International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I2P102

Volume 72 Issue 2, 9-15, February 2024
© 2024 Seventh Sense Research Group®

Review Article

MLOps Antipatterns and Mitigation Approaches

Ankit Virmani[1], Manoj Kuppam[2]

[1]*AI Advisor, Virufy Inc.,*
[2]*Site Reliability lead, Medline Industries.*

[1]*Corresponding Author : nktvirmani@gmail.com*

Received: 15 December 2023 Revised: 21 January 2024 Accepted: 05 February 2024 Published: 19 February 2024

Abstract *- Deploying machine learning models for various analytics and data applications at an enterprise scale brings diverse challenges. This paper breaks down these challenges and details the critical MLOps antipatterns - the practices to avoid while deploying the machine learning models. Like design patterns formalize software engineering wisdom, antipatterns help us recognize and communicate problematic methodologies. Some of these antipatterns stem from technical errors, while others arise from a lack of understanding of the context of ML usage. This paper aims to facilitate better documentation, collaboration, and faster problem resolution by establishing a common language around these antipatterns. In addition to outlining the antipatterns, the study provides solutions and best practices and suggests a path to a more mature MLOps approach.*

Keywords *- MLOps, Python, Security, Data, Deployment, DevOps.*

1. Introduction

While major advancements have been made in machine learning algorithms and models, significant gaps remain in managing operational complexities during real-world deployment. Studies show that over 50% of ML projects never make it to production [1]. This reveals a critical research gap in frameworks holistically addressing model governance, monitoring, and refinement through the entire machine learning lifecycle [2]. This perspective has led to the emergence of MLOps (Machine Learning Operations), a field dedicated to managing the entire lifecycle of ML models - from deployment and performance monitoring to ensuring the stability of production systems. Our work addresses this gap by providing a broader investigation into antipatterns spanning data, model development, deployment oversight, and evolution. Before delving into the antipatterns and the best practices, the study summarizes the key components of MLOps, which start from data extraction and preparation, model development, model training, tuning and deployment, and using the model and monitoring its performance.

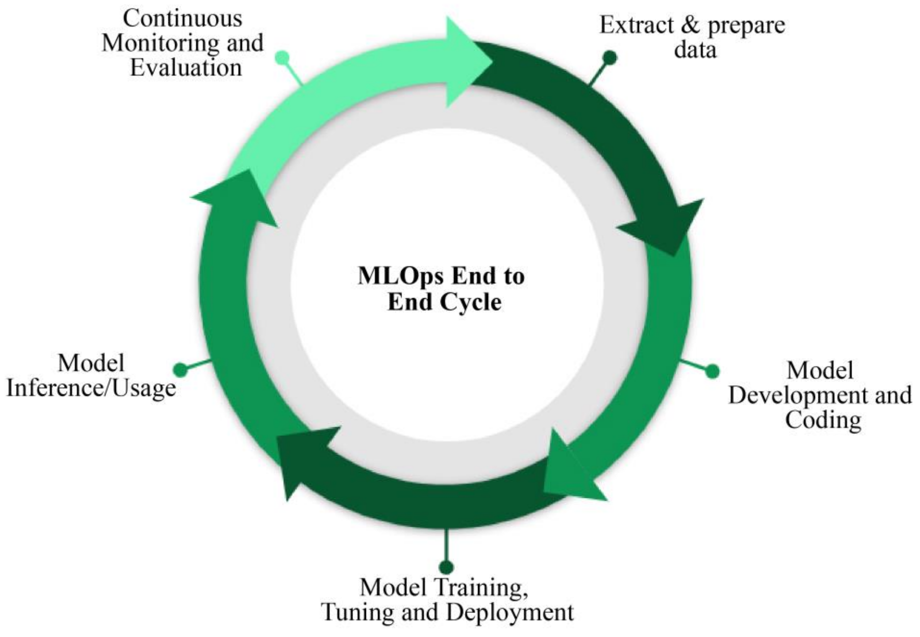
Continuous Monitoring and Evaluation
Extract & prepare data
MLOps End to End Cycle
Model Development and Coding
Model Training, Tuning and Deployment
Model Inference/Usage

Fig. 1 Different phases of an MLOps lifecycle

Antipatterns, much like design patterns in software engineering, provide a shared language to identify and communicate flawed practices. By recognizing and understanding antipatterns, ML teams can move beyond finger-pointing and work together to find effective solutions quickly. This work aligns with, yet complements, the efforts of [3], which explores MLOps through the concept of hidden technical debt. While [3] offers valuable software engineering insights, the authors take a broader view by focusing on data pipelines, the decision-making processes that rely on ML outputs, and the critical feedback loop that should be used to refine ML systems. Crucially, the study recognizes that the success of ML systems depends on multiple stakeholders, not just the ML developers.

The key contributions of this paper are:

- An Antipattern Vocabulary: it provides a list of antipatterns commonly encountered in ML pipelines, especially within financial analytics. While some of these may seem obvious after the fact, the authors believe that by formally documenting them, the study can foster a deeper understanding and improve the maturity of ML systems.
- Enterprise-Scale MLOps Recommendations: the article offers practical recommendations for documenting and managing MLOps at scale.

2. ANTIPATTERN: Disconnected Development and Operations

2.1. Problem

Data scientists develop models in isolated environments (e.g., laptops, notebooks) without collaboration with IT or operations. This leads to major challenges during production deployment due to compatibility, environment, and scalability issues.

1.2. Bad Practice Example

In this case, the data scientist is likely working in an isolated environment with specific library versions, hardware configurations, and dependencies. This model file, when handed off to production, can lead to compatibility and runtime issues.

```
**************************************************************************************/
*	Title: Xgboost in Python – Guide for Gradient Boosting
*	Author: Chauhan, G
*	Date: 2021
*	Availability: https://machinelearninghd.com/xgboost-in-python-guide-for-gradient-boosting/
*
**************************************************************************************/
```

```
# Data Scientist's development environment (e.g., local notebook)

import pandas as pd

import numpy as np

import sklearn

# ... Data loading, model development (not shown for brevity)

# Save the model using pickle (or similar)

import pickle

with open("my_model.pkl", "wb") as f:

        pickle.dump(model, f)
```

1.3. Solution

Foster DevOps principles in ML workflows. Use tools that bridge the gap between development and production using environment parity, version control for data and code, and standardized deployment processes. The image below illustrates the challenge and some possible solutions [4]:

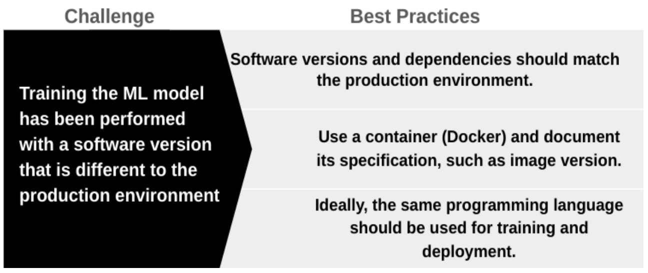
Challenge
Best Practices
Training the ML model has been performed with a software version that is different to the production environment
Software versions and dependencies should match the production environment.
Use a container (Docker) and document its specification, such as image version.
Ideally, the same programming language should be used for training and deployment.

Fig. 2 Challenges and best practices in training ML model

Below is the best practice where the dependencies are explicitly listed in their development environment. This helps bridge the gap towards the production environment.

```
#Development with a requirements.txt file:

pandas==1.3.0  # Use specific versions to match the production environment

numpy==1.22.1

scikit-learn==1.0.2

#Containerization with a Dockerfile:

#This Dockerfile defines a reproducible environment specifying the dependency #versions, base OS image, and runtime instructions.

#Code snippet

FROM python:3.9-slim  # Base image with Python

WORKDIR /app  # Set the working directory
```

```
# Copy the dependency list to the container

COPY requirements.txt requirements.txt

# Install dependencies before adding the model code for efficiency

RUN pip install -r requirements.txt

# Copy the model training code to the container

COPY train_model.py train_model.py

# Define the command to execute to train the model

CMD ["python", "train_model.py"]
```

3. ANTIPATTERN: Manual Experimentation and Tracking

3.1. Problem

Relying on manual spreadsheets or inconsistent methods to track experiments, hyperparameters, model versions, and results. This makes it extremely difficult to reproduce results, compare model performance, and understand lineage.

3.2. Bad Practice Example

In this case, the data scientist is likely working in an isolated environment with specific library versions, hardware configurations, and dependencies. This model file, when handed off to production, can lead to compatibility and runtime issues.

```
# ... train multiple models using a for loop, keeping track in a notebook or manually

for learning_rate in [0.01, 0.05, 0.1]:

        for num_layers in [2, 3, 4]:

         # ... train the model (code not shown)

         # ... log the results manually to a spreadsheet or notebook
```

3.3. Solution

Adopt experiment-tracking tools that automatically log metadata, model artifacts, and performance metrics. Centralize and visualize results across teams to facilitate collaboration and rapid comparison. The diagram below illustrates the triggers that cause models to change, the antipattern to maintain those changes and the best practices [7].

Below is an example of how to monitor metrics in mlflow.

```
import mlflow

# ... Within your model training function

with mlflow.start_run():

        mlflow.log_param("learning_rate", learning_rate)

        mlflow.log_param("num_layers", num_layers)

  # ... train the model (code not shown)

  mlflow.log_metric("accuracy", accuracy)
```

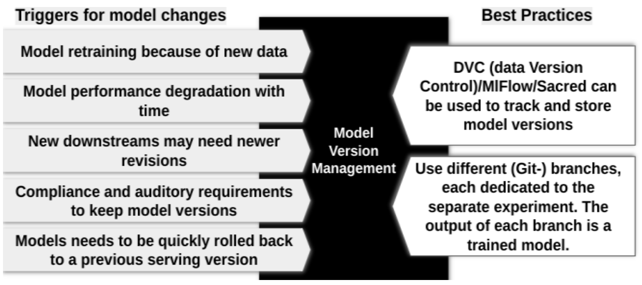
Triggers for model changes
Model retraining because of new data
Model performance degradation with time
New downstreams may need newer revisions
Compliance and auditory requirements to keep model versions
Models needs to be quickly rolled back to a previous serving version
Model Version Management
Best Practices
DVC (data Version Control)/MlFlow/Sacred can be used to track and store model versions
Use different (Git-) branches, each dedicated to the separate experiment. The output of each branch is a trained model.

Fig. 3 Triggers that cause models to change and the corresponding best practices

4. ANTIPATTERN: Lack of Security and Governance

4.1. Problem

Ignoring security best practices for sensitive data, model access, and change control. This can result in data breaches, unauthorized access, and model manipulation, leading to regulatory compliance issues.

4.2. Bad Practice Example

In this case, the data scientist is likely working in an isolated environment with specific library versions, hardware configurations, and dependencies. This model file, when handed off to production, can lead to compatibility and runtime issues.

```
# ... Model training on a single machine

# ... Access credentials are hardcoded in plain text

import some_cloud_sdk

client = some_cloud_sdk.Client("my_username", “my_password”)
```

4.3. Solution

Enforce strong authentication and authorization, implement role-based access control mechanisms, monitor model behavior for security anomalies, and regularly audit ML systems for vulnerabilities [7]. Create policies around data handling, model usage, and regulatory compliance. Some of the typical attacks on the machine learning models, along with the potential solutions, are listed below [7]:

4.3.1. Online Adversarial Attack

Attacker changes the online continuous stream of training data.
Best Practice: Data Versioning, Data Security controls

4.3.2. Distributed Denial of Service Attack

The attacker disrupts model operations by sending complicated problems.

Best Practice: A resilient production model service architecture can stand up to sudden spikes in service requests.

4.3.3. Data Phishing Privacy Attack

Hackers break confidentiality by reverse-engineering the datasets.
Best Practice: Data Encryption, Data access controls, encrypted input and encrypted model exposed to the users.

4.3.4. Adversarial Machine Learning Attack

Attacker changes the input data

Best Practice: Model Monitoring, Data and Model Access Controls, Addition of intentional noise in training.

4.3.5. Data Poisoning Attack

Attacker changes the training data

Best Practice: Model Monitoring, Data and Model Access Controls, model drift controls.

A good practice of using secrets management and environment variables is exemplified below:

```
import os

import some_cloud_sdk

username = os.getenv('CLOUD_USERNAME')

password = os.getenv('CLOUD_PASSWORD')

client = some_cloud_sdk.Client(username, password)
```

5. ANTIPATTERN: Ad-hoc Model Deployment and Monitoring

5.1. Problem

Deploying models in a haphazard manner without proper performance monitoring, health checks, or rollback procedures. This can cause downtime, unexpected behavior, and severe customer-facing issues.

5.2. Bad Practice Example

In this case, the data scientist is likely working in an isolated environment with specific library versions, hardware configurations, and dependencies. This model file, when handed off to production, can lead to compatibility and runtime issues.

Below is a code example:

```
# ... Manually copy the model to the production server over a file share

# ... Spin up a custom Flask application on a dedicated server

# ... No health checks or monitoring

client = some_cloud_sdk.Client("my_username", "my password")
```

5.3. Solution

Establish a robust CI/CD (Continuous Integration/Continuous Delivery) pipeline for model deployment. Include automated testing, health checks, A/B testing, canary deployments, and rollback mechanisms to minimize disruptions and quickly revert to stable versions in case of problems.

The table below illustrates some metrics to monitor and what actions to take [4]:

Table 1. Metrics and actions for models

What to monitor	Actions to take
Data Schema Variation	Alert if input data schema mismatches with training data schema [4].
Whether the training and serving features compute the same value.	Log a traffic sample. Calculate statistical metrics (min, max, avg. value, % of missing values, etc.) on training and sample features [4].
Numerical stability of the model.	Alert for nulls/infinity/undefined [4].
Degradation of the predictive quality of the ML model.	Measure statistical bias in predictions.
Computational performance of an ML system.	Measure the model performance, along with CPU memory utilization [4].

A code example for deployment is below:

```
*********************************************************************************************/

*	Title: MLFlow Models

*	Availability:

https://mlflow.org/docs/2.3.1/models.html

*********************************************************************************************/

# During Development

import mlflow.pyfunc
```

```
mlflow.pyfunc.save_model(path="my_model",
python_model=model)

# Deployment using CI/CD process

mlflow models serve -m ./my model -p 5001  # Serve
model with API
```

Example 2: monitoring using MLFlow :

```
#setup

conda env create --file conda.yaml

conda activate mlflow-model-monitoring

#train and register 2 models

python custom_model_train.py

#This will create two registered models: sklearn-monitor
and sklearn-monitor-custom

#start model server

mlflow models serve --port 5002 --model-uri
models:/sklearn-monitor/production

#start proxy server

python proxy_server.py --port 5001 --mlflow-model-
server-uri http://localhost:5002/invocations --log_dir out

#The proxy server forwards the request to the actual model
server and then logs the input and output data as a CSV
file.

#example predictions:

curl -X POST \

  -H "accept: application/json" \

  -H "Content-Type:application/json" \

  -d '{ "columns": [ "alcohol", "chlorides", "citric acid",
"density", "fixed acidity",

                    "free sulfur dioxide", "pH", "residual sugar",
"sulphates",

                    "total sulfur dioxide", "volatile acidity" ],

        "data": [[ 7,   0.27, 0.36, 20.7, 0.045, 45, 170, 1.001,
3,    0.45,  8.8 ],

 [ 6.3, 0.3,  0.34,  1.6, 0.049, 14, 132, 0.994,  3.3,  0.49,
9.5 ] ] }' \

                    http://localhost:5001/invocations
```

Each request for scoring will generate a CSV file containing the input data and the prediction. The following example shows data for three requests. The CSV file is written to the local directory specified in MLFLOW_MONITORING_DIR

6. ANTIPATTERN: Lack of Ethical Considerations

6.1. Problem

Not actively considering the ethical implications, fairness, or bias present in data and models. This can lead to discriminatory outputs, reputational damage, and a loss of trust. The diagram below illustrates how bias in the models can prevail across the various MLops stages:

6.2. Bad Practice Example

In this case, the data scientist is likely working in an isolated environment with specific library versions, hardware configurations, and dependencies. This model file, when handed off to production, can lead to compatibility and runtime issues.

# Training a model on a dataset that has potential biases or sensitive features

# ... (Model training code that does not evaluate fairness or biases)

6.3. Solution

Incorporate fairness and explainability checks into the ML pipeline. Evaluate models on different demographic groups, monitor feedback and bias reports, and provide clear explanations as to how model outputs are generated to foster transparency and accountability.

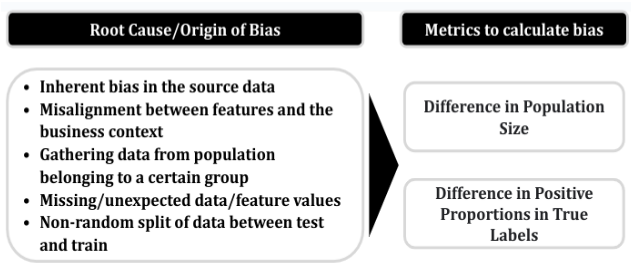
Root Cause/Origin of Bias
Metrics to calculate bias
• Inherent bias in the source data
• Misalignment between features and the business context
• Gathering data from population belonging to a certain group
• Missing/unexpected data/feature values
• Non-random split of data between test and train
Difference in Population Size
Difference in Positive Proportions in True Labels

Fig. 4 Root cause and metrics to measure bias in data engineering phases of MLOps

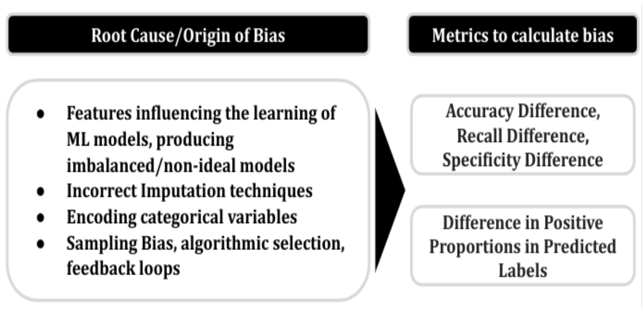
Root Cause/Origin of Bias
Metrics to calculate bias
• Features influencing the learning of ML models, producing imbalanced/non-ideal models
• Incorrect Imputation techniques
• Encoding categorical variables
• Sampling Bias, algorithmic selection, feedback loops
Accuracy Difference, Recall Difference, Specificity Difference
Difference in Positive Proportions in Predicted Labels

Fig. 5 Root cause and metrics to measure bias in machine learning phases of MLOps

The code example below uses SHAP- Shapley Additive Explanations to identify the weighting of the features on the label, explain the reasoning behind the prediction, and enhance the mode explainability. [7]

```
*****************************************************************************************/
*	Title: Shapley Additive Explanations: Unveiling the Black Box of Machine Learning
*	Author: Gomede, E
*	Date: 2023
*	Availability: https://medium.com/@evertongomede/shapley-additive-explanations-unveiling-the-black-box-of-machine-learning-477ba01ffa07
*
*****************************************************************************************/
import shap

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

# Generate synthetic data for demonstration purposes

np.random.seed(0)

X, y = shap.datasets.adult()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a simple model (Random Forest Classifier)

model = RandomForestClassifier(n_estimators=100, random_state=42)

model.fit(X_train, y_train)

# Create an explainer object

explainer = shap.Explainer(model, X_train)

# Calculate SHAP values for a single instance (e.g., the first instance in the test set)

sample_idx = 0

shap_values = explainer.shap_values(X_test.iloc[sample_idx, :])

# Calculate SHAP values for the entire test set

shap_values_all = explainer.shap_values(X_test)

# Visualize a summary plot for the entire test set

shap.summary_plot(shap_values_all, X_test, feature_names=X.columns)

# You can also visualize individual predictions

shap.initjs()

shap.force_plot(explainer.expected_value[1], shap_values_all[1][sample_idx], X_test.iloc[sample_idx, :])
```

7. ANTIPATTERN: Lack of Data Versioning

7.1. Problem

Using static datasets without tracking changes or not thoroughly validating data quality before model training. This leads to data drift, model performance degradation, and unexpected outcomes in production.

7.2. Bad Practice Example

In the example below, a static file is being used without any versioning:

```
*****************************************************************************************/
*	Title: How to Create a Machine Learning Model in Python for Sales Prediction
*	Author: Pandata
*	Date: 2023
*	Availability: https://anello.substack.com/p/how-to-create-a-machine-learning
*
*****************************************************************************************/

import pandas as pd

# Load data from a CSV file without any versioning

data = pd.read_csv("data/customer_data.csv")

# ... perform data cleaning and training (not shown)
```

7.3. Solution

Implement rigorous data versioning and quality control mechanisms. Track the evolution of data, identify schema changes, and maintain a history of transformations—automate data quality checks to ensure consistency and catch anomalies early in production.

Below is an example of using data versioning:

```
import pandas as pd

import dvc.api

# Retrieve the specific version of the data using DVC

with dvc.api.open(

    'data/customer_data.csv',

    repo='<your_repo_location>',  # Could be a remote repository
```

```
        rev='<version_tag_or_hash>'
) as fd:
            df = pd.read_csv(fd)
```

8. Conclusion

MLOps is important for managing and maintaining machine learning models in real-world applications. By establishing the best practices and tools within the CI/CD (Continuous Integration/Continuous Delivery) environment, MLOps aims to prevent 'technical debt' and ensure the smooth integration and deployment of ML models alongside other software components.

The fundamental tenets of MLOps include iterative development, automation, versioning, testing, reproducibility, and continuous monitoring to accelerate ML adoption and deliver intelligent software efficiently.

References

[1] Jaime Hampton, Half of AI Models Never Make It To Production: Gartner, Datanami.com, 2022. [Online]. Available: https://www.datanami.com/2022/08/22/half-of-ai-models-never-make-it-to-production-gartner/

[2] A. Kumar, S. Gupta, and P. Rai, "Explainable AI in Practice: From Academia to Industry," *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pp. 17-26, 2021.

[3] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," *NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems*, vol. 2, pp. 2503-2511, 2015. [Google Scholar] [Publisher Link]

[4] Larysa Visengeriyeva et al., MLOps Principles, Web.Archive.org. [Online]. Available: https://web.archive.org/web/20210620182858/https:/ml-ops.org/content/mlops-principles

[5] ML Model Security – Preventing the 6 Most Common Attacks, Excella, 2021. [Online]. Available: https://www.excella.com/insights/ml-model-security-preventing-the-6-most-common-attacks

[6] Vitomir Jovanović, Mlflow-Model-Monitoring, Dev Genius, 2022. [Online]. Available: https://blog.devgenius.io/mlflow-for-model-monitoring-cb8b2177b67a

[7] Everton Gomede, Shapley Additive Explanations: Unveiling the Black Box of Machine Learning, Medium, 2023. [Online]. Available: https://medium.com/@evertongomede/shapley-additive-explanations-unveiling-the-black-box-of-machine-learning-477ba01ffa07