Exploring the Test Driven Development of a Big Data Infrastructure Examining Gun Violence Incidents in the United States of America

Daniel Staegemann[a], Malte Rathjens[b], Hannes Hinniger, Vivian Schmidt[c] and Klaus Turowski[d]

Magdeburg Research and Competence Cluster VLBA, Otto-von-Guericke University Magdeburg, Magdeburg, Germany

Keywords: Software Engineering, Big Data, Test Driven Development, TDD, Testing, Quality Assurance, Microservice, Prediction, Gun Violence.

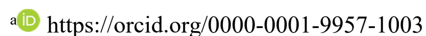
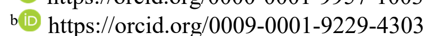
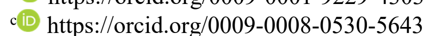
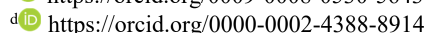
Abstract: Big data (BD) and the systems used for its harnessing heavily impact many aspects of today's society and it has been repeatedly shown that they can positively impact the operations of organizations that incorporate them. However, creating and maintaining these applications is extremely challenging. Therefore, it is necessary to pay additional attention to the corresponding quality assurance. One software engineering approach that combines high test coverage, the enabling of comprehensive regression tests, but also positively impacts the developed applications' design, is test driven development. Even though by now it has a somewhat long history in software development in general, its use in the context of BD engineering is not common. However, an approach for the test driven development of BD applications that is based on microservices has been proposed rather recently. To gain further insights into its feasibility, the publication at hand explores its application in the context of a prototypical project implementation. Hereby, the chosen use case is the analysis and prediction of gun violence incidents in the United States of America, which also incorporates NFL match game data, under the assumption that the games could potentially influence the occurrence of such incidents.

1 INTRODUCTION

In the contemporary landscape of technological advancement, the proliferation of data has emerged as a significant driver of innovation and progress, presenting both unprecedented opportunities as well as challenges. The exponential growth in the volume, velocity, and variety (Laney 2001), led to a situation, where common technologies were no longer able to keep up with the demands (Chang and Grady 2019), which is generally described by the term big data (BD), even though there is not a single, universally applied definition for it (Diebold 2021; Volk et al. 2022). Since its emergence, BD has been identified as a facilitator of increased productivity (Müller et al. 2018) and shifted how organizations perceive, analyze, and leverage information. Thus, the ability to harness the insights buried within all this data has become instrumental in addressing issues across various domains, like, for instance, enabling evidence-based policymaking and targeted interventions (Höchtl et al. 2016).

However, assuring the quality of the corresponding applications is a challenging task (Staegemann et al. 2019), whose importance also shows in the vastness of the related research endeavors (Ji et al. 2020).

In turn, test driven development (TDD) is a popular approach for developing software. Due to its associated advantages (Agha et al. 2023) that could be also valuable in the BD realm, the idea was proposed (Staegemann et al. 2020b) to apply it not only for common software but also in the context of big data engineering (BDE). However, despite the associated potential advantages and some corresponding works such as (Staegemann et al. 2023), it is still rather underexplored in the context of BD and the related applications. To contribute to the

 [a] https://orcid.org/0000-0001-9957-1003
 [b] https://orcid.org/0009-0001-9229-4303
 [c] https://orcid.org/0009-0008-0530-5643
 [d] https://orcid.org/0000-0002-4388-8914

Staegemann, D., Rathjens, M., Hinniger, H., Schmidt, V. and Turowski, K.
Exploring the Test Driven Development of a Big Data Infrastructure Examining Gun Violence Incidents in the United States of America.
DOI: 10.5220/0012829600003764
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 21st International Conference on Smart Business Technologies (ICSBT 2024)*, pages 103-114
ISBN: 978-989-758-710-8; ISSN: 2184-772X
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

corresponding knowledge base and help assess its viability, in the following, TDD will be applied to develop a microservice-based BD application that analyzes gun violence incidents in the United States of America (USA). More precisely, the application is designed to provide users with an overview of the expected incidents of gun violence in each state of the USA. By analyzing historical data, probabilities for incidents leading to injuries and/or deaths within a specified time period will be calculated and displayed. The calculations are based on a dataset (Ko 2018) on gun violence in the USA, which records more than 260,000 gun violence incidents from January 2013 to March 2018 (inclusive). This data will be processed along with seasonal patterns, specifically NFL game data (Horowitz 2018), by a predictive microservice to generate forecasts for the expected incidents. The results are then visually presented to the users. Based on its objectives, the application is, therefore, of a descriptive and predictive nature (Roy et al. 2022).

Even though the presented prototypical implementation uses historical data that are available in batch, stream processing will be simulated by transmitting the data in a somewhat continuous way over time, instead of providing it all at once. Thus, even though the dataset is rather small in volume in the context of BD, the developed application still showcases a highly typical BD use case (Volk et al. 2020). Moreover, the developed architecture is designed in a way to allow for future scaling, leaning further into the demands associated with BD.

While the pursued analytical task is generally interesting, it has to be noted that the publications' primary focus is not on the actual content of the use case but on the technical aspects with the goal of further exploring the application of TDD in BDE itself. For this reason, the actual quality of the prediction is of secondary importance. Moreover, the application is only of a prototypical nature and does not aspire to be fully refined.

The publication is structured as follows. Following this introduction, the general TDD approach is described. Afterward, the application's basic design is outlined. This is followed by a section that describes the actual implementation including the testing. Subsequently, the findings are discussed and, finally, a conclusion of the work is given.

2 THE DESIGN

The following section describes the developed application's design, including the BD infrastructure, and the utilized technologies, and broadly outlines the corresponding functionality. To meet the project requirements, a message-driven BD infrastructure setup was selected and automated, with Apache Kafka (Apache Software Foundation 2024a) being specifically chosen as the central component. In the scope of this work, Kafka and its ecosystem component ZooKeeper (Apache Software Foundation 2024b) are harnessed for scalability. This strategic implementation guarantees high availability, allowing the system to effortlessly adapt to growing data volumes and user requirements while maintaining consistent performance and minimizing downtime. Figure 1 provides an overview of the infrastructure created.

For the implementation of the functionalities, the programming language Python is predominantly used due to its widespread use in the data science field. Moreover, the open-source automation tool Ansible (Red Hat 2024) is also based on this programming language. For this reason, both the Kafka *Producer* and *Consumer* in the backend, as well as a Flask

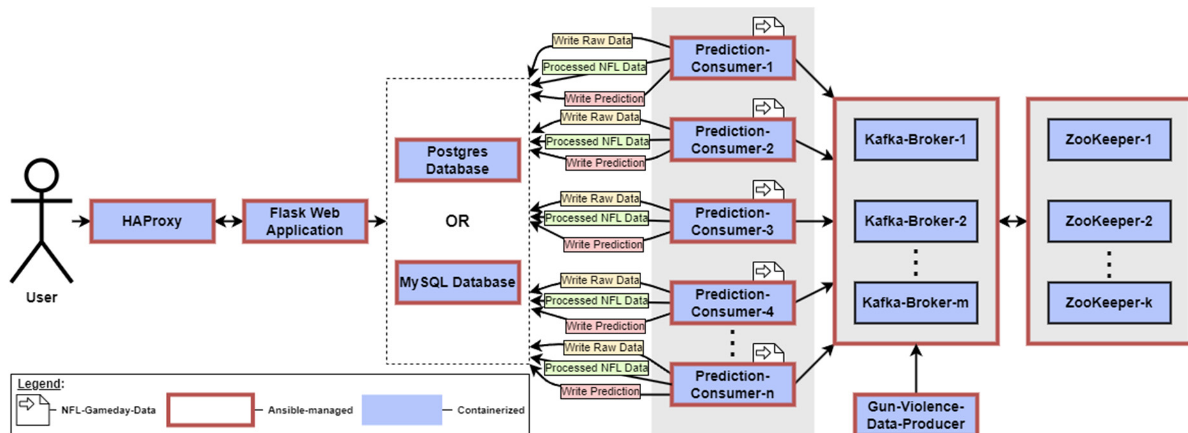


Figure 1: The Developed application's architecture.

Application (Ronacher 2024) on the frontend side, will be based on Python.

2.1 Ansible

Within the scope of this prototypical implementation, Ansible serves as the foundational technology for both, the deployment and the execution of automated tests. To meet the demands of the proposed TDD approach, it is mandated that tests are written for all services provided by Ansible, which are primarily Docker containers (Docker Inc. 2024), before composing the code for the automated deployment. This practice ensures that the functionality and reliability of each service are verified at the outset, laying a robust groundwork for facilitating continuous integration and continuous delivery (CI/CD) while adhering to the principles of TDD.

2.2 Docker

For this implementation, containerization is a fundamental element. The chosen development approach and the defined use case favor the decision to containerize individual functions of the infrastructure and the web application. For this, Docker has been chosen. In line with the prototype's requirements, the installation or general provisioning and subsequent availability check of Docker including Docker Networks are validated by tests that are predefined. The Docker network is essential to ensure that the various microservices can communicate with each other without disruption. With the aid of Docker and Ansible, the infrastructure for processing the data can then be set up and tested.

2.3 Zookeeper

For the operation of the presented application, ZooKeeper is indispensable, managing fundamental Kafka operations such as broker coordination and maintaining state. Its functionality is particularly crucial to ensuring a reliable messaging system and must be considered in the implementation phase when applying TDD.

2.4 Kafka Broker

In this project, a Kafka Broker enables the efficient management and transfer of data streams and is central to high availability and scalability. In connection with TDD, unit and integration tests must be carried out, with particular emphasis placed on correct interaction with the broker. When defining tests, it must be noted that the Kafka Broker is dependent on ZooKeeper.

2.5 Kafka Producer

Within the implemented infrastructure, a Kafka Producer is deployed and tested as a Python container. The Producer reads data on gun violence incidents from a given CSV file and sends it to Kafka. As part of the tests, it must be ensured that the Producer container is not only set up and running but also capable of reaching and processing the provided data.

2.6 Kafka Consumer

The Consumers are responsible for reading and processing the messages, where they filter, pre-process, and subsequently store the data in a database. Within this application, each Consumer is responsible for a certain number of (federal) states, which can vary depending on the number of Consumers. This ensures that with increased data traffic, the deployment of more Consumers and, hence, faster processing can be facilitated.

The NFL Game data is retrieved from a CSV file in the prototype, then processed and written to a database. In a production environment, this data would most likely be pulled from an API. This process takes place at regular intervals to ensure that the latest data is available and can be used by the prediction algorithm. The retrieved NFL data includes the *game ID*, *game day*, *home team*, and *away team*. This data is enriched in pre-processing by mapping the available team data with the *states of origin* of these teams. This information (*game ID*, *game day*, *home team*, *away team*, *home state*, *away state*) is then written to the database in order to make it persistently available to Consumers. A Consumer instance then retrieves the Gun Violence data relevant to it, which are those for the states assigned to the Consumer.

During the pre-processing step, the data provided by the Producer are filtered and cleaned. For instance, only those raw datasets that have at least an *incident ID*, a *date*, and an assigned *state* are processed. Additionally, fields such as *state*, *county*, and *city* are checked and cleansed of simple spelling errors, such as special characters. Moreover, for incidents that lack a value for killed persons, this value is set to zero. These pre-processing steps are essential to meaningfully feed the data into the *fbprophet* (Prophet) algorithm (Facebook Open Source 2024), in accordance with the objective.

The processing of data for the purpose of forecasting, using the Prophet service, subsequently also takes place within the producer. As soon as data for an additional month are available (these are consumed sorted by date), another forecast for the coming 365 days is carried out. This forecast is carried out for each state the Consumer is handling. The results of the forecast, as well as the incident numbers, are written into the connected database.

The Prophet forecasting procedure retrieves all incidents per state, as well as the NFL game dates, where an NFL team from the specific state is involved. With respect to the given data, Prophet will forecast the expected number of incidents and victims.

To complement the existing forecasting mechanism focused on fatalities within the Consumers, an additional Consumer will be introduced to establish predictions related to injuries. This data is already provided by the producer. This expansion introduces a microservice to the architecture that can, in theory, be seamlessly replaced, and is validated and secured by the corresponding existing tests (Staegemann et al. 2020b). Such adaptability needs to be considered not only in a TDD context but also in the actual deployment and maintenance of the system. This ensures that as the project evolves and different analytical needs emerge, the system remains agile and capable of integrating new functionalities with minimal disruption, which is common in real systems (Staegemann et al. 2020a).

For error analysis and monitoring purposes, in this artificial setting (Sonnenberg and Vom Brocke 2012), the preprocessed raw data are additionally stored in the database.

2.7 Database

To have the generated data persistently available for the web application, it is stored in a database. As previously introduced, the database stores a prediction value for each day and state per forecast. Since the forecast always projects one year ahead, after a successful run upon reaching the month-end of a given date, there will be 12 values per date and state in the table, comprising eleven predictions and the actual realized value.

By maintaining all forecasts, not only can transparency be ensured, but also the values can be provided as an additional feature to the users on the front end, allowing them to gain an implicit understanding of the error tolerance.

Within the microservices architecture, the database should be structured to enable the seamless integration and testing of different database services. Additionally, ensuring this modularity aligns with the proposed TDD for BDE approach (Staegemann et al. 2020b). This, in turn, needs to be considered in the implementation phase.

2.8 Access

The user can access the application through a Flask web application, which can be reached via an instance of *HAProxy* (Tarreau 2023), which is an open-source high-performance load balancer and proxy. However, these are of little relevancy to the actual use case and are, therefore, only occasionally mentioned but not further discussed in the following.

3 THE IMPLEMENTATION

The implementation of the showcased application can be generally divided into two logical areas. These are the *infrastructure* and the *application*.

Since the main motivation of the implementation is to showcase the use of TDD in BDE, both of these were strictly implemented following this approach.

3.1 General Implementation Approach

The general approach adopted in this study entails several sequential steps aimed at ensuring efficient implementation. Initially, the implementation process was conceptualized. Subsequently, thorough planning of necessary integration tests and unit tests was conducted to ensure comprehensive coverage. An essential aspect of the TDD is the test implementation prior to the development of code. Thus, the approach facilitates the creation of code that aligns closely with the predefined testing criteria. Finally, the implementation of the actual productive code follows after the completion of the test planning and implementation. The iterative nature of this process allows for the incorporation of additional tests as deemed necessary during the implementation phase.

The development approach for the infrastructure follows TDD principles. In doing so, the following decisions needed to be focused during the conceptualization of the tests.

Emphasizing scalability and efficiency, a primary objective of the endeavor is to construct a microservice-based infrastructure tailored for a BD application. This infrastructure is intended to encompass functionalities such as data creation,

storage, consumption, processing, analysis, and visualization.

To achieve these objectives, a message-driven architecture utilizing Apache Kafka was chosen. This architectural choice necessitates the inclusion of *ZooKeepers*, *Kafka Brokers*, *Kafka Producers*, and *Kafka Consumers*. Additionally, a database is required to store both the processed raw data and predictions based on these data, along with the Flask-based web application.

The web application, the Kafka producer responsible for processing CSV files, as well as the Consumers performing data processing and prediction tasks, are implemented in Python.

A crucial consideration in the design is the scalability of the Consumers to handle high message volumes. Consequently, the Kafka brokers are also engineered with scalability in mind.

Notably, the microservice architecture's flexibility as well as the availability of the corresponding tests allow for the interchangeability of components, including the database, which can be seamlessly switched between *PostgreSQL* and *MySQL*. The robustness of the test framework is paramount to ensure smooth transitioning between these database systems.

3.2 The Infrastructure

In accordance with the development approach, the development process for the infrastructure part adhered to rigorous testing procedures. Before the commencement of any coding activities, the testing strategy and specific test cases were planned. Given that the infrastructure setup was orchestrated using Ansible, the testing framework was conceptualized within the context of Ansible's operational environment.

Leveraging Ansible's modular structure encompassing plays, playbooks, tasks, and roles facilitated the planning of both unit and integration tests. In this framework, a unit is defined as either an individual Ansible task or a cohesive set of tasks (combined as a playbook) aimed at accomplishing a specific objective. Conversely, an integration test is a play consisting of a collection of roles or playbooks that are interdependent in at least one direction.

Each unit test adheres to a standardized procedure, encompassing the following sequential steps:

1) Verification of preconditions to ensure that the specified conditions were not fulfilled prior to executing the unit
2) Provision possible prerequisites
3) Execution of the unit under test
4) Validation to confirm that the prescribed conditions have been successfully met subsequent to the unit's execution
5) Cleanup of any artifacts or residual elements generated during the testing process to maintain the integrity of subsequent tests

By systematically following this testing regimen, the robustness and reliability of the infrastructure were continuously validated, ensuring adherence to defined specifications and facilitating the detection and resolution of potential issues at an early stage of development.

To ensure that each unit is tested properly and that the respective mapping is always clearly visible, there is a test in a test directory that has the same name as the tested unit. In most cases, a unit in this project is a playbook that contains a test playbook with the same name as the setup playbook.

3.2.1 Docker

In accordance with prior specifications, the Docker environment is configured using Ansible. The setup process is designed to be a singular playbook. In order to validate this playbook, initial verification ensures the absence of Docker. Subsequently, execution of the *setup-docker* playbook occurs, followed by validation to confirm Docker installation completion. These procedural steps are combined into an Ansible play with three distinct playbooks:

1) The initial playbook verifies preconditions.
2) The "setup-docker" playbook is imported.
3) The final playbook verifies postconditions.

The preconditions verification playbook utilizes Ansible's *service_facts* module to ascertain the absence of the Docker service. Additionally, the *package_facts* module confirms the non-existence of *container.io*, *docker-ce-cli*, and *docker-ce* packages.

Conversely, the postconditions verification playbook validates the presence of the Docker binary in *service_facts*, confirms the Docker service status as *running*, and verifies the installation of *container.io*, *docker-ce-cli*, and *docker-ce* packages. These validations collectively fulfill the requirements of the unit test.

It is deemed unnecessary to individually test minor tasks such as repository addition or GPG Key insertion, as failure in any of these tasks would result in the failure of the Docker installation playbook.

At this stage, a dedicated integration test is deemed unnecessary, as subsequent container tests inherently verify the integrity of Docker integration.

3.2.2 Docker Network

To enable inter-container communication, especially via Docker container names, a dedicated Docker bridge network is set up. This network configuration is encapsulated within a specialized playbook to facilitate ease of customization or substitution with alternative network setups.

Similar to the testing methodology outlined for Docker, the validation of this playbook can be approached through unit testing via an Ansible play. This combines a *pre-validation* playbook, the import of the *setup-docker-network* playbook, a *post-validation* playbook, and the incorporation of a *cleanup* playbook to remove the network upon test completion.

Pre-validation utilizes the *docker network* command to verify the non-existence of the designated network. An expected outcome is the failure of *docker network inspect NETWORK_NAME* if the network is absent.

Conversely, post-validation involves the verification that *docker network inspect NETWORK_NAME* no longer results in failure, indicating successful network creation.

Given the subsequent testing of inter-container communication during integration tests for components, additional integration tests specific to the Docker network are deemed redundant. The functionality of the Docker network will be implicitly affirmed during these broader integration tests.

The Ansible implementation of the mentioned test, as an example of how these look like, is depicted in Figure 2.

3.2.3 Zookeeper

The infrastructure, centered around Apache Kafka, necessitates the inclusion of *ZooKeepers* to oversee various aspects of the cluster such as Kafka topics and management. In consideration of ensuring high availability as well as scalability, for this implementation, a foundational configuration comprising three ZooKeepers has been established. Scaling the number of ZooKeepers is facilitated by a singular Ansible variable within the inventory.

The ZooKeeper setup, as well as previous setup steps, consists of a single Ansible playbook, and an Ansible play is used to verify the setup-zookeeper unit.

To ensure the correctness of the test, a *pre-validation* playbook within the Ansible unit test is essential. This playbook aims to verify that the specified number, denoted as *N*, of ZooKeepers has not yet been instantiated. This validation process involves iterating over the designated quantity of ZooKeepers and examining their existence through the utilization of the *docker_container_info* Ansible module. This module yields a crucial *exists* fact, indicating the presence or absence of the container.

After preconditions are validated, necessary dependencies are provisioned. This includes the setup of Docker as well as the setup of the Docker bridge network.

Subsequently, a post-validation playbook is executed following the execution of the imported setup-zookeepers playbook. Similar to the pre-validation phase, this post-validation playbook iterates over the specified number of ZooKeepers.

```
#Verify Docker Network is not installed yet
- hosts: all
  tags: test
  tasks:
    - name: Gather facts about docker network
      become: yes
      ansible.builtin.command: "docker network inspect {{ dependencies_setup_docker_network__docker_network_name }}"
      register: tests__docker_network_check_0
      ignore_errors: yes
    - name: Assert that network does not exist yet
      assert:
        that:
          - "{{ tests__docker_network_check_0.rc != 0 }}"

- import_playbook: ../dependencies/setup-docker-network.yaml

#Verify Docker Network is installed now
- hosts: all
  tags: test
  tasks:
    - name: Gather facts about docker network
      become: yes
      ansible.builtin.command: "docker network inspect {{ dependencies_setup_docker_network__docker_network_name }}"
      register: tests__docker_network_check_1
      ignore_errors: yes
    - name: Assert that network does exist
      assert:
        that:
          - "{{ tests__docker_network_check_1.rc == 0 }}"
    - include_tasks: ./cleanup-test-env-tasks.yaml
```

Figure 2: The docker network test.

However, in this phase, the playbook reassesses the existence of each container and further ensures that after a time interval of 20 seconds, the status of the container transitions to *running*, and the ExitCode attains a value of *0*. These additional checks serve to corroborate the successful instantiation and operational state of each ZooKeeper instance since misconfigurations of ZooKeepers will result in an instant fail on start-up.

This validation could be strengthened by parsing the logs of ZooKeeper, but this would lead to a fragile test, which was, therefore, omitted.

Upon completion of the post-validation phase, a *cleanup* task is initiated to remove all containers and associated dependencies. This measure ensures that the testing process leaves no residual artifacts within the system, maintaining the integrity and cleanliness of the testing environment.

Integration tests specific to ZooKeeper are considered redundant. The functionality of ZooKeeper will be implicitly affirmed during integration tests of Kafka.

3.2.4 Kafka Broker

In the context of the developed application, Apache Kafka serves as the central component. Given that the infrastructure design relies on Apache Kafka's integration with ZooKeeper, the testing of Kafka can be construed as an integration test for the ZooKeeper setup described earlier. Consequently, testing Kafka essentially entails testing ZooKeeper as well.

Just as the previously discussed test implementations, the testing process involves the utilization of Ansible Play, which combines various playbooks. Initially, a *pre-validation-checking* Ansible playbook is executed to ascertain the absence of the Kafka container. This verification is conducted using the *docker_container_info* and assert Ansible modules, which validate the absence of the container by assessing the boolean value of *exists*.

Subsequently, the setup entails configuring any requisite preconditions, including *Docker*, *Docker Bridge Network*, and *Zookeepers*. Following this, the *setup-kafka* playbook is imported to initiate the Kafka setup process.

Upon completion of the setup, a *post-validation* step is undertaken. This involves *verifying the existence of the container*, *ascertaining its operational status* as *running*, and *confirming the absence of any error exit codes*. This validation process employs the *docker_container_info* and *assert* Ansible modules, which assess the boolean values of *exists*, the status of the container, and the absence of error exit codes, respectively.

Finally, the *cleanup* phase ensues, wherein any artifacts generated during the testing process are removed to maintain a clean testing environment.

3.2.5 Kafka Producers

In the context of Kafka Producers within the infrastructure setup, it shall be noted that the decision to implement the Kafka Producer in Python is aligned with the utilization of the official Python container image. As outlined previously, the Producer connects to Kafka, establishes the *gun-violence-raw-data* Kafka topic, reads the *Gun Violence Dataset* from a CSV file, and subsequently writes each line to the specified Kafka topic. Upon successful execution of the *send command*, a log entry is generated, indicating the delivery of messages to the *gun-violence-raw-data topic*. The log message is *Message delivered* to the *gun-violence-raw-data* topic.

From an infrastructure standpoint, the entire setup of the Kafka Producer constitutes a unit, necessitating a unit test for the infrastructure setup using Ansible. This test encompasses the verification of a running Kafka broker, which relies on ZooKeeper, as well as the existence of a functional Docker bridge network and Docker. This comprehensive integration test is designed to specifically evaluate the Kafka Producer unit.

The testing methodology for the *setup-kafka-producer* unit is structured similarly to the tests conducted for other components discussed earlier. *Pre-validation* checks are conducted, *prerequisites* are established, the *setup-kafka-producer* playbook is imported, *post-validation* assessments are performed, and a *cleanup* playbook is executed to remove any generated artifacts.

The pre-validation stage involves confirming the existence of containers. From an infrastructure perspective, the validation of the Kafka Producer's functionality is based on the logging of the specific message upon successful message delivery to Kafka. This validation is incorporated into the post-validation playbook, wherein the Ansible *command* module is combined with the *until* module to retrieve container logs intermittently over a 30-second period, with a 3-second interval, to verify the occurrence of the expected log message.

3.2.6 Kafka Consumers

In alignment with the architectural decisions, the Kafka Consumer is implemented utilizing the official Python container image. As previously outlined, the

Consumer connects to Kafka, subscribes to the *gun-violence-raw-data* Kafka topic, retrieves messages, processes them, and subsequently stores both predictions and processed raw data in a designated database (either MySQL or PostgreSQL). Upon successful message processing, the Consumer logs a confirmation message, denoted as *Successfully processed message*.

Similar to the approach taken for the Kafka Producer, the entire setup of the Kafka Consumer is regarded as a unit from an infrastructure standpoint. Therefore, it necessitates a unit test for the infrastructure setup utilizing Ansible. This comprehensive test encompasses the verification of a running *database*, Kafka *Producer*, Kafka *Broker* (which in turn requires *ZooKeeper*), as well as the existence of a functional *Docker bridge network* and *Docker*.

As elaborated previously, the Kafka Consumer script also requires unit tests from a software development perspective, which will be discussed in section 3.3.

The Ansible Play designed to test the *setup-kafka-consumer* unit mirrors the structure of the Kafka Producer test described earlier. However, there are notable differences:

- Additional prerequisites include the presence of the database and Kafka Producer.
- The expected log message during post-validation differs.

From an infrastructure perspective, the validation of the Kafka Consumer's functionality depends on the logging of a specific message upon successful message processing. This validation process is similar to that of the Kafka Producer. It incorporates the Ansible *command* module combined with the *until* module. This combination facilitates the retrieval of container logs intermittently over a 30-second period, with a 3-second interval, to verify the occurrence of the expected log message.

3.2.7 Database

In accordance with the architecture described earlier and depicted in Figure 1, a switch for the database technology has been implemented. The choice between MySQL and PostgreSQL can be determined via the Ansible variable *database_technology*. Consequently, two technology-specific unit tests are necessary to validate the playbook's functionality. It is imperative that integration tests are robust enough to seamlessly handle the switch between databases without any additional complexities.

Integration tests are performed during the testing of the Kafka Consumer unit, as it necessitates a database and orchestrates the setup of the selected database beforehand. The unit tests for MySQL and PostgreSQL share a similar structure, as consistency in testing approaches for analogous components is paramount (Staegemann et al. 2022).

Similar to the aforementioned tests, both MySQL and PostgreSQL unit tests follow a standardized structure:

1) Pre-validations are executed
2) Prerequisites are established
3) The playbook responsible for setting up the database is imported
4) Post-validation procedures are undertaken
5) Artifacts generated during the testing process are cleaned up

To comprehensively test the database setup and functionality, the post-validation phase involves initiating a database client container. Subsequently, various *SQL* commands are executed on the database using the *docker_container* Ansible module and its *command* option. These commands include creating a table and inserting data into it. Following this, database logs are retrieved and analyzed to verify that the executed commands are appropriately reflected. This validation is conducted using the *assert* Ansible module to ensure the integrity of the database operations. Once the validation of database functionality is complete, the database instance is removed to simulate cleanup operations. Subsequently, the database is restarted to confirm the persistence of data through volume management. Finally, the persistence of data is validated by utilizing the database client and executing a *SELECT* statement to verify that the data remains intact within the newly created database container.

3.3 The Application

The testing of the application adheres to the same principles as outlined for infrastructure setup testing. However, unlike the infrastructure components implemented with Ansible, the application is developed in Python. It comprises three major components: *Kafka Producer logic*, *Kafka Consumer logic*, and a *web application*, whereby the latter, as previously mentioned, will not be further described.

Although each of these components undergoes integration and unit testing from the perspective of the infrastructure setup, as previously detailed, additional unit tests are required to validate the

internal implementation of the Python code. To facilitate this, the *unittest* Python module is employed.

Each Python script or project is accompanied by a *tests.py* file, which contains multiple unit tests validating the internal functionality of the project. These tests cover various aspects of the script's functionality, ensuring the correctness of its behavior.

From a procedural standpoint, these unit tests could be executed before deploying the logic via Ansible or integrated as a dedicated step within a CI/CD pipeline.

Again, the development process follows the principles of TDD as well, wherein tests are written prior to the implementation of logic, ensuring a robust and testable codebase.

3.3.1 Kafka Producer

As mentioned earlier, in this implementation, to simulate real data creation, the Kafka Producer ingests data from a CSV file, wherein each line of the file is transmitted as a message to the Kafka cluster.

In real-world scenarios, assessing the functionality of the Python function responsible for data production proves to be challenging due to its indefinite runtime. Consequently, unit testing of this function becomes challenging. As a result, validating the implementation relies on integration tests. These integration tests, implemented in Ansible, were described in section 3.2.5.

An aspect of the internal functionality can be tested, however, by ensuring that the function fails when presented with an invalid path to the CSV file. While this file is only used as an input for the simulated data streaming, to adhere to the TDD philosophy, this aspect will still be tested.

Furthermore, the Kafka producer's connectivity to the Kafka cluster is testable. This can be validated by calling the function with erroneous addresses, which shall lead to failure while valid addresses facilitate successful connection establishment, returning an instance. The Kafka connection test is shown in Figure 3.

```
def test_kafka_connect(self):
  with self.assertRaises(SystemExit):
    initialize_kafka_producer("127.0.0.1:80")
  with self.assertRaises(SystemExit):
    initialize_kafka_producer("192.168.56.102")
  self.assertIsNotNone(initialize_kafka_producer("192.168.56.102:29093"))
```

Figure 3: Kafka connection test.

3.3.2 Kafka Consumer

As shown in section 2.6, the Kafka Consumer serves a multi-faceted role within the project, extending beyond mere data consumption. To summarize, it engages in the consumption of *Gun violence data* from the Kafka broker by subscribing to a designated topic. A connection to the database is established, and it proceeds to iterate over the consumed data, determining its responsibility based on the US state mentioned in each message. Each message undergoes pre-processing, encompassing data cleanup, invalid data filtering, and relevancy assessment. The pre-processed data are then stored in a dedicated database table. Additionally, the Kafka Consumer periodically conducts predictions using fbprophet (with seasonality adjusted for NFL Game days) regarding gun violence per assigned state for the ensuing 365 days. By doing this, eleven predictions are created per state and date followed by the actual value. The predictions as well as the actual values are, subsequently, written to another database table, facilitating data comparison. Since the Kafka Consumer constitutes a sophisticated script, it necessitates comprehensive test coverage to validate critical functionalities. Accordingly, prior to implementation, multiple tests are provided.

Even though the script could mostly be tested locally, some of the tests require several dependencies to be deployed. This includes a working database, as well as a working Kafka Broker.

As described earlier, there can be *1 to n* Kafka Consumers. Each Consumer is responsible for handling a subset of all messages. The assignment is based on the US state the message mentions. Therefore, each Consumer instance calculates a subset of US states, based on the total number of Consumers as well as its own id, defined via input argument.

For this, a basic test ensures the script fails with a non-zero exit code, if the input data is invalid. Invalid could mean that the ID is greater than the total number of Consumers, or that an ID is smaller than 1. Additionally, it is crucial to validate that the summed-up number of all states distributed to the Consumers, is equal to the actual total number. This ensures that the logic constantly handles individual numbers of Consumers, without duplicated or missing entries. The corresponding code is shown in Figure 4.

To further validate the consistency, an additional check is performed, which does not count the entries but instead validates that the combined list of all states that were distributed between the Consumers only contains unique values.

```
def test_state_mapping_calculations(self):
  total_states = 0
  num_of_states = 50
  for i in range(1, consumers + 1):
    expected_length = math.ceil(num_of_states / consumers) if
i != consumers else num_of_states - (consumers - 1) *
math.ceil(num_of_states / consumers)
    total_states += expected_length
    self.assertEqual(len(initialize_states_list(i,
consumers)), expected_length)
  self.assertEqual(total_states, num_of_states)
```

Figure 4: Test to verify the number of states after distribution to the consumers.

Moreover, there are specific fields like *state*, *incident_id*, and *date* that are mandatory for further processing. Additionally, there are invalid messages that need to be either cleaned up or ignored. Based on the fact that the raw data is static and well-known, all critical errors within the data are known as well.

The function *parse_kafka_message* is used to parse the input and perform the preprocessing for each consumed message. It's evident that this is a crucial part of the logic. Therefore, several tests are planned and implemented.

The test *test_parsing_messages* validates:

- parsed valid messages contain 29 comma-separated attributes
- parsed valid messages contain non-empty *incident_id*, *state*, and *date* attributes
- parsed invalid messages that contain empty *incident_id*, *state*, or *date* attributes are expected to be filtered out by the function
- special characters, thus all characters other than *alphanumeric signs (letters and numbers)* and *"."* *(period)*, *" " (space)*, + *(plus)*, and *"-" (hyphen)*, are removed from parsed messages

Further tests are provided for the mapping of teams from the NFL data to US states. Here,

- the function should return an error if an invalid state is provided
- the function should return an empty list if a state has no NFL Team with scheduled games
- the function should fail if the input is not a string
- the function should return a list if a state has *1 to n* teams with scheduled games
- the function should return a unique list of games even if two teams in the same state are opponents

4 DISCUSSION

During the development of the described BD application, the utilization of TDD has yielded several benefits.

Since BD projects entail unique challenges (Staegemann et al. 2019), including scalability, high availability, handling peaking heavy loads of data, and managing times of reduced data flow, an effective infrastructure must be capable of addressing these challenges. TDD, in turn, compels developers to consider edge cases and possibilities that may otherwise be overlooked, which, in turn, can help to reduce the developed applications' susceptibility to errors that might impact correctness or performance.

A notable example arises from the dynamic handling of Consumers and workload distribution across all instances. An unexpected edge case identified was a potential scenario, where the total number of US states may not be evenly divisible by the total number of Consumers.

Additionally, several situations were encountered where mishandling a single invalid message could disrupt consumption, potentially leading to the need for manual intervention or overlooking missing information. Yet, the use of TDD facilitated the identification and proactive handling of errors as crucial aspects of the development process. This is important because potential malfunctions that are not proactively eliminated during the development phase might, at worst, not even be noticed in the environment of BD applications in productive operation but still negatively impact the obtained results.

Consequently, to minimize the risk of such issues, without adhering to the TDD approach, most likely, several additional iterations of error search and code refactoring would have been necessary during the

implementation of the infrastructure as well as the application logic.

However, the extensive data infrastructure, which is characterized by numerous co-dependencies, has led to many integration tests. Although these prove to be more costly than unit tests, they are essential for validating the interactions between all the interconnected components. Further, ensuring the smooth implementation of unit tests in the presence of interdependencies presented challenges that required prior consideration of testing strategies.

Nevertheless, overall, the proposed TDD approach for BDE has shown its feasibility and utility. Hereby, especially the added flexibility through the option to swap a part of the infrastructure rather effortlessly (in this case the database) with an alternative, while being protected through a net of existing tests, highlighted the approach's potential to increase the flexibility and maintainability of the developed solutions. This, in turn, can prove highly valuable in dynamic business environments as discussed in (Staegemann et al. 2020a).

5 CONCLUSION

The harnessing of BD has been shown to have a positive impact on organizational operations. However, creating the corresponding applications is a demanding task and one of the big challenges is to ensure that the applications have the necessary quality. While quality assurance for BD applications is widely researched, this is still far from a solved problem. In response, one rather recent suggestion was the utilization of the TDD approach in the BD domain.

Thus, to further explore the application of TDD in BDE, in this research, a corresponding implementation was created. As a use case for this, the prediction of incidents of gun violence in the USA was chosen as this is a typical BD use case. Further, even though the data were available in batch, stream-processing was simulated, since this is a rather typical requirement in BD scenarios.

Within the implementation, there are two major parts, the infrastructure and the application logic, which were both created in a test driven manner.

For the prediction itself, historical gun violence data were used, which were amended by NFL match day data that were considered a potentially impactful influencing factor. In doing so, the combination of data from different sources, as another typical BD requirement, was reflected.

Through the project, it was not only shown that it is possible to apply the test driven approach for the regarded use case but also that it benefitted the development. This happened on one hand through positive impacts on the developed application's design and on the other hand through the associated quality assurance that helped to avoid errors that would have necessitated cumbersome and time-consuming additional code reviews to find them or might, in a worst case scenario, even have been undetected at all without the use of TDD.

Moreover, as a particularly noteworthy feature, the increased flexibility and maintainability of the developed application stand out because the availability of the corresponding tests facilitates the swapping of components with alternatives with comparatively low effort while being protected through the previously implemented tests. Thereby, the proposed test driven approach seems especially valuable in dynamic business environments, where the applications continuously need to evolve to adapt to changing environments, circumstances, and business needs.

However, it has to be noted that the focus was decisively on the technical realization of the approach itself, therefore, the quality of the results of the prediction algorithm and the frontend application used for visualization were not part of the main priority.

As part of similar future research endeavors, beyond the here employed unit- and integration tests, for instance, End-to-End testing could be conducted to validate the system's entire functionality from end to end. Performance and security tests on the other hand could help to identify other issues. Further, exploring the implementation of hardening measures for strengthening the system's security, such as TLS-encrypted communications, might also be worthwhile, since it could potentially significantly influence the required test framework and tests themselves.

REFERENCES

Agha, D., Sohail, R., Meghji, A. F., Qaboolio, R., and Bhatti, S. (2023). "Test Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review," *VAWKUM Transactions on Computer Sciences* (11:1), pp. 268-280 (doi: 10.21015/vtcs.v11i1.1494).

Apache Software Foundation. (2024a). "Apache Kafka," available at https://kafka.apache.org, accessed on Apr 11 2024.

Apache Software Foundation. (2024b). "Welcome to Apache ZooKeeper," available at https://zookeeper.apache.org/, accessed on Apr 11 2024.

Chang, W. L., and Grady, N. (2019). "NIST Big Data Interoperability Framework: Volume 1, Definitions," *Special Publication (NIST SP),* Gaithersburg, MD: National Institute of Standards and Technology.

Diebold, F. X. (2021). "What's the big idea? "Big Data" and its origins," *Significance* (18:1), pp. 36-37 (doi: 10.1111/1740-9713.01490).

Docker Inc. (2024). "docker," available at https://www.docker.com/, accessed on Apr 11 2024.

Facebook Open Source. (2024). "Prophet: Forecasting at scale.," available at https://facebook.github.io/prophet/, accessed on Apr 12 2024.

Höchtl, J., Parycek, P., and Schöllhammer, R. (2016). "Big data in the policy cycle: Policy decision making in the digital era," *Journal of Organizational Computing and Electronic Commerce* (26:1-2), pp. 147-169 (doi: 10.1080/10919392.2015.1125187).

Horowitz, M. (2018). "Detailed NFL Play-by-Play Data 2009-2018: nflscrapR generated NFL dataset wiith expected points and win probability," available at https://www.kaggle.com/datasets/maxhorowitz/nflplaybyplay2009to2016/data, accessed on Apr 12 2024.

Ji, S., Li, Q., Cao, W., Zhang, P., and Muccini, H. (2020). "Quality Assurance Technologies of Big Data Applications: A Systematic Literature Review," *Applied Sciences* (10:22), p. 8052 (doi: 10.3390/app10228052).

Ko, J. (2018). "Gun Violence Data: Comprehensive record of over 260k US gun violence incidents from 2013-2018," available at https://www.kaggle.com/datasets/jameslko/gun-violence-data, accessed on Apr 12 2024.

Laney, D. (2001). "3D data management: Controlling data volume, velocity and variety," *META group research note* (6:70).

Müller, O., Fay, M., and Vom Brocke, J. (2018). "The Effect of Big Data and Analytics on Firm Performance: An Econometric Analysis Considering Industry Characteristics," *Journal of Management Information Systems* (35:2), pp. 488-509 (doi: 10.1080/07421222.2018.1451955).

Red Hat. (2024). "Ansible," available at https://www.ansible.com/, accessed on Apr 11 2024.

Ronacher, A. (2024). "Flask," available at https://flask.palletsprojects.com/en/3.0.x/, accessed on Jan 16 2024.

Roy, D., Srivastava, R., Jat, M., and Karaca, M. S. (2022). "A Complete Overview of Analytics Techniques: Descriptive, Predictive, and Prescriptive," in *Decision Intelligence Analytics and the Implementation of Strategic Business Management*, P. M. Jeyanthi, T. Choudhury, D. Hack-Polay, T. P. Singh and S. Abujar (eds.), Cham: Springer International Publishing, pp. 15-30 (doi: 10.1007/978-3-030-82763-2_2).

Sonnenberg, C., and Vom Brocke, J. (2012). "Evaluations in the Science of the Artificial – Reconsidering the Build-Evaluate Pattern in Design Science Research," in *Design Science Research in Information Systems. Advances in Theory and Practice*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, K. Peffers, M. Rothenberger and B. Kuechler (eds.), Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 381-397 (doi: 10.1007/978-3-642-29863-9_28).

Staegemann, D., Chadayan, A. K., Mathew, P., Sudhakaran, S. N., Thalakkotoor, S. J., and Turowski, K. (2023). "Showing the Use of Test-Driven Development in Big Data Engineering on the Example of a Stock Market Prediction Application," in *Proceedings of Eighth International Congress on Information and Communication Technology*, X.-S. Yang, R. S. Sherratt, N. Dey and A. Joshi (eds.), Singapore: Springer Nature Singapore, pp. 867-877 (doi: 10.1007/978-981-99-3243-6_70).

Staegemann, D., Volk, M., Daase, C., and Turowski, K. (2020a). "Discussing Relations Between Dynamic Business Environments and Big Data Analytics," *Complex Systems Informatics and Modeling Quarterly* (23), pp. 58-82 (doi: 10.7250/csimq.2020-23.05).

Staegemann, D., Volk, M., Jamous, N., and Turowski, K. (2020b). "Exploring the Applicability of Test Driven Development in the Big Data Domain," in *Proceedings of the 31st Australasian Conference on Information Systems (ACIS),* Wellington, New Zealand. 01.12.2020 - 04.12.2020.

Staegemann, D., Volk, M., Nahhas, A., Abdallah, M., and Turowski, K. (2019). "Exploring the Specificities and Challenges of Testing Big Data Systems," in *Proceedings of the 15th International Conference on Signal Image Technology & Internet based Systems,* Sorrento, Italy. 26.11.2019 - 29.11.2019.

Staegemann, D., Volk, M., Pohl, M., Haertel, C., Hintsch, J., and Turowski, K. (2022). "Identifying Guidelines for Test-Driven Development in Software Engineering—A Literature Review," in *Proceedings of Seventh International Congress on Information and Communication Technology*, X.-S. Yang, S. Sherratt, N. Dey and A. Joshi (eds.), Singapore: Springer Nature Singapore, pp. 327-336 (doi: 10.1007/978-981-19-2397-5_30).

Tarreau, W. (2023). "HAProxy," available at https://www.haproxy.org/, accessed on Apr 11 2024.

Volk, M., Staegemann, D., Trifonova, I., Bosse, S., and Turowski, K. (2020). "Identifying Similarities of Big Data Projects–A Use Case Driven Approach," *IEEE Access* (8), pp. 186599-186619 (doi: 10.1109/ACCESS.2020.3028127).

Volk, M., Staegemann, D., and Turowski, K. (2022). "Providing Clarity on Big Data: Discussing Its Definition and the Most Relevant Data Characteristics," in *Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management,* Valletta, Malta. 24.10.2022 - 26.10.2022, SCITEPRESS - Science and Technology Publications, pp. 141-148 (doi: 10.5220/0011537500003335).