**Intuit Project**

1. **Overview**
2. **Installation and Execution Instructions**
   1. **Local installation**
   2. **Installation with Docker**
3. **Additional requirements - Error and Exception handling, Testing and edge cases,** **Performance enhancements (caching), REST standards**
4. **Future work**
   1. **CI/CD**
   2. **Big scale design**

**1. Overview**

This microservice exposes a REST API for accessing player data from a CSV file. It provides two primary endpoints to fetch all players or retrieve a specific player by their unique playerID.

**Endpoints**

* **GET /api/players** - Returns the list of all players, with optional pagination parameters (skip and limit).
* **GET /api/players/{playerID}** - Returns a single player by ID.

**implementation Details**

**Code Structure**

* **FastAPI Application**: FastAPI was chosen for its ease of use, speed, and asynchronous capabilities.
* **PySpark Integration**: PySpark reads and processes player data from the CSV file.
* **Caching**: @lru\_cache is used to cache player data to optimize repeated access.
* **Logging**: Detailed logging helps to monitor the application's status and debug issues.

**Key Files**

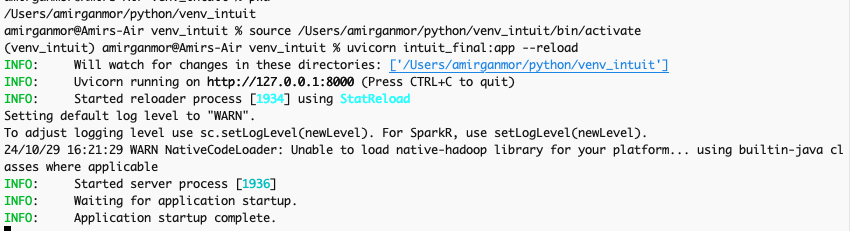
1. intuit\_final.py: Main application file with FastAPI endpoints and helper functions.
2. requirements.txt: Dependencies list for Docker and environment setup.
3. Dockerfile: Defines the environment to run the application in a container.

**Application Components and Requirements**

1. **GET /api/players Endpoint**:
   * **Retrieves all players with optional pagination (using skip and limit parameters).**
   * **Utilizes caching to store the list of players, improving performance for repeated queries.**
   * **Handles cases where the pagination parameters might lead to an empty response.**
2. **GET /api/players/{playerID} Endpoint**:
   * **Retrieves a specific player based on their playerID.**
   * **Returns a 404 error if the player is not found.**
3. **Data Loading**:
   * **The player data is loaded from a CSV file into a Spark DataFrame at startup. This approach leverages PySpark's performance for large datasets, enabling scalable operations on data.**
   * **File Path: In the example code, the CSV path is set to /Users/amirganmor/python/venv\_intuit/player.csv. If deploying in Docker or other environments, ensure this path is accessible or configurable.**
4. **Error and Exception Handling**:
   * **Error handling is implemented at different stages:**
     + **Data Loading: If loading fails, an error is logged and an exception is raised to halt the service.**
     + **Endpoint-Level Errors: Custom exceptions with detailed messages (404 for player not found, 500 for server errors) ensure that users receive informative feedback.**
5. **Testing and Edge Cases**:
   * **The pytest file tests several cases:**
     + **Fetching all players and verifying the return type.**
     + **Fetching a valid player by ID and verifying the returned player data.**
     + **Attempting to fetch an invalid player ID and verifying the 404 responses.**
   * **Edge cases considered include empty results due to pagination parameters and invalid player IDs.**
6. **Dockerization**:
   * **The Dockerfile installs Python, Java (for PySpark), and dependencies, setting up the application to run in a container.**
   * **Exposes port 8000 for API access.**
   * **The CMD command launches the FastAPI app using Uvicorn, making it accessible at 0.0.0.0:8000.**

**Installation and Execution Instructions**

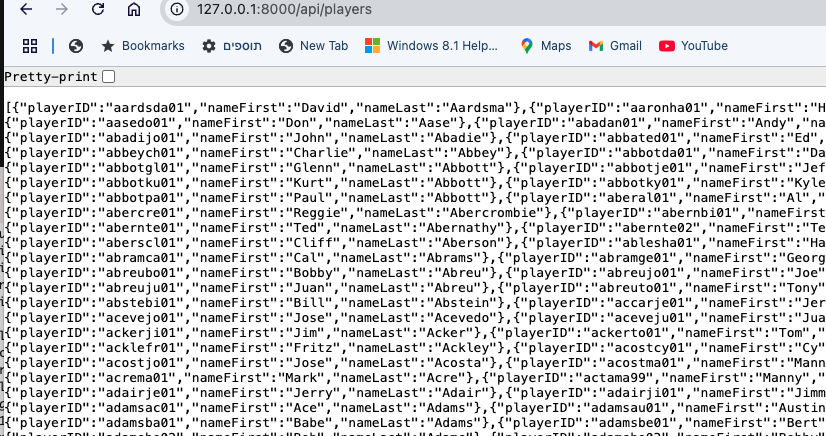
1. **Local Setup**:
   * **Ensure Python 3.9, Java, and PySpark are installed.**
   * **Install FastAPI and other dependencies from requirements.txt.**
   * **Go to local repo**
   * **uvicorn intuit\_final:app --reload**



On local server:

<http://127.0.0.1:8000/api/players>

If hit the pretty-print checkbox it will make it Json prettify





Pagination through the url:

<http://127.0.0.1:8000/api/players?skip=10&limit=5>

**skip**: This parameter defines the starting point in the all-players list from where the records will be returned. For example, if skip=10, the returned list will start from the 11th player (because lists are 0-indexed). **limit**: This parameter specifies the maximum number of records to return. For example, if limit=5, only 5 players will be returned from the skip point

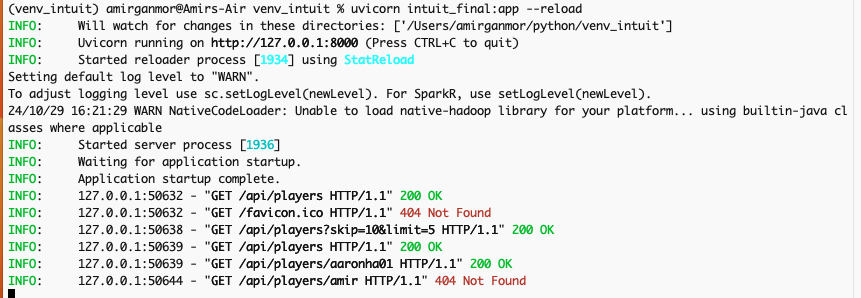
[http://127.0.0.1:8000/api/players/{playerID}](http://127.0.0.1:8000/api/players/%7bplayerID%7d)



If no such player:

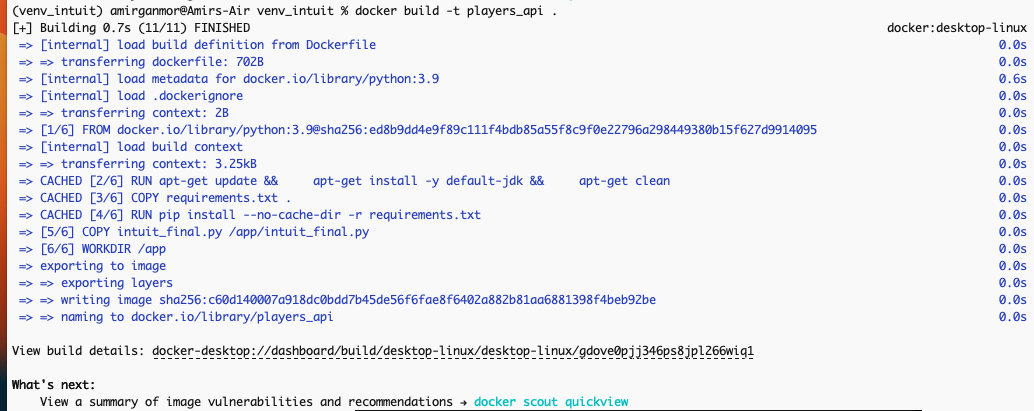


Logs in the terminal:



1. **Dockerized Setup**:

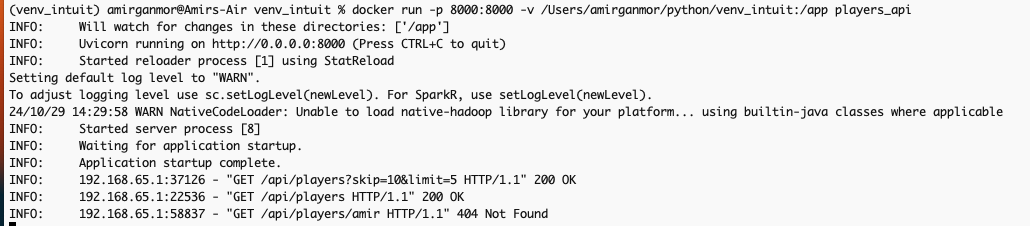
* Goto local repo
* Build the Docker image:
* docker build -t players\_api .



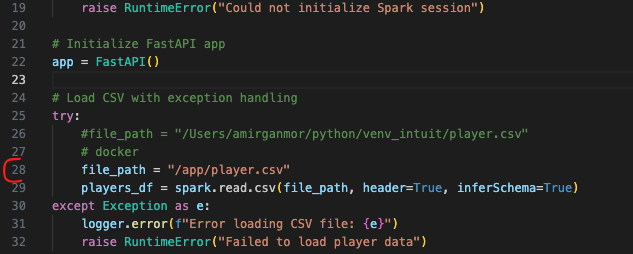
For me it didn’t took a long time since it was no first build…

* Run the Docker container:
* docker run -p 8000:8000 -v /Users/amirganmor/python/venv\_intuit:/app players\_api

logs and, rest call, and Ui in localhost is same as in local run



 Pay Attention! U may want to use different csv or another file and path will needed to change



**Additional requirements**

As per the assignment instructions, the following were also taken into account:

* Error and Exception handling
* Testing and edge cases
* Pagination
* Performance enhancements (caching)
* REST standards

**Handling Edge Cases**

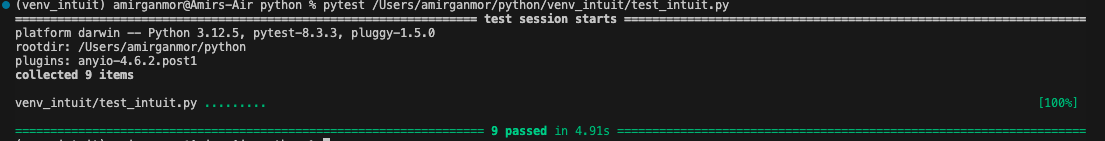
* **Empty Data**: If the CSV is empty or data is missing, return a clear message.
* **ID Not Found**: Already handled with a 404 response.
* **Invalid Request Format**: Use try-except for input validation if necessary.

**Testing and Edge Cases**

Unitests in teat\_intuit.py file

**Descriptions for Each Test**

1. **test\_get\_all\_players**: Verifies that the /api/players endpoint returns a list of all players with a 200-status code.
2. **test\_get\_single\_player\_valid\_id**: Checks the endpoint with a known valid playerID, expecting a 200 response and correct player data.
3. **test\_get\_single\_player\_invalid\_id**: Tests the response for an invalid playerID, expecting a 404 error.
4. **test\_get\_all\_players\_with\_limit**: Validates pagination by limiting the result to 5 players and confirming the count.
5. **test\_get\_all\_players\_with\_skip\_and\_limit**: Verifies combined skip and limit pagination functionality, returning a specific subset.
6. **test\_get\_all\_players\_with\_invalid\_skip**: Ensures that a negative skip parameter results in a 404 (Unprocessable Entity) error.
7. **test\_get\_all\_players\_with\_non\_integer\_pagination**: Checks that non-integer values in skip and limit parameters return a 422 error.
8. **test\_large\_dataset\_fetch\_all\_players**: Tests performance and response consistency by fetching a large set (up to 1000 entries).
9. **test\_api\_health\_check**: Checks that the /api/players endpoint is accessible and returns JSON data, indicating the API’s basic availability.



**Optimizations**

* Pagination

Pagination through the url:

<http://127.0.0.1:8000/api/players?skip=10&limit=5>

**skip**: This parameter defines the starting point in the all-players list from where the records will be returned. For example, if skip=10, the returned list will start from the 11th player (because lists are 0-indexed). **limit**: This parameter specifies the maximum number of records to return. For example, if limit=5, only 5 players will be returned from the skip point

* Cache

I used this:

For simple and fast caching within the same application instance, use functools.lru\_cache. This approach works well if your dataset fits into memory and doesn’t change often.

tro• tunctools I.ort tru_cache 
rou can adjust maxsize based on the expected dataset s" 
def 
players _ list 
"nameFIrst", 
return Irow.asoict() for rcne in players _ Listl 
* update your endpoint to the cached function 
up. get ( • • 
response_modelzList [dict] ) 
async def 
return 

This caches results in memory and refreshes them only if the underlying function parameters change.

**REST Standards**

To adhere to REST standards:

* Use proper HTTP status codes.
* Ensure endpoints are plural (/api/players).
* Use 404 for missing resources and 500 for internal server errors.

**Future work:**

**CI/CD Process Overview**

1. **Source Code Management (SCM)**:
   * Use a version control system (e.g., Git) to manage the codebase. All changes to the code should be committed to a shared repository (e.g., GitHub, GitLab, Bitbucket).
2. **Continuous Integration (CI)**:
   * **Code Review:** Implement a code review process using pull requests (PRs) before merging changes to the main branch. Reviewers should check for code quality, adherence to standards, and potential issues.
   * **Automated Testing:**
     + **Unit Tests:** Write unit tests for FastAPI endpoints, data processing logic, and any other components. Use frameworks like pytest for Python testing.
     + **Integration Tests:** Create integration tests to verify interactions between different components (e.g., FastAPI, Kafka, databases). Ensure that the API responds correctly to various inputs and that the data flows through the system as expected.
     + **Data Quality Tests**: Implement checks to ensure data integrity and quality in the processing pipeline (e.g., schema validation, data completeness).
3. **Continuous Deployment (CD)**:
   * **Build Process:**
     + Use a CI/CD tool (e.g., Jenkins, GitHub Actions, GitLab CI, CircleCI) to automate the build process. The pipeline should build Docker images for the FastAPI application and data processing jobs.
     + Ensure that the build process includes steps for linting and code quality checks (e.g., using flake8 or black for Python).
   * **Containerization:**
     + Create Docker containers for all components of your application, including FastAPI, Spark/Flink jobs, and any other microservices.
   * **Deployment Stages:**
     + **Staging Environment**: Deploy the application to a staging environment for further testing. This environment should closely mirror production.
     + **Testing in Staging**: Run the automated tests again in the staging environment. Ensure that the application behaves as expected with real data flows.
     + **Production Deployment**: Once all tests pass, automate the deployment to the production environment using the CI/CD tool. Use tools like Kubernetes for orchestrating the deployment of containers.
4. **Monitoring and Feedback**:
   * Set up monitoring tools (e.g., Grafana, ELK Stack) to track the performance and health of the application in production.
   * Implement logging to capture application behavior and any errors that occur during operation.
   * Use alerts to notify the team of any issues or anomalies detected in the production environment.

**CI/CD Pipeline Steps**

Here’s a high-level overview of the CI/CD pipeline steps:

1. **Commit Code**: Developers push code changes to the repository.
2. **Trigger CI Pipeline**: The CI/CD tool triggers the pipeline automatically upon detecting new commits.
3. **Run Unit Tests**: Execute unit tests against the new code.
4. **Run Integration Tests**: Execute integration tests to validate interactions between components.
5. **Build Docker Images**: Build Docker images for the application and any microservices.
6. **Deploy to Staging**: Deploy the new images to a staging environment.
7. **Run End-to-End Tests**: Validate the entire flow in the staging environment, including data ingestion, processing, and API interactions.
8. **Deploy to Production**: Upon successful testing in staging, automatically deploy to the production environment.
9. **Monitor Performance**: Continuously monitor the application and data processing pipeline in production for performance and errors.

flowchart TD

A[Developers Commit Code] --> B[CI/CD Tool]

B --> C[Run Unit Tests]

C --> D[Run Integration Tests]

D --> E[Build Docker Images]

E --> F[Deploy to Staging]

F --> G[Run End-to-End Tests]

G --> H[Deploy to Production]

H --> I[Monitor Performance]

This CI/CD process ensures that your FastAPI application and data processing components are continuously integrated and deployed in a controlled manner. Automated testing and monitoring at each stage of the process help maintain high-quality standards and minimize the risk of errors in production.

the CI/CD process outlined above is designed to trigger automated tests on each commit. Here’s how that works in detail:

**Triggering Automated Tests**

1. **Commit Code**: When a developer pushes changes to the code repository (e.g., Git), this action triggers the CI/CD pipeline.
2. **Continuous Integration (CI) Pipeline**:
   * The CI/CD tool (like Jenkins, GitHub Actions, or GitLab CI) listens for changes in the repository.
   * Upon detecting a new commit, the pipeline starts executing the defined steps.
3. **Run Automated Tests**:
   * **Unit Tests:**
     + The pipeline first executes the unit tests, which validate individual components of the code (e.g., FastAPI endpoints, functions).
     + These tests should cover various scenarios, including edge cases, to ensure that the individual pieces of code work correctly.
   * **Integration Tests:**
     + After the unit tests pass, integration tests are run. These tests check the interactions between multiple components (e.g., the FastAPI application communicating with Kafka or databases).
     + Integration tests help catch issues related to how components work together, ensuring that data flows correctly through the system.
4. **Failing Tests**:
   * If any of the tests (unit or integration) fail, the pipeline stops at that step. The developers are notified (via email, chat, etc.) about the failure, allowing them to address the issues before any further steps are taken.
5. **Successful Tests**:
   * If all tests pass, the pipeline proceeds to build the Docker images and continues through the remaining steps, eventually deploying the application to staging and then to production if all tests in the staging environment are also successful.

**Benefits of Triggering Tests on Each Commit**

* **Immediate Feedback**: Developers receive quick feedback on their code changes, allowing them to catch and fix issues early in the development process.
* **Higher Code Quality**: Regularly running automated tests helps maintain code quality and reduces the likelihood of bugs reaching production.
* **Continuous Improvement**: Automated testing encourages developers to write tests alongside their code, leading to a more robust and reliable codebase over time.

**Summary**

In summary, the CI/CD process ensures that automated tests are triggered with every commit. This is a crucial aspect of modern software development, fostering a culture of continuous integration and delivery while maintaining high standards of code quality and reliability.

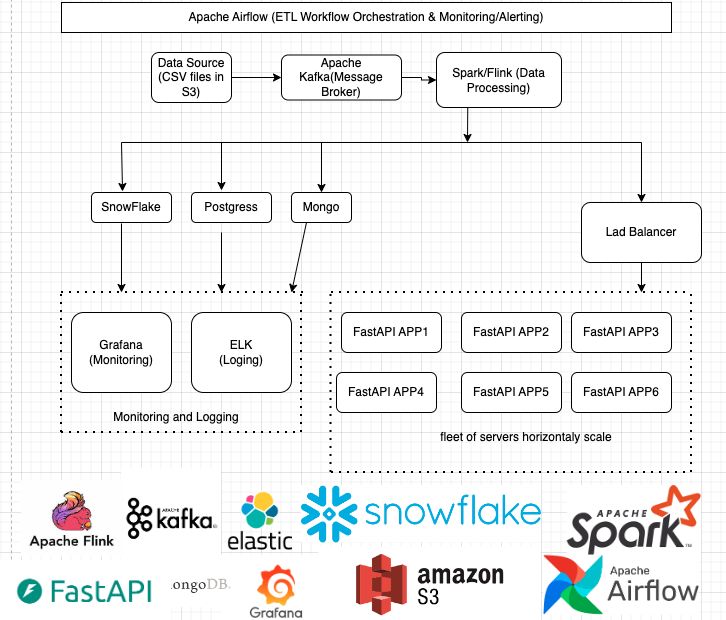
Big Scale Design:

**System Design Overview**

1. **Data Ingestion**:
   * **S3 as Data Lake:** All CSV files are uploaded to S3, allowing for easy storage and retrieval of large datasets. S3 scales seamlessly to accommodate growing data volumes.
2. **Message Broker**:
   * **Apache Kafka:** Acts as the intermediary for streaming data. Kafka can handle a high throughput of messages, enabling the system to efficiently process large volumes of incoming data without bottlenecks.
3. **Data Processing**:
   * **Spark/Flink**: These distributed processing frameworks allow for scalable data transformations. They can read from Kafka and simultaneously write to multiple data sinks (Snowflake, PostgreSQL, MongoDB), which enhances performance and reduces latency.
4. **Data Storage**:
   * **Snowflake:** A cloud-based data warehouse that can automatically scale to accommodate fluctuating workloads.
   * **PostgreSQL:** Provides structured SQL storage; can be scaled vertically (by increasing resources) and horizontally (by partitioning data across multiple servers).
   * **MongoDB**: NoSQL database designed for high scalability and flexibility, capable of handling large volumes of unstructured data.
5. **API Layer**:
   * **FastAPI:** Designed for high performance, FastAPI can handle many simultaneous user requests. Connection pooling and caching can be implemented to further enhance responsiveness.
   * **Load Balancer:** Distributes incoming API requests across multiple FastAPI instances, ensuring optimal resource utilization and improved fault tolerance.
6. **Monitoring and Logging**:
   * **Grafana**: Provides real-time monitoring and visualization of the system’s performance metrics, allowing for proactive scaling and troubleshooting.
   * **ELK Stack:** Used for logging, providing insights into application behavior and facilitating debugging.

**Example Flow**

1. **User uploads CSV file to S3**.
2. **S3 triggers an event** that sends a message to Kafka.
3. **Kafka receives the message** and triggers a Spark/Flink job.
4. **Spark/Flink processes the data** and writes to:
   * **Snowflake for data warehousing.**
   * **PostgreSQL for structured relational queries.**
   * **MongoDB for flexible, document-based storage.**
5. **FastAPI** handles user requests and can query data from all three data sources.
6. **Grafana and ELK** provide monitoring and logging capabilities.



**Scalability Considerations**

* **Horizontal Scaling**: Components like Kafka, Spark/Flink, PostgreSQL, and MongoDB can be scaled out by adding more instances or partitions to handle increased loads.
* **Elasticity**: Snowflake's architecture inherently supports automatic scaling based on the workload, optimizing performance during peak times.
* **Concurrent Processing**: The ability to process and write data to multiple databases in parallel ensures that the system can handle multiple data sources and sinks efficiently.
* **High Availability**: The load balancer and multiple FastAPI instances ensure that the API layer remains available even under heavy traffic.

Sub sections:

* **FastAPI and Load Balancing**

1. **Multiple Instances**:
   * **Deploy multiple instances of your FastAPI application. Each instance can be a separate container (e.g., using Docker) or a virtual machine (VM).**
   * **This approach allows for concurrent handling of multiple requests, significantly improving throughput.**
2. **Load Balancer Positioning**:
   * **Place a Load Balancer (such as AWS Elastic Load Balancer, NGINX, or HAProxy) in front of the FastAPI instances.**
   * **The load balancer will route incoming API requests to one of the available FastAPI instances based on the load and health of each instance.**
3. **How It Works**:
   * **When a request comes in, the load balancer determines which FastAPI instance to route it to, ensuring that no single instance becomes a bottleneck.**
   * **The load balancer can also perform health checks to ensure that only healthy instances receive traffic.**

**Scalability Strategies for FastAPI**

* **Horizontal Scaling**: As traffic increases, you can add more FastAPI instances behind the load balancer. This can be done manually or automatically (using autoscaling groups in cloud environments).
* **Asynchronous Handling**: FastAPI is designed to handle asynchronous requests efficiently. Using asynchronous endpoints (async def), FastAPI can serve more requests concurrently without blocking on I/O operations.
* **Connection Pooling**: If your FastAPI app connects to a database (e.g., PostgreSQL, MongoDB), implement connection pooling. This allows multiple requests to reuse database connections instead of opening new ones, reducing latency and resource consumption.
* **Caching**: Implement caching mechanisms (like Redis) for frequently requested data. This will reduce the load on the FastAPI instances and speed up response times.
* **Writing simultaneously DB, and to the FASTAPI**

it is possible to write simultaneously to multiple data sources (PostgreSQL, MongoDB, and Snowflake) from Spark or Flink, and also to communicate with the FastAPI. This design can help reduce latency in the REST API by ensuring that the data is readily available in the required formats and locations for consumption. Here’s how this can be accomplished:

**Writing Simultaneously to Multiple Data Sources**

1. **Data Processing Framework (Spark or Flink)**:
   * Both Spark and Flink support writing data to multiple sinks (data destinations) in parallel. You can implement a pipeline that reads from your source (e.g., Kafka or S3) and processes the data before writing to different databases.
2. **Sinks Configuration**:
   * You can configure multiple sinks in your Spark or Flink application to write the processed data to PostgreSQL, MongoDB, and Snowflake simultaneously.
   * Each sink will handle the specifics of connecting to and writing data to its respective database.
3. **Batch or Streaming Mode**:
   * Depending on your use case, you can choose to write data in batch mode (e.g., periodically) or streaming mode (e.g., in real time as data arrives).
   * For streaming applications, make sure your data flow is designed to handle backpressure and manage resource consumption appropriately.

**Example of Simultaneous Writes in Spark**

Here’s a simplified example of how you might implement simultaneous writes to PostgreSQL, MongoDB, and Snowflake using Spark:

 from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder \

.appName("MultiSinkPipeline") \

.getOrCreate()

# Read data from Kafka or another source

df = spark.read.format("kafka").option("kafka.bootstrap.servers", "localhost:9092").load()

# Process the data as needed

processed\_df = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Write to PostgreSQL

processed\_df.write \

.format("jdbc") \

.option("url", "jdbc:postgresql://your\_postgres\_host:5432/your\_database") \

.option("dbtable", "your\_table") \

.option("user", "your\_username") \

.option("password", "your\_password") \

.mode("append") \

.save()

# Write to MongoDB

processed\_df.write \

.format("mongo") \

.option("uri", "mongodb://your\_mongo\_host:27017/your\_database.your\_collection") \

.mode("append") \

.save()

# Write to Snowflake

processed\_df.write \

.format("net.snowflake.spark.snowflake") \

.option("url", "your\_snowflake\_account.snowflakecomputing.com") \

.option("user", "your\_username") \

.option("password", "your\_password") \

.option("dbtable", "your\_table") \

.option("warehouse", "your\_warehouse") \

.option("db", "your\_database") \

.option("schema", "your\_schema") \

.mode("append") \

.save()

**Communicating with FastAPI**

1. **Data Availability**:
   * **After writing to the databases, ensure that the FastAPI application queries these sources as needed to serve requests.**
   * **Depending on your application architecture, you might want to trigger a refresh or cache update in FastAPI after a write operation to reduce latency.**
2. **Direct Data Access**:
   * **FastAPI can be configured to directly access data from the databases (PostgreSQL and MongoDB) as needed, which allows it to serve the latest data without delays.**
   * **For Snowflake, you can use its native connectors to retrieve data efficiently.**
3. **Event-Driven Architecture**:
   * **If you want to reduce latency further, consider implementing an event-driven architecture where FastAPI listens for events (e.g., via Kafka) signaling that new data is available. This way, FastAPI can update its state or cache accordingly.**

**Key Considerations**

1. **Concurrent Writes**:
   * **Make sure your Spark/Flink application is configured to write to all databases simultaneously, leveraging the capabilities of distributed computing.**
2. **FastAPI Performance**:
   * **FastAPI should be optimized for performance with connection pooling for each database to minimize latency when serving requests.**
   * **Consider implementing caching strategies (e.g., Redis) to further reduce response times for frequently requested data.**
3. **Event-Driven Updates**:
   * **Implement event-driven mechanisms (using Kafka) for FastAPI to listen to data availability or changes, so it can update its state or cache as needed.**
4. **Monitoring**:
   * **Use Grafana to visualize the performance and health metrics of each component in the system.**
   * **Use ELK for logging error messages, processing times, and any other relevant information for better observability.**

**Conclusion**

This architecture is designed to be robust, flexible, and capable of scaling to meet the demands of growing data volumes and user requests. It allows for simultaneous processing and updating of multiple data resources, ensuring low latency and high performance for users interacting with the FastAPI application.

**Further work with ML Based on the data aggregated in the DBs:**

**1. Performance Prediction**

* **Player Performance Models**: Create models to predict player performance based on historical data (e.g., past games, injuries, etc.). This could involve regression analysis or time-series forecasting.
* **Recommendation System**: Implement a recommendation engine to suggest players for teams or trades based on performance metrics and historical data.

**2. Player Comparison Tool**

* Develop a feature that allows users to compare players based on various statistics. This could use clustering algorithms to group players with similar attributes or performance metrics.

**3. Sentiment Analysis**

* Integrate sentiment analysis on player-related news, social media, or fan interactions to gauge public opinion about players. This can be useful for team management decisions or marketing strategies.

**4. Injury Prediction**

* Use machine learning models to analyze data from player training and games to predict injury risks. This could involve analyzing biomechanical data or training load metrics.

**5. Data Enrichment**

* Integrate external data sources (e.g., social media sentiment, weather conditions during games) to enrich player data. Machine learning can be used to correlate this data with player performance.

**6. Automated Insights**

* Create automated insights and reports that summarize player performances, trends, and predictions based on the accumulated data. This could involve natural language generation (NLG) techniques.

**7. Training Recommendations**

* Develop a system to provide personalized training recommendations for players based on their performance data, helping them improve in specific areas.

**8. Anomaly Detection**

* Implement anomaly detection to identify unusual patterns in player performance or training data. This can help in early identification of potential issues or trends.

**9. Real-Time Analytics**

* Use streaming analytics to analyze player performance data in real-time during games. This could involve setting up a stream processing framework (e.g., Apache Kafka with Apache Flink) to provide live insights.

**10. Visualizations and Dashboards**

* Create interactive dashboards that visualize player statistics, predictions, and analyses. Tools like Tableau or Power BI can help present the data effectively.

**11. Chatbot for Player Queries**

* Develop a chatbot that uses NLP (Natural Language Processing) to answer questions about player statistics, performances, and comparisons, enhancing user interaction.

**Implementation Considerations**

* **Model Training**: Decide how to train your ML models. You may need historical data and could use tools like TensorFlow or PyTorch for model development.
* **Data Pipeline**: Implement a data pipeline for collecting, processing, and storing data for ML analysis. This could involve using tools like Apache Airflow for orchestrating ETL tasks.
* **API Integration**: Ensure that your REST API exposes endpoints for AI/ML functionalities, such as retrieving predictions or analysis results.
* **Performance Monitoring**: Track the performance of your AI/ML models in production, using monitoring tools to detect any issues or degrade in accuracy over time.

Integrating AI/ML capabilities can significantly enhance your microservice's functionality, making it not only a data-serving application but also a valuable tool for analysis and decision-making.