

ML serving and monitoring with FastAPI and Evidently

In this code tutorial, you will learn how to set up an ML monitoring system for models deployed with FastAPI.

14 min. read · [View original](#)

In this code tutorial, you will learn how to set up an ML monitoring system for models deployed with FastAPI. You will use Evidently open-source Python library to track production model quality and target drift.

This is a complete deployment blueprint for ML serving and monitoring using open-source tools. You can copy the repository and adapt the reference architecture for your use case.

Code example: if you prefer to head straight to the code, open [this example folder](#) on GitHub.

⚠ Disclaimer:

This example uses the Evidently API as available in version 0.6.7 or lower. Please ensure you are using the correct version when running this example. For updated and new examples, visit our [documentation](#).

[fs-toc-omit]Background

FastAPI is a high-performance web framework for building APIs with Python 3.7+ based on standard Python-type hints. It is easy to use, robust, and fast, making it an excellent choice for serving ML models.

Evidently is an open-source Python library to evaluate, test and [monitor ML models](#). It has 100+ built-in metrics and tests on data quality, data drift, and model performance and helps interactively visualize them.

Combining the two tools, you can deploy an ML model and monitor its performance.

[fs-toc-omit]Get started with AI observability

Try our open-source library with over 25 million downloads, or sign up to Evidently Cloud to run no-code checks and bring all the team to a single workspace to collaborate on AI quality.

[Sign up free →](#)

[Or try open source →](#)

Tutorial scope

In this tutorial, you will create a FastAPI application to serve ML model predictions, log them to the PostgreSQL database and create an Evidently monitoring dashboard to keep track of the model performance.

By the end of this tutorial, you will learn how to implement an ML monitoring architecture using:

- *FastAPI* to serve an ML model, generate predictions, and build monitoring reports.
- *Evidently* to perform data quality, data drift, and model quality checks.
- *PostgreSQL* database to store the predictions.
- *Streamlit* as a simple UI to run and view monitoring reports.

You will run the ML monitoring solution in a Docker environment for easy deployment and scalability.

[fs-toc-omit]Pre-requisites

We expect that you:

- Went through the Evidently [Get Started Tutorial](#) and can generate visual and JSON reports with metrics.

You also need the following tools installed on your local machine:

Note: we tested this example on macOS/Linux.



Installation

Let's prepare to run the demo! This section explains the instructions in the example README: [head there](#) for more details.

1. Fork / Clone the repository

Clone the Evidently GitHub repository with the example code. This repository provides the necessary files and scripts to integrate Evidently, FastAPI, PostgreSQL, and Streamlit.

```
git clone git@github.com:evidentlyai/evidently.git
cd evidently/examples/integrations/fastapi_monitoring
```

2. Build the Docker images

Create Docker images for the example. Before building the Docker images, set the **USER_ID** environment variable to your user ID. You can use this environment variable (**USER_ID**) in Dockerfiles or **docker-compose.yml** files to set permissions or perform other user-specific tasks.

```
export USER_ID=$(id -u)
docker compose build
```

3. Launch the monitoring cluster

Launch the monitoring cluster by using Docker Compose.

The **docker-compose.yml** specifies the cluster components:

- *streamlit_app* - Streamlit application, available on **http://localhost:8501**
- *fastapi* - FastAPI application, available on **http://localhost:5000**
- *monitoring-db* - PostgreSQL, available on **http://localhost:5432**

4. Create the database table

Before storing predictions in the PostgreSQL database, create the necessary tables. Run a Python script below to set up the database structure.

```
python src/scripts/create_db.py
```

5. Download data & train model

This example is based on the [NYC Taxi dataset](#). The data and the model training are out of the scope of this tutorial. We prepared a few scripts to download data, pre-process it and train a simple machine learning model.

```
# Download data for NYC Taxi to 'data/raw' & prepare features
python src/pipelines/load_data.py
python src/pipelines/process_data.py
```

```
# Train a model and save it to 'models/'
python src/pipelines/train.py
```

6. Prepare the reference data for monitoring

Some ML monitoring tasks require a reference dataset. For example, you can use data from model validation. You will compare your production data against this reference.

For this demo, we pre-computed the reference dataset. We assume it remains static.

Execute the following command to prepare your reference data:

```
# Save to 'data/reference'
python src/pipelines/prepare_reference_data.py
```

Run ML model serving and monitoring

Now, let’s launch the pre-built example that shows how to serve an ML model with FastAPI and monitor it with Evidently.

1. Simulate inference

At this point, you should already have a running *fastapi_app*.

> docker ps				
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
467b99d624fe	fastapi_app:latest	"/bin/sh -c 'uvicorn..."	2 hours ago	Up 2 hours
93bc169044b7	streamlit_app:latest	"/bin/sh -c 'cd stre..."	2 hours ago	Up 2 hours
e7a263dc96f4	postgres:15.2-alpine	"docker-entrypoint.s..."	2 hours ago	Up 2 hours (healthy)

FastAPI provides interactive API documentation that you can access using a web browser. It shows all available endpoints, their expected parameters and even allows sending test requests directly from the browser.

To view the API methods available for your project, navigate to the following URL: **http://0.0.0.0:5000/docs#/**

You can explore the different API methods, check their functionality, and try them out. You will see the required request format and expected response format for each method.

default

GET	/	Index	⌵
POST	/predict	Predict	⌵
GET	/monitor-model	Monitor Model Performance	⌵
GET	/monitor-target	Monitor Target Drift	⌵

Let's generate some predictions using a prepared script.

```
python src/scripts/simulate.py
```

This script emulates requests sent to the machine learning model. It selects a batch of data entries at random from the **data/features/green_tripdata_2021-02.parquet** file. Then, it uses these entries to make requests to the **/predict** endpoint.

```
...
resp: requests.Response = requests.post(
    url='',
    json={'features': batch.to_json()}
)
...
```

2. Run monitoring reports

We prepared two monitoring reports: a **Model Performance Report** and a **Target Drift Report**. You must invoke specific endpoints in the FastAPI application to generate them.

1. Model Performance Report:

This report provides insights into the model performance over time. You can access it via the following endpoint:
`http://0.0.0.0:5000/monitor-model`

2. Target Drift Report:

This report allows tracking changes in the target variable over time. You can access it via the following endpoint:
`http://0.0.0.0:5000/monitor-target`

☒ You can select other metrics. Evidently has multiple Metrics and Tests to evaluate data quality, data drift, and model performance. Browse through available [presets](#) to choose what might be relevant for your use case.

You can specify the size of the prediction data window used to generate the report. To do this, add a **window_size** parameter to the URL.

For instance, if you want to generate the report based on the last 300 predictions, use the following URLs:

- **`http://0.0.0.0:5000/monitor-model?window_size=300`**
- **`http://0.0.0.0:5000/monitor-target?window_size=300`**

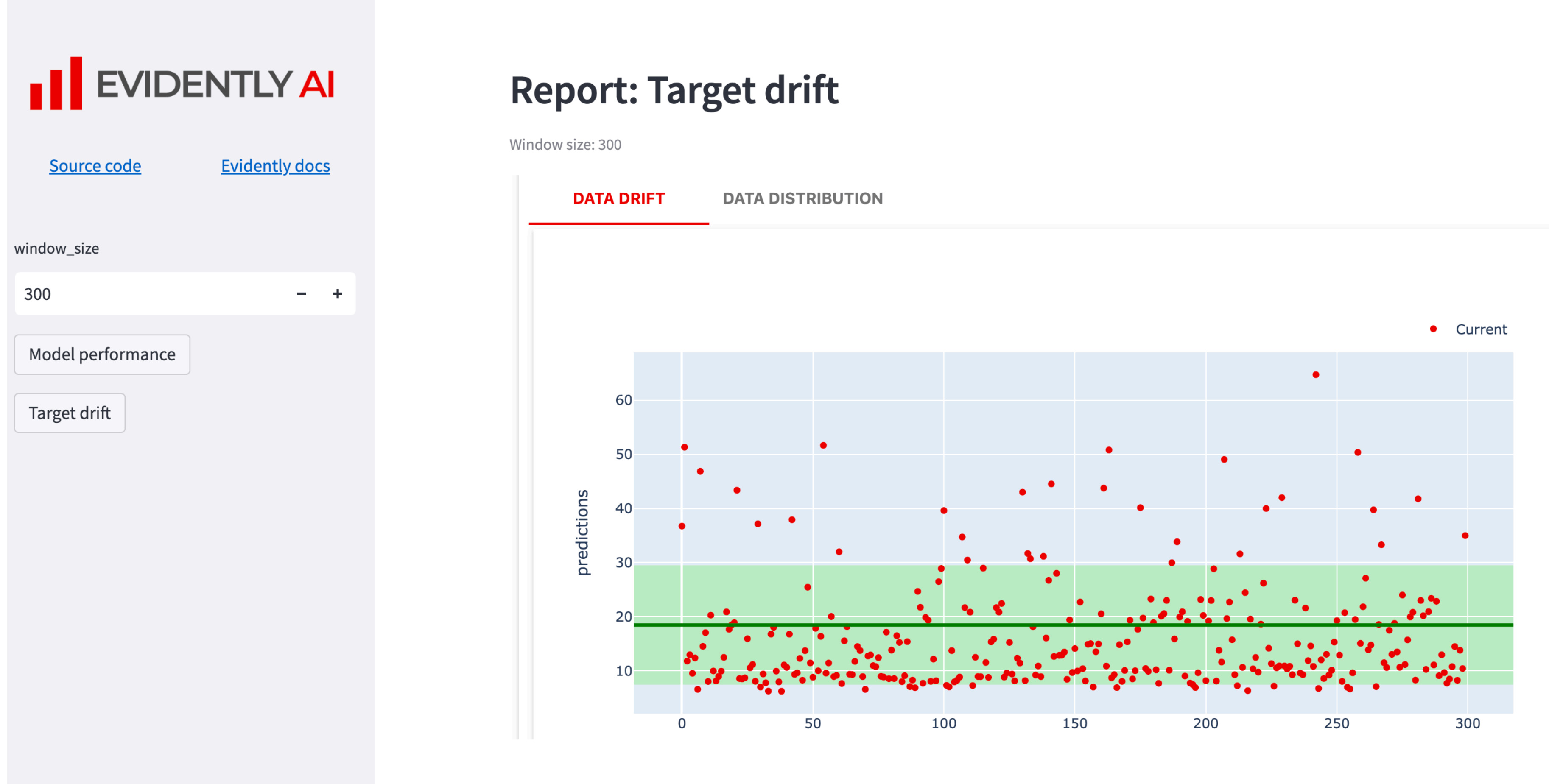
If you do not specify the **window_size**, the system defaults to using the last 3000 predictions.

These reports provide insight into the ML model's performance and the stability of the target variable. You can check them regularly to identify any unexpected behaviors or trends and take appropriate action.

3. View reports in Streamlit UI

This is an optional step.


[Streamlit](#) helps quickly build and deploy shareable web apps. In our case, we can use Streamlit to create a user-friendly interface to run and display our monitoring reports.



To generate the monitoring reports in Streamlit UI, follow these steps:

1. **Access the Streamlit application.** Navigate to the app on <http://localhost:8501/>
2. **Specify the window size.** Set the window size for the monitoring report. The default is 3000, but you can pick a different option.
3. **Select the report type.** Choose between the “Model performance” and “Target drift” reports.

The application will take care of the rest and render the requested report.

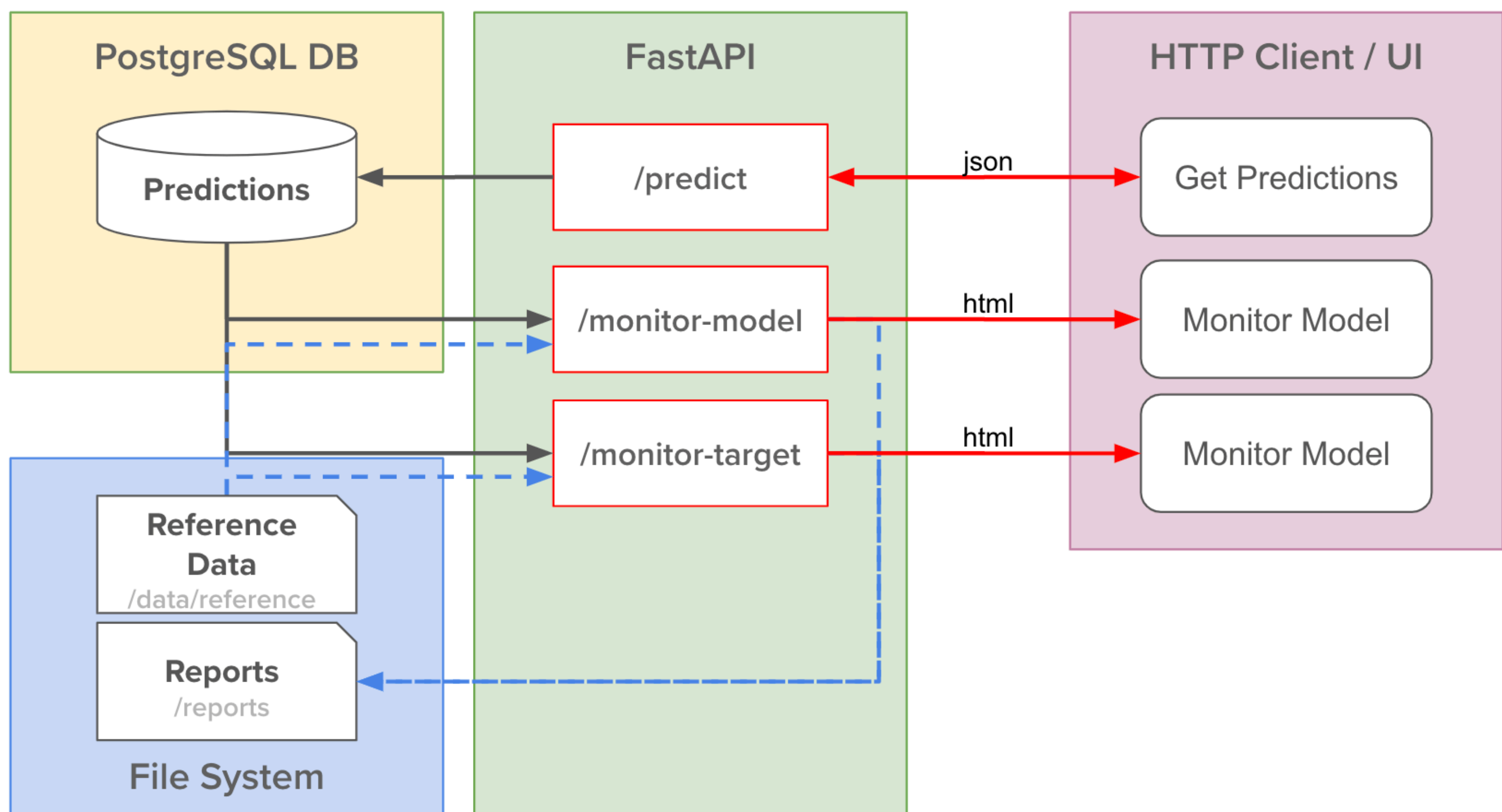
 Want to understand the integration with Streamlit better? Here is a code tutorial on building ML and data monitoring dashboards with [Evidently and Streamlit](#).

Design ML serving and monitoring

Now, let's explore the ML serving and monitoring architecture with FastAPI and Evidently in detail.

1. Solution architecture

The integration has several interconnected components: the FastAPI server, the PostgreSQL database, the user interface (UI) or HTTP client, and the file system.



Let's look at each component.

FastAPI. FastAPI acts as the main server in this architecture. It exposes several endpoints to generate predictions and run model monitoring.

- **/predict:** This endpoint receives a JSON input representing feature data. It processes the data through the ML model and outputs the predictions. The predictions are logged into the PostgreSQL database.
- **/monitor-model:** This endpoint uses Evidently to generate model performance reports based on a sliding window of recent predictions and reference data. It returns the generated reports in HTML and saves them to the file system under the **/reports** directory.

- **/monitor-target:** This endpoint generates target drift reports, comparing the distributions of the target variable in the reference data and the recent predictions. These reports are also returned as HTML and saved to the **/reports** directory.

PostgreSQL. The PostgreSQL database stores the generated predictions. You can later query these predictions using the ML monitoring endpoints to generate the model performance and target drift reports.

HTTP Client / UI: This component is responsible for interacting with the FastAPI server. It sends requests to the **/predict** endpoint to generate predictions and fetches monitoring reports from the **/monitor-model** and **/monitor-target** endpoints. In this demo, we use the Streamlit app to build a simple UI.

File System. Some artifacts are saved into the file system. The reference data is stored in the **/data/reference** directory. This is necessary to compare the model's current performance against its historical performance. Additionally, the generated monitoring reports are in the **/reports** directory.

These components provide a streamlined system to serve ML model predictions, log them, and run and store ML monitoring reports.

We use Docker Compose to manage and orchestrate our services: FastAPI application, Streamlit application, and a PostgreSQL database.

```
version: "3.9"

services:

  fastapi_app:
    image: fastapi_app:latest
    ports:
      - 5000:5000
    networks:
      - monitoring
    ...

  monitoring-db:
    image: postgres:15.2-alpine
    ports:
      - 5432:5432
    networks:
      - monitoring
    ...

  streamlit_app:
    image: streamlit_app:latest
    ports:
      - 8501:8501
    networks:
      - monitoring
    ...

networks:
  monitoring:
    name: monitoring
```

Let's take a look at the **docker-compose.yaml** file:

- **fastapi_app.** This service runs the FastAPI application, which provides the API endpoints for model prediction and performance monitoring. We map port 5000 of the container to port 5000 of the host machine, meaning that the FastAPI server is accessible via **http://localhost:5000**.
- **monitoring-db.** This service runs the PostgreSQL server that stores the prediction results. We map port 5432 of the container (the default PostgreSQL port) to port 5432 of the host machine. This means you can access the PostgreSQL server directly via localhost on port 5432 using any PostgreSQL client.
- **streamlit_app:** This service runs the Streamlit application, which provides the UI for viewing the model monitoring reports. We map port 8501 of the container (the default Streamlit port) to port 8501 of the host machine, making the Streamlit server accessible via **http://localhost:8501**.

All three services are connected to the **monitoring** network, enabling it to access the FastAPI server and the PostgreSQL database. By running these services with Docker Compose, you can ensure that all components work together seamlessly in a controlled and isolated environment.

2. Set up the FastAPI app

In the example FastAPI app, we define three key endpoints. Each serves a specific purpose: generate predictions, monitor model quality, and monitor target drift.

The first endpoint is **/predict**. This endpoint is set up to receive a POST request containing input features. The **predict** function is designed to receive the request, compute and save predictions into a database.

```
@app.post('/predict')
def predict(response: Response, features_item: Features, background_tasks: BackgroundTasks):
    try:
        features: pd.DataFrame = pd.read_json(features_item.features)
```

```

        features['predictions'] = get_predictions(features, MODEL)
        background_tasks.add_task(save_predictions, features)
        return {'predictions': features.to_json()}
except Exception as e:
    response.status_code = 500
    return {'error_msg': str(e)}

```

Here's how it works:

- It accepts three arguments: a **Response** object to modify the HTTP response sent back to the client, a **Features** object that holds the feature data sent with the request, and a **BackgroundTasks** object for managing background tasks.
- The feature data is converted to a *pandas.DataFrame*, and the **get_predictions()** function is invoked to compute predictions using the model.
- The computed predictions are saved into the database as a background task.

The second endpoint **/monitor-model** is a GET endpoint that, when accessed, invokes the **monitor_model_performance()** function. This function is responsible for monitoring the performance of the model.

```

@app.get('/monitor-model')
def monitor_model_performance(window_size: int = 3000):

    current_data: pd.DataFrame = load_current_data(window_size)
    reference_data = load_reference_data(...)

    column_mapping: ColumnMapping = get_column_mapping(...)
    report_path: Text = build_model_performance_report(
        reference_data=reference_data,
        current_data=current_data,
        column_mapping=column_mapping
    )

    return FileResponse(report_path)

```

Here's how it works:

- The function accepts an optional **window_size** parameter with a default value of 3000. You can use this parameter to specify the number of recent predictions to consider when computing performance metrics.
- **load_current_data(window_size)** is a function that reads the current data, i.e., the most recent data available up to the specified **window_size**.
- **load_reference_data()** is a function that reads the reference dataset.
- **build_model_performance_report()** function creates the model performance report. It requires the reference and current data and a **ColumnMapping** object (created by the **get_column_mapping()** function). It helps Evidently correctly process the input data structure.
- Finally, **FileResponse(report_path)** returns the generated report in response to the GET request. The report is sent as an HTML file you can view directly in the browser.

The third endpoint **/monitor-target** is another GET endpoint. When accessed, it invokes the **monitor_target_drift()**function to monitor the drift in the target variable. Similar to the **monitor_model_performance** endpoint, this function accepts an optional **window_size** parameter and returns an HTML file.

```

@app.get('/monitor-target')
def monitor_target_drift(window_size: int = 3000) -> FileResponse:

    current_data: pd.DataFrame = load_current_data(window_size)
    reference_data = load_reference_data(...)
    column_mapping: ColumnMapping = get_column_mapping(...)

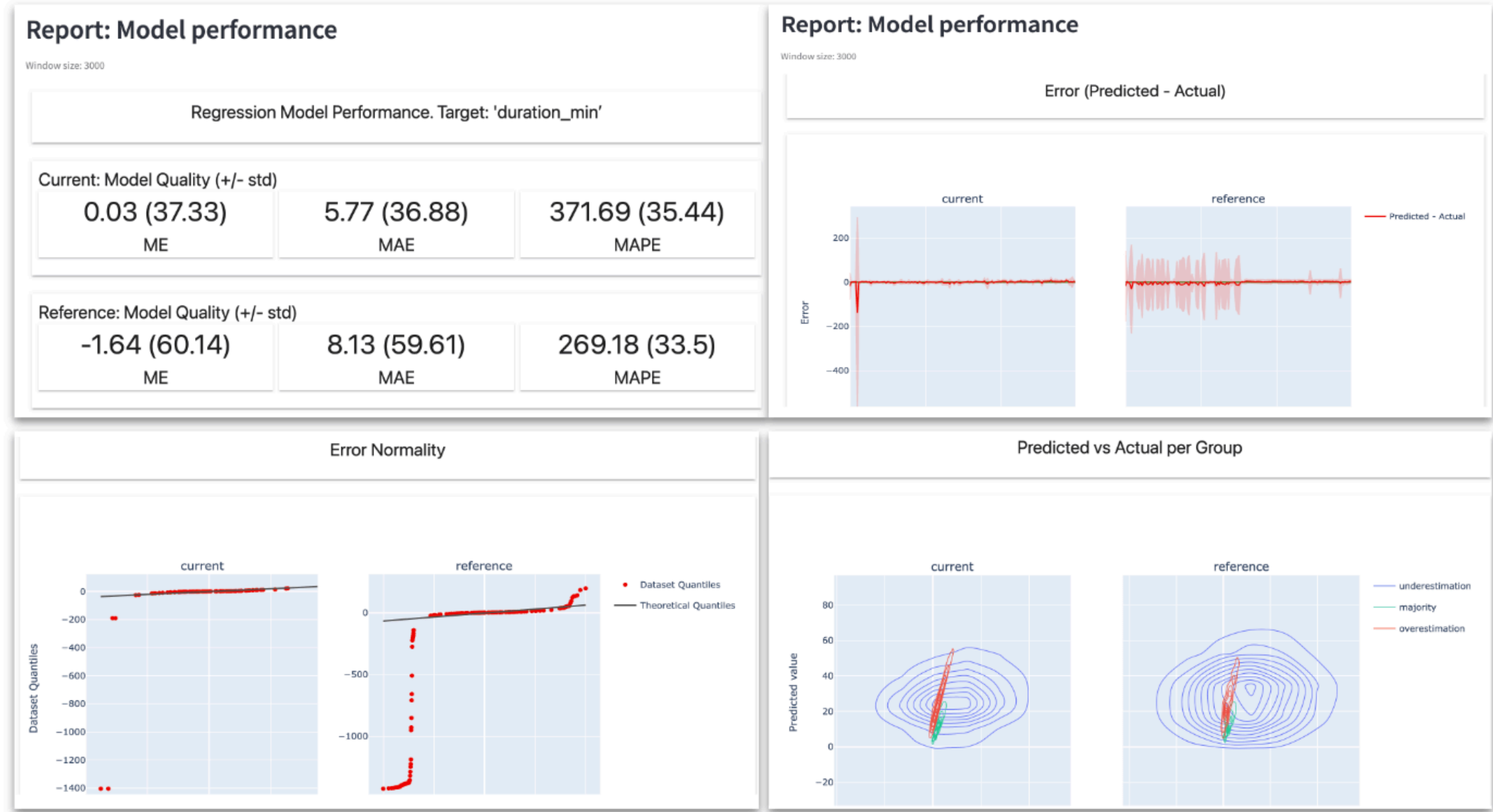
    report_path: Text = build_target_drift_report(
        reference_data=reference_data,
        current_data=current_data,
        column_mapping=column_mapping
    )

    return FileResponse(report_path)

```

By setting up these three endpoints, you effectively establish an API to generate model predictions and give visibility into the model quality.

3. Monitor ML performance



The function `build_model_performance_report()` from the `src/utils/reports.py` module is responsible for creating the model performance report using Evidently.

```
def build_model_performance_report(
    reference_data: pd.DataFrame,
    current_data: pd.DataFrame,
    column_mapping: ColumnMapping
) -> Text:

    model_performance_report = Report(metrics=[
        RegressionQualityMetric(),
        RegressionPredictedVsActualScatter(),
        RegressionPredictedVsActualPlot(),
        RegressionErrorPlot(),
        RegressionAbsPercentageErrorPlot(),
        RegressionErrorDistribution(),
        RegressionErrorNormality(),
        RegressionTopErrorMetric()
    ])
    model_performance_report.run(
        reference_data=reference_data,
        current_data=current_data,
        column_mapping=column_mapping
    )
    report_path = 'reports/model_performance.html'
    model_performance_report.save_html(report_path)

    return report_path
```

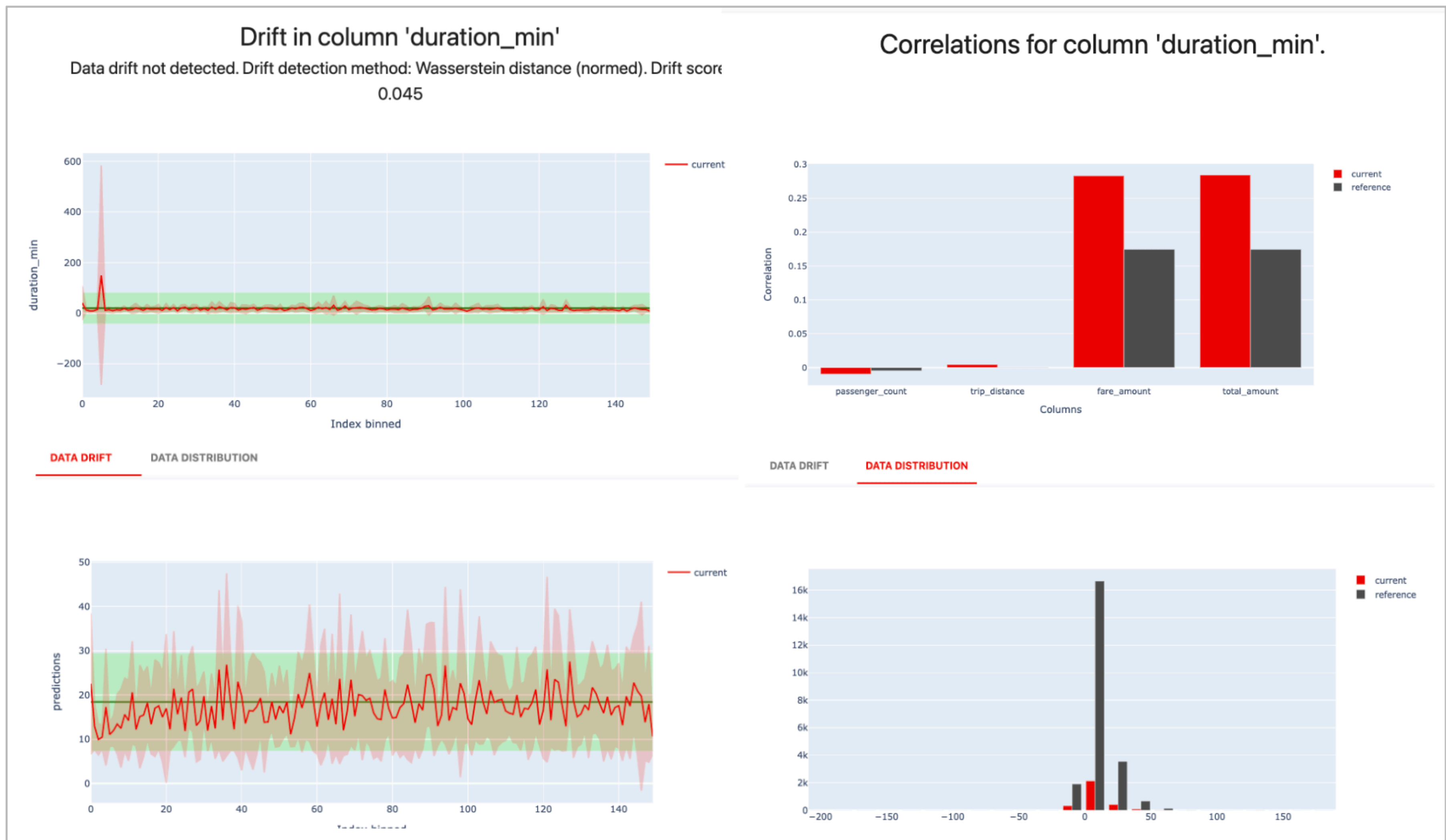
Here's a detailed look at what each part of the function does:

- Reference and current data.** The function takes as input the `reference_data` and `current_data` in the `pandas.DataFrame` format. The `reference_data` is the historical data, while the `current_data` is the most recent data. Both datasets include the features and target values.
- Column mapping.** The function takes the `ColumnMapping` object as input. This object defines the structure of the data, specifying which columns in the datasets correspond to the target, prediction, and features.
- Model performance report.** The function initializes a new Evidently **Report** object with a list of chosen metrics. These metrics are all built-in Evidently metrics for regression analysis.
- Saving the report.** The report is saved as an HTML file. The file name and path are specified in the `save_html` method of the `Report` object. This way, you can view the report in a web browser.
- Returning the report path.** Finally, the function returns the path to the saved HTML report. You can use it to access the report after it is created.

This function helps get a comprehensive model performance report using Evidently's built-in metrics for regression analysis. The report provides a detailed look at how the model performs, identifying potential issues and areas for improvement.

Generating Evidently reports. This function follows the Evidently API for creating and customizing Reports. Check out the official [docs](#) to explore how Column Mapping works and the additional parameters available for report customization. You might also prefer to include raw data in the visualizations like scatter plots.

4. Monitor target drift



The function `build_target_drift_report` generates a target drift report using Evidently.

```
def build_target_drift_report(
    reference_data: pd.DataFrame,
    current_data: pd.DataFrame,
    column_mapping: ColumnMapping
) -> Text:

    target_drift_report = Report(metrics=[TargetDriftPreset()])
    target_drift_report.run(
        reference_data=reference_data,
        current_data=current_data,
        column_mapping=column_mapping
    )
    report_path = 'reports/target_drift.html'
    target_drift_report.save_html(report_path)

    return report_path
```

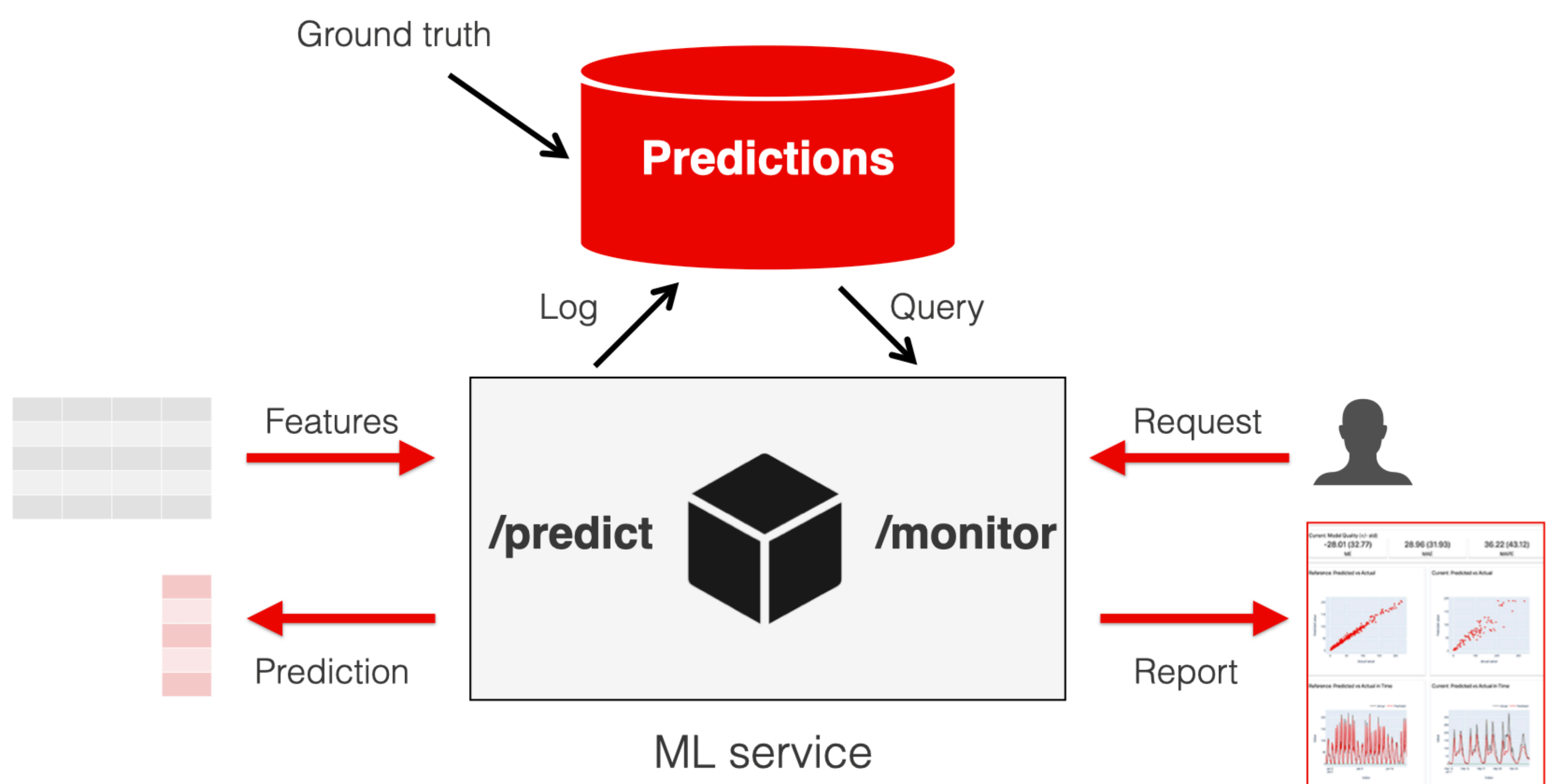
It follows a similar structure as the function used to generate the model performance report.

1. **Reference and current data.** As an input, it takes the `reference_data` and `current_data` datasets.
2. **Column mapping.** The function takes in a `ColumnMapping` object.
3. **Target drift report.** The function initiates a new Evidently `Report` object, but this time with the `TargetDriftPreset` metric. The `TargetDriftPreset` metric evaluates distribution drift in the target variable and provides an overview of the differences between the reference and current data.
4. **Saving the report.** The report is also saved as an HTML file.
5. **Returning the report path.** The function returns the path to the saved HTML report.

To sum up, this function helps create a detailed target drift report, providing insights into the changes in the target variable over time. This helps identify potential issues and the need for model updates.

☒ **Monitoring without ground truth.** This example assumes that actual values are available to compute the true model quality. In practice, they might come with a delay. In this scenario, you can run other checks: such as detecting drift in the model input data and model predictions and monitoring the data quality for any changes. Browse through available Evidently presets and metrics in the [docs](#) to choose what might be relevant for your use case.

Architecture pros and cons



[fs-toc-omit]Pros

On-demand metrics computation. In this example, metrics computation is integrated into the model serving, and you calculate them on demand. This way, you always get the most up-to-date view of your data and model performance. Since you do not compute metrics continuously, you might also need less storage.


Robust ML monitoring capabilities. Evidently provides comprehensive model monitoring: you can choose from 100+ metrics on data quality, data drift, and model performance, adjusting it to the specifics of your model and data pipelines.

Easy to customize. You can build upon this example architecture, extend it, or replace the specific components. You are not limited to the HTML reports: you can return the metrics computed by Evidently in JSON format, log them, and visualize them in the external system.

Friendly UI. Using a Streamlit app on top can help users browse the reports and simplify access to the metrics.

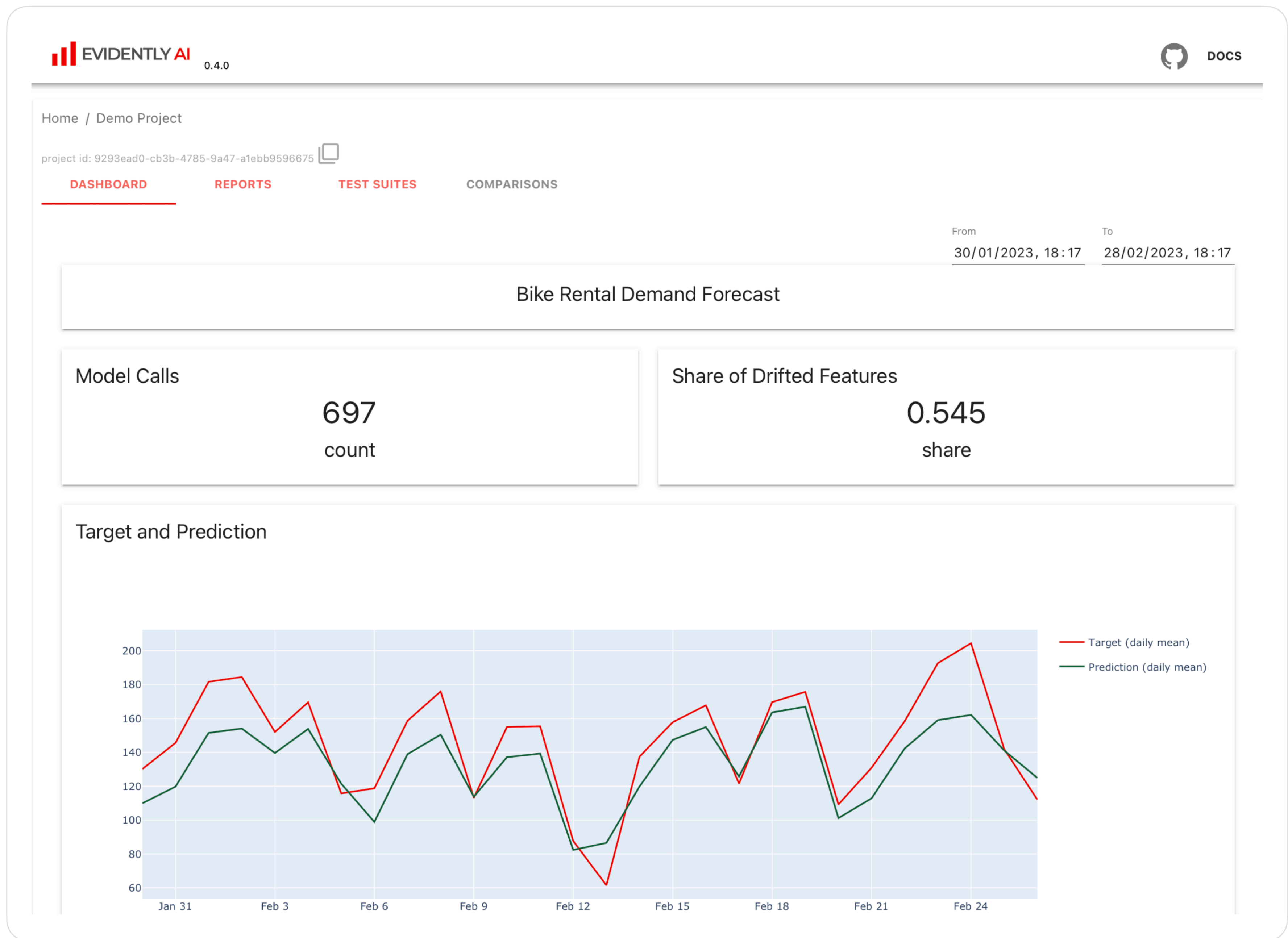
[fs-toc-omit]Cons

Static monitoring. With this architecture, you generate the reports with a “snapshot” of model quality. However, if you want to log and track model metrics in time, you might need to add more components. For example, you can host the Evidently Monitoring interface instead of Reports.

 **Track metrics over time.** You can add a dynamic visualization layer and host a monitoring dashboard to visualize metrics as they are computed over time. Here is [an example](#) of how to do this with Evidently.

Scalability and serving latency. In this example, the metric computation is incorporated within the model-serving application. While convenient, this can potentially affect serving latency, especially in high-load scenarios. To improve performance and scalability, you can decouple model serving and monitoring. For example, you can add a pipeline manager like Airflow or Prefect to schedule regular monitoring tasks.

 **End-to-end deployment blueprint.** Here is a [code example](#) that shows how to schedule model and data monitoring jobs and track metrics over time.



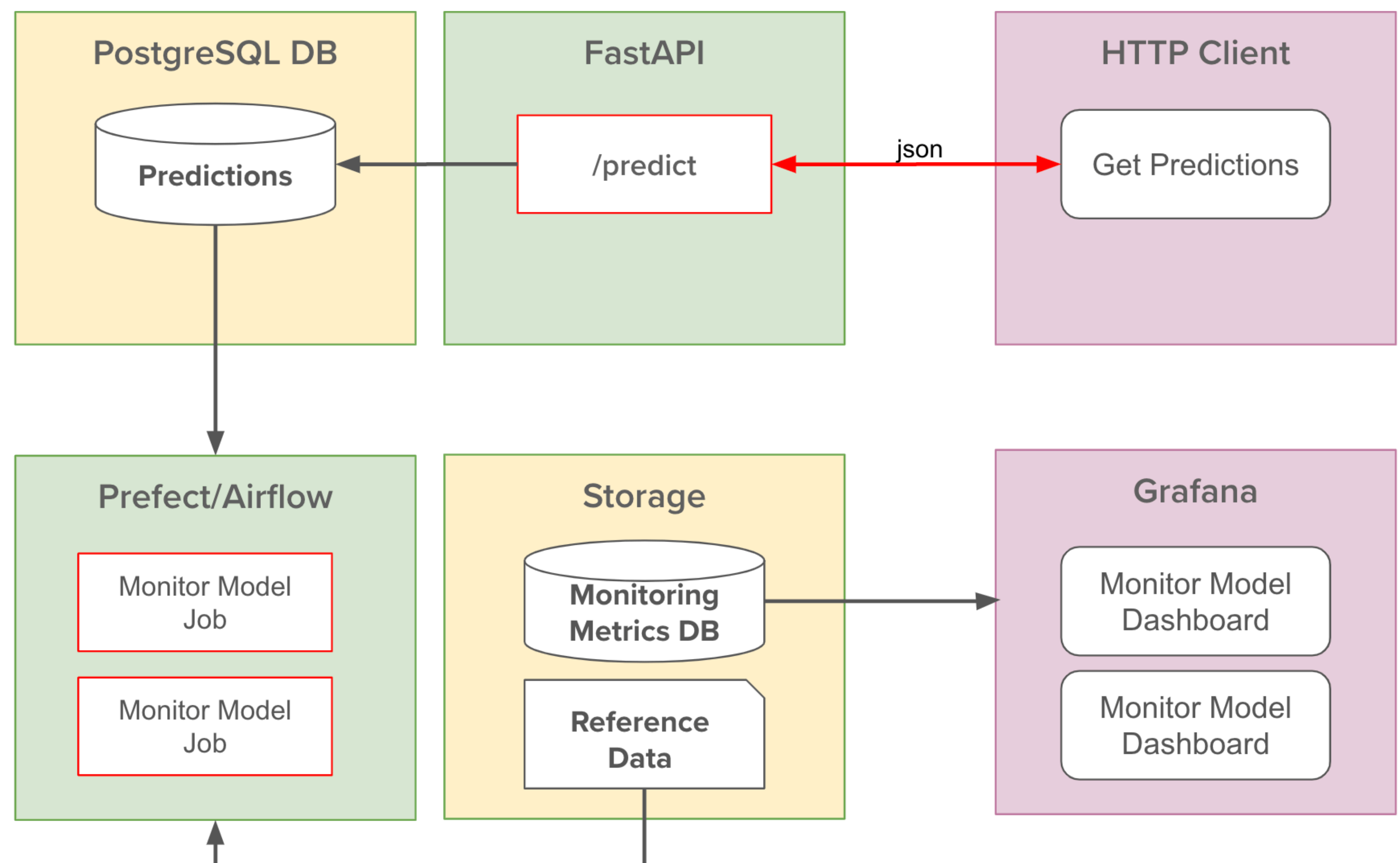
[Dynamic ML Monitoring](#) dashboard using [Evidently](#).

Add an orchestrator

The initial monitoring architecture is suitable for a range of applications and is simple to run. However, some of the **cons** listed above might be important. Consider extending the system design if you care about near real-time monitoring, logging historical model performance, scalability, and improved serving latency.

You can add the following components:

- A pipeline manager, like Prefect or Airflow.
- Evidently UI or a BI dashboard like Grafana.



Updated architecture with FastAPI, Prefect/Airflow and Grafana

Follow these general steps to extend the solution:

1. Delegate monitoring tasks to pipeline orchestration tools.

Use a tool like Prefect or Airflow (or any other orchestrator) to manage and schedule monitoring jobs at regular intervals. This will separate the ML model serving from monitoring. To do this, you should:

- **Set up a Prefect or Airflow environment.** Both tools provide Docker images to get started quickly.
- **Design the monitoring workflow.** This includes fetching data from the PostgreSQL database, calculating monitoring metrics using Evidently, and storing the metrics in a separate monitoring metrics database. In this scenario, you can generate the Evidently outputs as JSON or Python dictionary format.
- **Schedule the workflow.** Both Prefect and Airflow provide a scheduling feature to define the frequency of the monitoring job runs.

2. Design monitoring dashboards.

You can use the [Evidently UI](#) to host a live monitoring dashboard. You can save the Evidently Reports as JSON snapshots, and then launch a live dashboard that will automatically parse data from multiple Reports and help visualize it over time. You can choose which panels to visualize on the dashboard.

You can also design the workflow using other tools, like Grafana. In this case, you need to export the Evidently Metrics and store them in a database like PostgreSQL. You will then connect it as a data source for Grafana and can design dashboards and alerts.

 Want a detailed tutorial with code? Here is the [batch ML monitoring blueprint](#) with Evidently, PostgreSQL, Prefect, and Grafana.

Summing up

This tutorial showed how to build an ML monitoring solution for models deployed with FastAPI.

- You used FastAPI to serve ML models and expose a prediction endpoint.
- You simulated production model inference and logged the predictions to the PostgreSQL database.
- You used Evidently to evaluate the quality of ML models and data drift and served on-demand reports at FastAPI monitoring endpoints.
- You used Streamlit to create an interface for viewing monitoring reports.

You can further extend this example:

- Adapt it for your data and use case, replacing the demo model.
- Customize the contents of ML monitoring by choosing suitable metrics from the Evidently library.
- Extend this architecture by adding a pipeline manager and a visualization layer.

References

Thanks to Duarte O.Carmo for preparing the blog that inspired this integration example: [Monitoring ML models with FastAPI and Evidently AI](#), Duarte O.Carmo