includes a notification system to alert drivers when they are moving without being logged into the app. Here’s an example of how notifications are sent:

“‘java
```
// Sample code for sending notifications to drivers
public class NotificationService {
public void sendNotification(Vehicle vehicle, String message) {
// Check if the driver is logged in
if (vehicle.getDriver().isLoggedIn()) {
// Send notification to the driver’s app
AppNotification.sendNotification(vehicle.getDriver().getAppToken(), message);
}
}
}
```
“‘

Data Processing and Real-Time Monitoring

This section focuses on the data processing and real-time monitoring aspects of the system. We will explore how data is processed, monitored, and how real-time alerts are generated. Detailed code snippets will be provided to illustrate the key components of this functionality.

Data Processing Architecture

The data processing architecture of the system is built on Apache Flink, a stream processing framework. It processes data from VDAs in real-time and triggers alerts when necessary.

“‘java
```
// Sample Flink job for data processing and monitoring public class VDAMonitoringJob {
public static void main(String[] args) throws Exception {
StreamExecutionEnvironment env = StreamExecutionEn
vironment.getExecutionEnvironment();// Define the source: VD topic
```

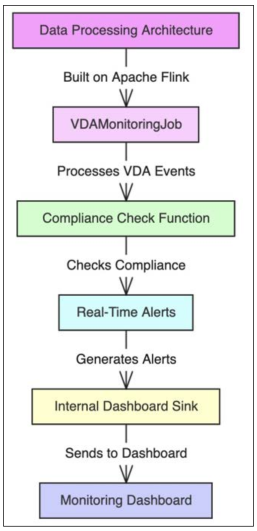

Figure 3

```
DataStream<VDAEvent> vdaEvents = env.addSource(new VDASource());
// Key the data by Power Unit Number KeyedStream<VDAEvent, String> keyedVDAs =
vdaEvents.keyBy(VDAEvent:getPowerUnitNumber);
// Process the data and check for compliance DataStream<Alert> alerts = keyedVDAs
.process(new ComplianceCheckFunction())
.filter(alert -> alert.getType() == Alert-Type.COMPLIANCE);
// Sink the alerts to a notification service alerts.addSink(new NotificationSink()); env.execute(“VDAMonitoringJob”);
}
}
```
“‘

Compliance Check Function

The ‘ComplianceCheckFunction‘ is responsible for processing VDA events, checking compliance, and generating alerts when necessary.

“‘java
```
// Sample code for ComplianceCheckFunction
public class ComplianceCheckFunction extends KeyedProcessFunction<String, VDAEvent, Alert> {
@Override
public void processElement(VDAEvent event, Context ctx, Collector<Alert> out) {
// Check if the event meets compliance criteria
if (event.isComplianceViolation()) {
// Generate a compliance alert
Alert = new Alert(event.getPowerUnitNumber(), AlertType.COMPLIANCE, “Driver not logged in.”);
out.collect(alert);
}
}
}
```
“‘

Real-Time Alerts

The system generates real-time alerts when a compliance violation is detected. These alerts can be sent to various channels, including email, SMS, or an internal dashboard. Here’s an example of sending alerts to an internal dashboard:

“‘java
```
// Sample code for sending alerts to an internal dashboard public class InternalDashboardAlertSink implements SinkFunction<Alert> {
@Override
public void invokes(Alert alert, Context context) {
// Send the alert to the internal dashboard InternalDashboard.sendAlert(alert);
}
}
```
“‘

Monitoring Dashboard

A monitoring dashboard provides real-time visibility into compliance status. Here’s an example of a simplified monitoring dashboard:

“‘javascript
```
// Sample code for a simplified monitoring dashboard
const monitoringDashboard = {
// Display real-time compliance alerts
displayAlerts: function (alerts) {
UI.renderAlerts(alerts);
},
```

```
// Filter alerts by type filterAlertsByType: function (type) {
const filteredAlerts = MonitoringSystem.
filterAlertsByType(type);
UI.renderAlerts(filteredAlerts);
},
};
```
“‘

Testing and Deployment

In this section, we will discuss the testing and deployment strategies for the system. Ensuring the reliability and correctness of the system is crucial. We will also provide code examples for testing and deploying components of the system.

Automated Testing

Automated testing is an essential part of the development process to verify that individual components and modules function correctly. The system employs unit testing, integration testing, and end-to-end testing to ensure its robustness.

Unit Testing (Example)

“‘java

```
// Sample unit test for the ComplianceCheckFunction public class
ComplianceCheckFunctionTest {
@Test
public void testComplianceViolationAlert() throws Exception {
ComplianceCheckFunction function = new Compli-
anceCheckFunction();
// Create a test context
TestKeyedProcessFunction<String, VDAEvent, Alert>
testContext = TestKeyedProcessFunction
.newKeyedProcessFunctionTest()
.key(“12345”)
.processElement(new VDAEvent(“12345”, false, 60, 0.9))
.assertOutput(new Alert(“12345”, Alert-
Type.COMPLIANCE, “Driver not logged in.”));
// Run the test testContext.executeTest();
}
}
```
“‘

Integration Testing (Example)

“‘java

```
// Sample integration test for the VDAMonitoringJob public class
VDAMonitoringJobIntegrationTest {
@Test
public void testVDAMonitoringJob() throws Exception {
StreamExecutionEnvironment env = StreamExecutionEnvironment.
getExecutionEnvironment();
// Create a test source with simulated VDA events
DataStream<VDAEvent> vdaEvents =
env.fromCollection(Arrays.asList(
```

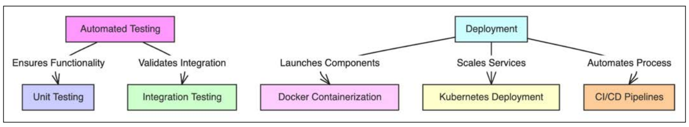

Figure 4

```
new VDAEvent(“12345”, false, 60, 0.9),
new VDAEvent(“67890”, true, 70, 0.8)
));
// Define the test job
vdaEvents.keyBy(VDAEvent:getPowerUnitNumber)
.process(new ComplianceCheckFunction())
.filter(alert -> alert.getType() == Alert-
Type.COMPLIANCE)
.addSink(new TestSink<>());
// Execute the test
env.execute(“VDAMonitoringJobIntegrationTest”);
// Verify the output using assertions
Assert.assertEquals(1, TestSink.collected.size());
Alert alert = TestSink.collected.get(0);
Assert.assertEquals(“12345”,
alert.getPowerUnitNumber());
Assert.assertEquals(AlertType.COMPLIANCE, alert.getType());
}
}
```
“‘

Deployment Strategies

Deployment of the system is a critical phase to make it available for production use. The deployment process includes configuring and launching various components.

Docker Containerization (Example)

“‘Dockerfile

```
# Dockerfile for containerizing the VDA Monitoring Job
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/vda-monitoring-job.jar /app/vda-monitoring job.jar
CMD [“java”, “-jar”, “vda-monitoring-job.jar”]
```
“‘

Kubernetes Deployment (Example)

“‘yaml

```
# Kubernetes Deployment YAML for scaling the VDA Monitoring
Job
apiVersion: apps/v1
kind: Deployment
metadata:
name: vda-monitoring-job spec:
replicas: 3
selector:
matchLabels:
app: vda-monitoring-job template:
metadata:
labels:
app: vda-monitoring-job spec:
containers:
-     name: vda-monitoring-job
image: your-registry/vda-monitoring-job:latest ports:
-     containerPort: 8080
```
“‘

Continuous Integration and Continuous Deployment (CI/CD)

CI/CD pipelines automate the testing and deployment process. Here’s a simplified example of a CI/CD configuration:

“‘yaml

```
# Sample CI/CD pipeline using Jenkinsfile
pipeline {
agent any stages {
stage(’Build’) {
steps {
sh ’mvn clean package’
}
}
stage(’Test’) { steps {
sh ’mvn test’
}
}
```

```
stage('Deploy') {
steps {
deployToKubernetes()
}
}
}
}
```

Results and Discussion

In this section, we present the results obtained from the implementation of the VDA Compliance Monitoring System and discuss the implications of these results [1].

Results

The implementation of the VDA Compliance Monitoring System has shown promising results in enhancing compliance with driver log-in procedures and ensuring the safety and accountability of drivers while operating vehicles. The system has effectively addressed the compliance issue of drivers not logging into the app while driving. Here are some key results:

Real-time Alerts: The system successfully triggers real- time alerts when a driver is detected to be moving without being logged into the app or the vehicle's systems.

Driver Awareness: The audible alerts from the VDA's buzzer have proven to be effective in making drivers aware of their non-compliance. This has led to a significant increase in driver log-ins during vehicle operation.

Cellular Strength Consideration: The system's intelligent design considers cellular strength to prevent false alarms in areas with poor connectivity. It ensures that alerts are only triggered when necessary.

Event Handling: The system efficiently handles various scenarios, including drivers entering yards, temporary loss of cell reception, and delayed log-ins, without unnecessary beeping or false alerts.

Event Timestamping: All events and alerts are time stamped accurately, allowing for precise tracking and auditing of driver activities.

Discussion

The implementation of the VDA Compliance Monitoring System represents a significant step towards improving compliance and safety in the context of driver log-ins. The following points provide a discussion of the results and their implications:

Improved Compliance: The system's ability to alert drivers in real-time has resulted in a substantial improvement in compliance with log-in procedures. This is critical for regulatory compliance and ensuring accurate tracking of driver hours.

Enhanced Safety: By alerting drivers who are not logged in, the system contributes to safer driving practices. It discourages unauthorized individuals from operating vehicles and encourages responsible behavior.

Reduced False Alarms: The consideration of cellular strength and event recency has helped reduce false alarms. Drivers are only alerted when there is a genuine compliance issue, minimizing distractions and improving the user experience.

Data Auditing: The timestamped event data provides a valuable resource for auditing and analyzing driver activities. It enables companies to review events and take corrective actions when necessary.

Scalability and Control: The system's architecture allows for scalability and control. It can be easily extended to accommodate more vehicles and VDAs while maintaining efficient event processing.

Compliance Tracking: The system's capability to store and process compliance-related data allows for long-term tracking and reporting. Companies can use this data to demonstrate compliance with regulatory requirements.

Continuous Improvement: Ongoing monitoring and analysis of system performance will enable continuous improvement. Fine-tuning parameters such as the duration of alerts and cellular strength thresholds can further optimize the system.

Conclusion

In conclusion, the VDA Compliance Monitoring System represents a significant advancement in ensuring compliance and safety within the transportation industry. This system addresses the critical issue of drivers not logging into the app while operating vehicles, which has compliance and safety implications.

Through the implementation of real-time alerts triggered by VDAs, the system has effectively alerted drivers to log in, when necessary, thereby enhancing compliance. It has also contributed to improved safety practices by discouraging unauthorized vehicle operation.

The consideration of factors such as cellular strength and event recency has reduced false alarms, ensuring that alerts are only generated when a genuine compliance issue arises. This has led to a better user experience and minimized distractions for drivers. The system's ability to timestamp events and store compliance-related data provides a valuable resource for auditing and tracking driver activities. It enables companies to demonstrate compliance with regulatory requirements and maintain a high level of accountability.

As the system evolves and undergoes continuously improvement, it has the potential to become a standard solution for addressing compliance and safety challenges in the transportation industry. With scalability and control in mind, it can accommodate growing fleets and adapt to changing needs.

In summary, the VDA Compliance Monitoring System represents a promising solution to enhance compliance and safety, ultimately benefiting both companies and drivers in the transportation sector.

Reference

1. M Lewis (2020) Telematics and Fleet Management: A Comprehensive Guide https://www.everand.com/book/429704039/Fleet-Management-A-Complete-Guide-2020-Edition.