Mitigating Drift in Machine Learning Systems through Continuous Input Monitoring

An Architectural Proposal and Empirical Evaluation of Detection Methods

Lucas Helfstein Rocha

Thesis presented to the
Institute of Mathematics, Statistics
and Computer Science
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program: Computer Science
Advisor: Prof.ª Dr.ª Kelly Rosa Braghetto

São Paulo
January, 2026

Mitigating Drift in Machine Learning Systems through Continuous Input Monitoring

An Architectural Proposal and Empirical Evaluation of Detection Methods

Lucas Helfstein Rocha

This is the original version of the thesis
prepared by candidate Lucas Helfstein Rocha,
as submitted to the Examining Committee.

Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Rocha, Lucas Helfstein
Mitigating Drift in Machine Learning Systems through Continuous Input Monitoring: An Architectural Proposal and Empirical Evaluation of Detection Methods / Lucas Helfstein Rocha; orientadora, Kelly Rosa Braghetto. - São Paulo, 2026.
94 p.: il.

Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão original

1. Machine Learning Systems. 2. Data Drift. 3. Concept Drift. 4. MLOps. I. Braghetto, Kelly Rosa. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Acknowledgements

You must unlearn what you have learned.
— Master Yoda

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Kelly Rosa Braghetto, who has guided me since my undergraduate years. Her expertise, insightful feedback, and constant encouragement have been fundamental to making this work possible.

I am also profoundly thankful to my family for the unwavering support they have given me throughout my academic journey, which ultimately led me to pursue this master's degree. My mother and grandmother have always believed in me, and I would not be here without their encouragement and trust.

To my partner, who shared most of the time I dedicated to this research, I owe special thanks. Your patience, understanding, and support during the most challenging moments of this program have been invaluable to me.

I am grateful as well to my friends, both inside and outside academia, who never let me consider giving up and continuously motivated me to reach this goal.

Lastly, I extend my thanks to the entire Computer Science community at IME-USP. It is a place where I have always felt welcome and where I have seen science treated with the seriousness and care it truly deserves.

Resumo

Lucas Helfstein Rocha. **Mitigando Desvios em Sistemas de Aprendizado de Máquina por meio de Monitoramento Contínuo das Entradas: *Uma Proposta Arquitetural e Avaliação Empírica de Métodos de Detecção***. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2026.

A engenharia de sistemas de aprendizado de máquina (AM) representa uma mudança de paradigma em relação à engenharia de software convencional. Enquanto sistemas tradicionais são governados por lógica determinística, os sistemas de AM são fundamentalmente moldados pelos dados, fazendo com que sua robustez não resida apenas na qualidade do código, mas na estabilidade da relação entre os dados de treinamento e os dados de produção. Essa relação, no entanto, é inerentemente frágil. Flutuações, tendências sazonais ou mudanças no ambiente operacional podem levar a desvios de dados (*data drifts*) e de conceito (*concept drifts*), erodindo de forma silenciosa e contínua o desempenho dos modelos de AM. Portanto, garantir a robustez e a confiabilidade de sistemas de AM em produção é um desafio que transcende a manutenção de código, exigindo uma vigilância contínua sobre seu componente mais volátil: os dados.

A adoção de sistemas de aprendizado de máquina em grande escala avançou mais rapidamente do que o desenvolvimento de arcabouços arquiteturais formais para orientar seu projeto, governança e manutenção de longo prazo. Esta dissertação aborda essa lacuna ao propor uma arquitetura conceitual baseada em componentes que operacionaliza práticas robustas de MLOps. A arquitetura aprimora a rastreabilidade e a adaptabilidade ao organizar o sistema em subsistemas claramente definidos, responsáveis por gerenciar todo o ciclo de vida do aprendizado de máquina. Ela é apoiada por um modelo de dados abrangente que garante que todos os artefatos, desde dados brutos até modelos implantados, sejam localizáveis, acessíveis, interoperáveis e reutilizáveis, possibilitando ciclos de *feedback* eficazes e monitoramento contínuo das entradas.

O princípio central da arquitetura é que o monitoramento contínuo dos dados de entrada é essencial para manter o desempenho dos modelos em ambientes dinâmicos. Para validar esse princípio, foi conduzida uma série de experimentos empíricos com conjuntos de dados que apresentam tanto desvio de conceito quanto desvio de dados, simulando os desafios de cenários reais de produção. O método *Hellinger Distance Drift Detection* foi utilizado como técnica principal de detecção, complementado por uma variante proposta baseada na divergência de Jensen–Shannon e por uma análise comparativa com o teste de Kolmogorov–Smirnov.

Os resultados demonstram que o retreinamento oportuno de modelos de AM, acionado pelo monitoramento das entradas, gera ganhos substanciais de desempenho em todos os tipos de desvio. Embora o teste de Kolmogorov–Smirnov tenha apresentado alta sensibilidade, a abordagem baseada em Jensen–Shannon apresentou ganhos significativos de eficiência computacional em relação à distância de Hellinger. Esses achados confirmam o papel crítico do monitoramento das entradas e fornecem validação empírica para a arquitetura proposta, apontando um caminho concreto para a construção de sistemas de aprendizado de máquina mais robustos, adaptativos e sustentáveis.

Palavras-chave: Sistemas de Aprendizado de Máquina. MLOps. Arquitetura de Software. Desvio de Dados. Desvio de Conceito.

Abstract

Lucas Helfstein Rocha. **Mitigating Drift in Machine Learning Systems through Continuous Input Monitoring: *An Architectural Proposal and Empirical Evaluation of Detection Methods***. Thesis (Master's). Institute of Mathematics, Statistics and Computer Science, University of São Paulo, São Paulo, 2026.

Machine learning (ML) systems engineering represents a paradigm shift from conventional software engineering. While traditional systems are governed by deterministic logic, ML systems are fundamentally shaped by data, meaning that their robustness lies not only in the quality of the code but also in the stability of the relationship between training data and production data. This relationship, however, is inherently fragile. Fluctuations, seasonal trends, or changes in the operating environment can lead to data drift and concept drift, silently and continuously eroding the ML model's performance. Therefore, ensuring the robustness and reliability of ML systems in production is a challenge that transcends code maintenance, requiring continuous vigilance over its most volatile component: data.

The adoption of large-scale machine learning systems has advanced more rapidly than the development of formal architectural frameworks to guide their design, governance, and long-term maintenance. This dissertation addresses this gap by proposing a component-based conceptual architecture that operationalizes robust MLOps practices. The architecture improves traceability and adaptability by organizing the system into clearly defined subsystems responsible for managing the entire machine learning lifecycle. It is supported by a comprehensive data model that ensures all artifacts, from raw data to deployed models, are findable, accessible, interoperable, and reusable, enabling effective feedback loops and continuous monitoring of inputs.

The central principle of the architecture is that continuous monitoring of input data is essential to maintain model performance in dynamic environments. To validate this principle, a series of empirical experiments was conducted with datasets that exhibit both concept drift and data drift, simulating the challenges of real production scenarios. The Hellinger Distance Drift Detection method was used as the main detection technique, complemented by a proposed variant based on Jensen–Shannon divergence and a comparative analysis with the Kolmogorov–Smirnov test.

The results demonstrate that timely model retraining, triggered by input monitoring, generates substantial performance gains across all types of drift. Although the Kolmogorov–Smirnov test showed high sensitivity, the Jensen–Shannon-based approach offered significant computational efficiency gains over the Hellinger distance. These findings confirm the critical role of input monitoring and provide empirical validation for the proposed architecture, pointing to a concrete path for building more robust, adaptive, and sustainable machine learning systems.

Keywords: Machine Learning Systems. MLOps. Software Architecture. Data Drift. Concept Drift.

List of Abbreviations

IaaS	Infrastructure as a Service
HD	Hellinger distance
HDDDM	Hellinger distance Drift Detection Method
JS	Jensen-Shannon divergence
JSDDM	Jensen-Shannon Drift Detection Method
KL	Kullback-Leibler divergence
KS	Kolmogorov-Smirnov
KSDDM	Kolmogorov-Smirnov Drift Detection Method
LLM	Large language model
ML	Machine learning
MLOps	Machine learning operations
MLaaS	Machine Learning as a Service
SE4ML	Software Engineering for Machine Learning
XAI	Explainable Artificial Intelligence

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software systems incorporating machine learning components have been widely adopted across multiple domains. Studies carried out by McKinsey (Cam *et al.*, 2019; Balakrishnan *et al.*, 2020) indicate not only a significant increase in the use of applications based on machine learning, but also that the most successful companies in this space consistently adopt advanced practices and techniques throughout their development processes. Among these practices is *MLOps*, a set of methodologies and tools aimed at building, deploying, and maintaining machine learning systems. Although originally described in early reports as an emerging discipline, *MLOps* has since matured and become a well-established component of modern machine learning pipelines, with its adoption expanding significantly in recent years. Developing systems with machine learning models poses different challenges than those faced in traditional software systems, since the performance of these models is strongly dependent on the nature of the input data. Once a model has been deployed in a production environment, it is common for it to receive data that differs in varying degrees from that used during its training phase (Gama and Castillo, 2006). This discrepancy can lead the system to exhibit unexpected behavior, with the potential to significantly compromise the quality of its results. In this context, decisions regarding operational requirements – such as the availability, monitoring, and retraining of models – become fundamental for ensuring the proper functioning of systems (Sculley *et al.*, 2015).

Monitoring plays a central role in ensuring the reliability of ML systems, especially in production environments where data is constantly evolving. As highlighted by Schröder and Schulz (2022), these systems are particularly prone to subtle and hard-to-detect failures, such as silent performance degradation and issues stemming from high-dimensional or adversarial inputs. These risks reinforce the need for continuous validation of input data and performance metrics throughout the system's lifecycle. Rather than relying solely on traditional testing strategies, which are often insufficient after deployment, effective ML systems require monitoring practices that operate as ongoing, adaptive checks—anticipating changes and preserving robustness in dynamic environments.

Given this scenario, this master's thesis aimed to investigate the challenges involved in developing and maintaining systems that incorporate machine learning components. In particular, it aims to understand how problems related to model performance degradation

can be mitigated through effective monitoring practices and a systematic organization of the data used throughout the model's life cycle.

1.1 Objectives

The main objective of this work is to investigate how drift monitoring can be incorporated into machine learning systems and to evaluate its impact on maintaining the overall quality of such systems. In particular, this work seeks to answer the following research questions:

- **RQ1:** How should the architecture of a machine learning system be structured to enable effective drift monitoring across all stages of the model life cycle?
- **RQ2:** In what ways can drift monitoring contribute to improving model performance in production environments?

Building on these objectives, this work proposes a generic software architecture for supervised machine learning systems designed to facilitate their evolution and long-term maintainability. This architecture emphasizes the role of components that enable effective drift monitoring, ensuring that changes in data distribution can be detected and handled throughout the model life cycle. Furthermore, this work presents empirical experiments that compare the behavior of a system equipped with drift monitoring against one that operates without it, demonstrating the practical impact of monitoring on maintaining model performance in production.

In contrast to other approaches found at the intersection of machine learning and software engineering literature (Serban and Visser, 2022; Nascimento *et al.*, 2020; Washizaki *et al.*, 2019), this thesis addresses the maintainability challenges of machine learning–based systems through an architecture specifically tailored for drift-aware operation. The proposed design draws inspiration from widely adopted open-source tools used across different stages of the machine learning pipeline and reflects practices observed in real-world development environments.

1.2 Contributions

Based on the outlined objectives, this work presents two main contributions that aim to address the challenges associated with the continuous operation of machine learning systems. Both contributions are based on the premise that monitoring is essential to mitigate the impacts caused by changes in data and to promote the constant evolution of models.

1. The first contribution consists of the proposition of an architecture for software systems that incorporate supervised machine learning, encompassing the necessary components to support the monitoring of all stages in the life cycle of deployed models. This architecture was designed to offer a comprehensive and structured view of model behavior in production, enabling developers to make informed decisions that enhance the system's robustness and reliability through continuous improvement.

2. The second contribution consists of an empirical analysis of data drift detection techniques applied to the monitoring of model input data. This analysis aimed to demonstrate how such techniques can benefit machine learning systems that are exposed to various types of input data variations, highlighting their potential to improve model performance over time. The results were described in a paper published in the proceedings of the 39th Brazilian Symposium on Databases (Helfstein and Braghetto, 2024) and in an article to appear in the Journal of Information and Data Management (Helfstein and Braghetto, n.d.).

1.3 Thesis Structure

This section outlines the organization of this master's thesis. Following this introductory chapter, which establishes the study's objectives and highlights its key contributions, the subsequent chapters are structured as follows: Chapter 2, Concepts, establishes the theoretical foundation, covering machine learning fundamentals, various forms of data drift, and aspects of ML systems, MLOps, MLaaS, and AutoML. Chapter 3, Related Work, reviews pertinent literature, focusing on drift detection, software engineering for machine learning, and relevant development tools.

Chapter 4, Architectural Proposal, presents the core architectural framework, detailing its feedback cycle, data flow, and components for the ML lifecycle. Chapter 5, Detecting Drift in Datasets, describes the empirical methodology, including the datasets and an analysis of drift detection techniques.

Chapter 6, Using drift detection within ML Systems, describes the approach to improving classifier performance, followed by a discussion of experimental results from multi-dataset and multi-run synthetic experiments. Finally, Chapter 7, Conclusion, summarizes the key findings and proposes future research directions.

Chapter 2

Concepts

This chapter presents the fundamental concepts in the field of machine learning that are necessary for understanding the research results discussed in this master's thesis. More specifically, it introduces core notions related to supervised learning, data streams, and the different types of drift that can affect machine learning models, including *data drift* and *concept drift*. Additionally, the chapter reviews several drift detection methods commonly employed in the literature, as well as the principles underlying monitoring strategies used in machine learning systems. Together, these concepts establish the theoretical foundation upon which the proposed architecture and experimental evaluation are built.

2.1 Machine Learning

Machine learning is a branch of artificial intelligence that focuses on using data and algorithms to enable computers to recognize patterns and learn desired behaviors. The data that is used in learning is called input data. On the other hand, the data generated by the machine's learned behavior is called output data. The definitions presented in the following are based on the books *Learning From Data* (Abu-Mostafa *et al.*, 2012) and *Understanding Machine Learning* (Shalev-Shwartz and Ben-David, 2014), with some adaptations made to facilitate a better understanding of the objectives and results of this work.

We define an *instance* (or *example*) as a vector $x = \{a_1, a_2, \ldots, a_M\}$, where each a_i is called an *attribute* (or *feature*) of x, with $1 \leq i \leq M$. For each attribute a_i, there is a set A_i that represents its domain. Thus, the domain of x is given by $X = A_1 \times A_2 \times \cdots \times A_M$, where M is the dimension of x and A_i corresponds to the domain of attribute a_i, with $1 \leq i \leq M$.

When A_i is a discrete and finite set composed of categories, the attribute a_i is referred to as a *categorical attribute*. When A_i is a set of ordered numerical values, the attribute a_i is commonly referred to as a *numerical attribute* or, more precisely, a *continuous attribute* when its domain is an interval or a real-valued range. Note that some attributes may be numerical but still take only a finite set of discrete values (e.g., counts or encoded categories), in which case they are numerical but not continuous.

Consider a set Y, called *label set*, which represents the domain of the output data.

Consider also a set $\mathscr{D}$, composed of N input data, called *training set*. Each element of $\mathscr{D}$ is called a *training instance*.

Supervised machine learning, which is the focus of this thesis, deals with a set $\mathscr{D}$ of the form $\mathscr{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, with $(x_i, y_i) \in X \times Y$, where each y_i is called the *label* of x_i.

Consider also a space of functions H, called *hypothesis space*, which contains functions h, called *hypotheses*, which perform mappings $h : X \to Y$. These hypotheses are formulated and evaluated by the learning algorithm based on the set $\mathscr{D}$. We assume the existence of an ideal and unknown function f that correctly maps X into Y, i.e. $f : X \to Y$, such that $f(x_i) = y_i$ for all $(x_i, y_i) \in \mathscr{D}$, with $1 \leq i \leq N$.

The algorithm's task is to find a function $g \in H$ such that $g \approx f$, i.e. that best approximates the behavior of the ideal function. The challenge is to find a function g that can correctly predict the labels in Y for instances x that do not belong to the training set $\mathscr{D}$.

2.2 Classification

Classification is a subfield of machine learning that uses supervised learning techniques in which the labels of the training data belong to a finite, discrete and predefined domain $Y = \{Y_1, Y_2, \ldots, Y_K\}$, where each Y_i, with $1 \leq i \leq K$, is called a *class*.

The task of a classifier is to predict the class $\hat{Y} \in Y$ of an unlabeled instance x, i.e. $g(x) = \hat{Y}$ (Shalev-Shwartz and Ben-David, 2014). It is common to refer to $\hat{Y}$ as the *target variable* of the classifier.

A classification problem is evaluated on the basis of two main properties: *generalization* and *approximation*. Generalization corresponds to the model's ability to correctly label instances that differ from those used in its construction, i.e., that do not belong to the training set. Approximation refers to the ability to correctly label instances from the training set itself (Abu-Mostafa *et al.*, 2012).

If a model shows good approximation in the training set, but poor generalization – i.e., poor performance when applied to data other than that used for training – it is said that overfitting has occurred. In this case, the model learned has little effective practical applicability. Overfitting can occur in three main situations: when the volume of data in the training set is too small; when the model adjusts excessively to the noise present in the data; or when the training set is not sufficiently representative of the reality of the problem domain (Domingos, 2012).

To estimate the generalization capacity of a model, it is common to divide the training set $\mathscr{D}$, of size N, into two disjoint parts: a subset $\mathscr{D}_{\text{training}}$, used to train the model, and a subset $\mathscr{D}_{\text{test}}$, containing W elements of the form (x_w, y_w). Initially, the model g is built from $\mathscr{D}_{\text{training}}$. Its performance is then evaluated by applying it to the instances $x_w \in \mathscr{D}_{\text{test}}$, in order to calculate a performance metric called *accuracy*, denoted by $A(g)$.

Accuracy is calculated by comparing the true labels y_w with the predicted labels $\hat{y}_w = g(x_w)$, for $1 \leq w \leq W$, so that accuracy estimates the model's ability to generalize. Its formula is given by Abu-Mostafa *et al.* (2012):

$$A(g) = \frac{1}{W} |\{(x, y) \in \mathscr{D}_{\text{test}} \ : \ g(x) = y\}| \tag{2.1}$$

The performance of a classifier can be evaluated using metrics such as *precision*, *coverage* (or *sensitivity*), *F-measure* (*F-Score*) and *AUC* (*Area Under the Curve*), which measure the prediction error considering the classes assigned to the instances (Han *et al.*, 2022).

These metrics are based on the concepts of *true positives*, *true negatives*, *false positives* and *false negatives*, with respect to a given class $Y_k \in Y$. The *true positives* (TP) correspond to the instances correctly assigned to class Y_k by the classifier, while the *false positives* (FP) refer to the instances wrongly assigned to class Y_k, whose labels belong to another class.

Similarly, the *true negatives* (TN) are the instances that do not belong to Y_k and were correctly classified as not belonging to that class. On the other hand, *false negatives* (FN) are the instances that belong to Y_k, but have been classified as belonging to another class. The Table 2.1 summarizes these scenarios.

	$g(x) = Y_k$	$g(x) \neq Y_k$
$f(x) = Y_k$	TP	FN
$f(x) \neq Y_k$	FP	TN

Table 2.1: *Confusion matrix for a class* Y_k.

Based on these concepts, the *precision* metric (P) quantifies the proportion of instances correctly classified as belonging to class Y_k, in relation to the total number of instances classified as such. The *coverage* metric (C) evaluates the effectiveness of the classifier in identifying all the instances that really belong to the Y_k class (Sokolova and Lapalme, 2009). These metrics are calculated using the following expressions:

$$P = \frac{TP}{TP + FP} \qquad \text{e} \qquad C = \frac{TP}{TP + FN} \tag{2.2}$$

The *F1-score* is a harmonic mean between accuracy and coverage, and is defined by:

$$\text{F1-score} = \frac{2PC}{P + C} \tag{2.3}$$

It is important to note that the relevance assigned to each metric may vary depending on the application. Some applications may be more tolerant of errors than others. For example, an error in predicting a small bank loan has different consequences from an error in predicting the diagnosis of a disease in a patient.

In systems deployed and used in the real world, it is essential that the forecasts made by the model are validated. To do this, the forecast data must be properly stored and monitored by those responsible for the system. The definition of what constitutes a correct forecast depends on the specific objectives of each system. In an online store, a correct prediction could be a product recommendation that results in a purchase. In a bank, a successful prediction could be granting credit to a customer who makes their payments

on time. Regardless of the context, it is essential that the forecast data is organized in a way that allows for validation and ensures the metrics accurately reflect the system's actual performance.

2.3 Drifting

This section explains two important types of deviation that can be found in machine learning systems: *data drift* and *concept drift*. *Data drift* refers to the phenomenon in which the distribution of data passing through the system in production deviates from the distribution of data used during model training. This difference between distributions can be detected using statistical methods.

On the other hand, *concept drift* occurs when the relationship between the input attributes and the classes predicted by the model changes (Webb *et al.*, 2016). In other words, attributes that previously determined a certain class now determine another.

In the following sections, these two phenomena will be presented in greater detail.

2.3.1 Data Drift

The field of detecting deviations in data is very present in the daily lives of developers of machine learning systems, since the task of monitoring the behavior of models in production involves continuous monitoring of the systems' input and output data. The data in the $\mathscr{D}$ set, used in the construction and validation stages of a classification model, has specific distributions. From the moment the model is deployed, it is exposed to real-world data instances, whose distributions may eventually diverge from the distributions observed in $\mathscr{D}$.

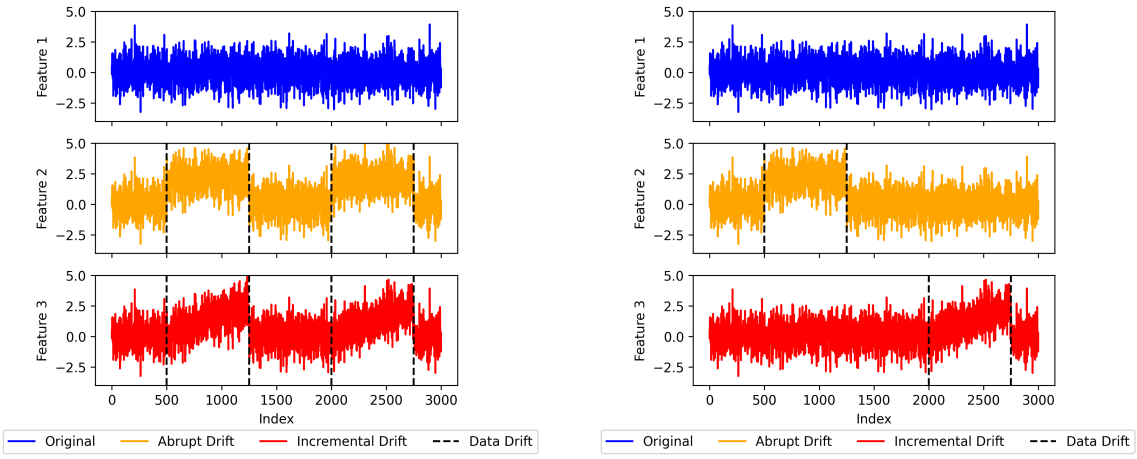
For each attribute $a_i \in \mathscr{D}_{\text{training}}$, it is possible to obtain its probability function $P^{a_i}(x), \forall x \in A_i$, from the relative frequencies of the attribute's values observed in the instances of the training set $\mathscr{D}_{\text{training}}$. In the case of numerical attributes, it is also possible to determine the cumulative distribution function $F^{a_i}(x)$, defined as $F^{a_i}(x) = P(a_i \leq x)$, for all $x \in \mathbb{R}$.

Considering the moment the model is deployed as ground zero on a time scale, the data observed in production within a specific time interval can be represented by the set $\mathscr{R}_{t,u}$, with $t > 0$ and $u \geq t$, where t and u are the start and end instants of the interval, respectively.

Just as the training set $\mathscr{D}_{\text{training}}$ has probability functions $P^{a_i}(x)$ and cumulative distribution functions $F^{a_i}(x)$ for its attributes, the data from $\mathscr{R}_{t,u}$ also allows these functions to be calculated. In other words, given a time interval $[t, u]$, it is possible to estimate the relative frequencies $P^{a_i}_{t,u}(x)$ and cumulative distributions $F^{a_i}_{t,u}(x)$ for the attributes a_i from $\mathscr{R}_{t,u}$.

Thus, for numeric attributes, we define that a data deviation occurs in a_i in the time interval $[t, u]$ when:

$$F^{a_i}(x) \neq F^{a_i}_{t,u}(x) \tag{2.4}$$



(a) *Recurring parallel drift with two drift windows*

(b) *Switching drifts with alternated drift windows*

Figure 2.1: *Illustration of different types of data drift patterns.*

Alternatively, for categorical attributes, it is defined that there is data deviation in a_i in the time interval $[t, u]$ when:

$$P^{a_i}(x) \neq P^{a_i}_{t,u}(x) \tag{2.5}$$

Data drift may manifest according to different temporal and structural patterns. *Abrupt* drift refers to situations in which the data distribution changes suddenly from one state to another, whereas *incremental* drift denotes a gradual transition in which the distribution evolves over time. In settings involving multiple input attributes, drift can also occur in distinct coordination modes. When several attributes experience distributional change simultaneously, the phenomenon is characterized as *parallel* drift; conversely, when different attributes drift at different moments, it is described as *switching* drift. Additionally, drift may reappear cyclically, giving rise to *recurring* drift patterns in which previously observed distributions return after a certain interval. These forms of drift are pertinent to this study, as they are also present in the experimental scenarios used to assess the detectors.

Figure 2.1a illustrates the effect of abrupt and incremental drifts on a data stream of 3,000 instances, where all features initially follow the same normal distribution. Drifts were introduced at 1,500 points and occur in parallel within two distinct drift windows. In contrast, Figure 2.1b presents a switching drift scenario, where changes affect only Feature2 in the first drift window and Feature 3 in the second.

2.3.2 Concept Drift Detection

Concept drift occurs when the statistical properties of a target variable change over time (J. Lu *et al.*, 2018). In machine learning systems, it is crucial to verify whether this phenomenon occurs between the data used to create a model and the data encountered in the production environment. In the era of *big data*, concept drift has emerged as a significant challenge, stemming from uncertainties in data types and the absence of prior knowledge regarding their underlying distributions (Gama and Castillo, 2006).

Consider the training set $\mathscr{D}_{\text{training}}$ and its respective joint distribution $P(X, Y)$, which includes the relative frequencies of the attributes of X – both categorical and numerical – represented by $P^{a_1}, P^{a_2}, \ldots, P^{a_M}$. Taking the model's training moment as the initial instant on a time scale $[0, t]$, concept drift is said to occur if there is an instant t such that:

$$P_{0,t}(X, Y) \neq P_{t+1,u}(X, Y), \quad \forall u > t \tag{2.6}$$

J. Lu *et al.* (2018) conducted a comprehensive literature review on concept drift detection techniques. The paper presents formal definitions of the main concepts related to this phenomenon and compares various algorithms in the field. In addition, the authors propose a generic approach to concept drift detection, structured into four stages:

1. **Data retrieval**: consists of extracting blocks of data from continuous flows. As a single instance may not contain enough information to infer the general distribution, the way in which the blocks are organized becomes fundamental to obtaining significant patterns or knowledge in data stream analysis tasks (Ramírez-Gallego *et al.*, 2017).

2. **Data modeling**: seeks to abstract the retrieved data and extract the main attributes that contain sensitive information, i.e., characteristics whose changes can significantly impact system performance. This stage is optional and is mainly related to reducing dimensionality or sample size, in order to meet storage and speed restrictions in online environments (Liu *et al.*, 2017).

3. **Calculating test statistics**: involves measuring dissimilarity or estimating the distance between data distributions. It quantifies the magnitude of deviation and produces test statistics for the subsequent hypothesis testing stage. This is widely regarded as one of the most challenging aspects of concept drift detection, as defining accurate and robust dissimilarity metrics remains an open research problem. Such metrics are also employed in cluster evaluation (Silva *et al.*, 2013) and in measuring dissimilarity between sample sets (Dries and Rückert, 2009).

4. **Hypothesis testing**: uses specific statistical tests to assess the significance of the change observed in the previous step, for example, by calculating the p-value. This step is essential to give statistical validity to the detection, making it possible to establish a confidence interval that indicates the probability that the change observed is in fact caused by concept drift, and not by noise or bias resulting from the random selection of samples (N. Lu *et al.*, 2014).

Algorithms for Concept Drift Detection Based on Error Rate

Concept drift detection can also be carried out using algorithms that monitor the data passing through the systems in which the learning models are implemented. The error rate-based approach is based on the use of learning algorithms that can adapt to the data's characteristics as it is processed in real-time.

When a significant variation in the error rate — either an increase or decrease – is detected with statistical significance, a deviation alarm is triggered, indicating the need to update the model. The most frequently referenced algorithms in the literature for this

approach include the *Drift Detection Method* (DDM) (Gama, Medas, *et al.*, 2004), the *Early Drift Detection Method* (EDDM) (Baena-Garcia *et al.*, 2006), and the *Adaptive Windowing* (ADWIN) (Bifet and Gavalda, 2007).

2.3.3 Data Drift Detection

To detect data drifts, it is necessary to monitor the extent of the divergence between the distributions of the sets. In this work, we focus on two categories of drift detection techniques that are commonly used in practice: *statistical* and *distance-based* methods. The following subsections present a selection of techniques from these two categories, which will be employed in the experimental analysis conducted later in this work.

Statistical Methods

Statistical methods typically use a formal statistical hypothesis test to compare the distributions of two datasets (e.g., current data vs. reference data). They generate a statistical measure (e.g., p-value) indicating the likelihood that the two distributions are the same. A low p-value suggests that the distributions are significantly different, indicating drift. They can be sensitive to sample size and assumptions about the underlying distributions. When data scarcity is not a concern, nonparametric methods (which do not assume any specific data distribution) are more often used (Dasu *et al.*, 2006). Common nonparametric tests include the Wilcoxon, Kolmogorov-Smirnov, and multinomial tests.

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (KS) statistical test can be used to test whether two samples came from the same distribution. Assuming F and G are the empirical distribution functions of the two samples and the sample sizes are n and m, respectively, the Kolmogorov–Smirnov statistic $KS(F, G)$ is defined as follows (Hodges Jr, 1958):

$$KS(F, G) = \sup_x |F(x) - G(x)| \tag{2.7}$$

The decision rule is to reject the hypothesis at the significance level α if $KS(F, G) > c(\alpha)\sqrt{\frac{n+m}{n \times m}}$, where $c(\alpha)$ is given in the Kolmogorov table (Hodges Jr, 1958).

Multiple Univariate Kolmogorov-Smirnov Test

While the univariate KS Test assesses the difference between the empirical distribution of two datasets in one dimension, the Multiple Univariate KS Test does this across multiple dimensions simultaneously. However, testing multiple hypotheses increases the probability of observing a rare event, thereby increasing the likelihood of incorrectly rejecting a null hypothesis. According to Rabanser *et al.* (2019), a conservative aggregation method can be used to reduce the probability of this type of error, the Bonferroni correction (Bland and Altman, 1995), to adjust the significance level for multiple comparisons.

Applying the Bonferroni correction to the Multiple Univariate KS Test for d distributions, the decision rule is to reject the hypothesis at the significance level α if $\min_{k=1,2,\ldots,d} KS(F_k, G_k) > c(\frac{\alpha}{d})\sqrt{\frac{n+m}{n\times m}}$, where $KS(F_k, G_k)$ is the KS Test for the empirical distribution functions F and G of the k-th dimension, whose respective sample sizes are n and m. Note that Bonferroni correction is applied by testing the hypothesis at a significance level of α/d.

Distance-Based Methods

Distance-based methods offer a more direct measure of the distance or dissimilarity between two probability distributions than statistical methods. Moreover, they do not rely on specific distributional assumptions. They provide a numerical value indicating the degree of difference: the larger the distance, the greater the difference, indicating drift.

Kullback–Leibler Divergence

For two discrete probability distributions P and Q, the Kullback–Leibler (KL) Divergence (or relative entropy) $KL(P||Q)$ from Q to P is defined as (Dasu *et al.*, 2006):

$$KL(P||Q) = \sum_{x\in\mathcal{X}} P(x)log(\frac{P(x)}{Q(x)}) \tag{2.8}$$

Jensen–Shannon Divergence

The Jensen–Shannon (JS) Divergence is a method based on the KL Divergence, but with an improvement: it is symmetric. For two discrete probability distributions P and Q, the Jensen–Shannon Divergence $JS(P||Q)$ is defined as:

$$JS(P||Q) = \frac{1}{2}KL(P||\frac{(P+Q)}{2}) + \frac{1}{2}KL(Q||\frac{(P+Q)}{2}) \tag{2.9}$$

Hellinger Distance

For two discrete probability distributions $P(p_1, p_2, \ldots, p_n)$ and $Q(q_1, q_2, \ldots, q_n)$, the Hellinger distance $H(P, Q)$ is defined as:

$$H(P,Q) = \frac{1}{\sqrt{2}}\sqrt{\sum_{i=1}^{n}(\sqrt{p_i} - \sqrt{q_i})^2} \tag{2.10}$$

Data Drift Detection Methods

Choosing a distance function to measure change is only one aspect of the drift detection problem. Another crucial aspect is determining the statistical significance of the observed

change (Dasu *et al.*, 2006). This involves specifying a null hypothesis (i.e., that no change has occurred) and assessing how likely it is that the observed measurement could occur under this hypothesis. Some drift detection approaches, such as HDDDM, measure changes in a distance metric between the incoming data distribution and a baseline distribution, which may be updated either when drift is detected or continuously. This way, these methods can be used to detect changes in data distribution over time, which can help maintain the performance of machine learning models in dynamic environments.

The following sections define some methods that use this approach and are evaluated in this work. The first one, HDDDM, was presented by Ditzler and Polikar (2011). The second and third methods, JSDDM and KSDDM, are the new methods proposed in this work. All of these approaches assume an incremental learning setting, where new datasets are presented in batches over time; they are not based on the classifier's performance (i.e., they rely only on raw features); and they are classifier-free.

It is worth noting that, in terms of computational cost, these techniques are comparable. Each method derives empirical distributions from the same input data. The computation of drift is linear with respect to the number of bins, which is dictated by the batch size.

Hellinger Distance Drift Detection Method (HDDDM)

Ditzler and Polikar (2011) proposed a drift detection algorithm that operates solely on raw data features to identify drift in a supervised incremental learning setting, using the Hellinger distance between successive training data batches and an adaptive threshold.

The Hellinger Distance-based Drift Detection Method (HDDDM) proposed by Ditzler and Polikar (2011) assumes that data arrives in batches, with dataset $\mathscr{D}_t$ becoming available at time t. The following steps summarize the method, assuming as initialization $\lambda = 1$, $\mathscr{D}_\lambda = \mathscr{D}_1$, and for $t = 2, 3, \ldots$:

1. Generate histograms P from $\mathscr{D}_t$ and Q from $\mathscr{D}_\lambda$, each one with $b = \lfloor\sqrt{|\mathscr{D}_t|}\rfloor$ bins.
2. Calculate the Hellinger distance $\delta_H(t)$ between P and Q and its difference from $\delta_H(t-1)$:

$$\delta_H(t) = \frac{1}{d}\sum_{k=1}^{d}\sqrt{\sum_{i=1}^{b}\left(\frac{P_{i,k}}{\sum_{j=1}^{b}P_{j,k}} - \frac{Q_{i,k}}{\sum_{j=1}^{b}Q_{j,k}}\right)^2} \qquad (2.11)$$
$$\epsilon(t) = \delta_H(t) - \delta_H(t-1)$$

   where d is the dimensionality of the data, and $P_{i,k}$($Q_{i,k}$) is the frequency count in bin i of the histogram corresponding to P (Q) of feature k.

3. Update the adaptative threshold $\hat{\epsilon}$ and the standard deviation $\hat{\rho}$:

$$\hat{\epsilon} = \frac{1}{t-\lambda-1}\sum_{i=\lambda}^{t-1}|\epsilon(i)| \qquad \hat{\rho} = \sqrt{\frac{\sum_{i=\lambda}^{t-1}(|\epsilon(i)| - \hat{\epsilon})^2}{t-\lambda-1}} \qquad (2.12)$$

4. Compute the actual threshold $\beta(t) = \hat{\epsilon} + \gamma\hat{\rho}$, where γ is some positive constant, indicating how many standard deviations of change around the mean indicate drift.

5. Determine if drift occurred:

   Drift is present if $|\epsilon(t)| > \beta(t)$. In this case, it is necessary reset $\mathscr{D}_\lambda$ by making $\mathscr{D}_\lambda = \mathscr{D}_t$ and $\lambda = t$.
   When drift is not present, $\mathscr{D}_\lambda$ must be expanded to include $\mathscr{D}_t$: $\mathscr{D}_\lambda = \{\mathscr{D}_\lambda, \mathscr{D}_t\}$.

HDDDM signalizes a drift when the magnitude of the change ($|\epsilon(t)|$) is significantly greater than the average of the change since the last detected change ($\hat{\epsilon}$). The significance is controlled by γ and the standard deviation of the divergence differences ($\hat{\rho}$).

Jensen-Shannon Drift Detection Method (JSDDM)

To create a drift detection method based on the JS Divergence, we adapted the HDDDM method by replacing the Hellinger Distance in Equation 2.11 in Step 2 of the method with the JS Divergence defined in Equation 2.9, as shown in the following:

$$\delta_{JS}(t) = \frac{1}{d}\sum_{k=1}^{d} JS(P_k||Q_k) \qquad \epsilon(t) = \delta_{JS}(t) - \delta_{JS}(t-1) \tag{2.13}$$

where d is the data dimensionality, and $P_k(\ Q_k)$ is the distribution histogram of feature k.

Kolmogorov-Smirnov Drift Detection Method (KSDDM)

In order to detect data drifts using the Kolmogorov-Smirnov (KS) Test, we devised the following simple algorithm, which assumes that data arrives in batches, with dataset $\mathscr{D}_t$ becoming available at time t, $\lambda = 1$, $\mathscr{D}_\lambda = \mathscr{D}_1$, and for $t = 2, 3, \ldots$:

1. Generate the empirical distribution functions F_k from $\mathscr{D}_t$ and G_k from $\mathscr{D}_\lambda$ for each feature k in data.

2. Obtain the minimum of the KS Test of all features' empirical distributions:

   $$\delta_{KS}(t) = \min_{k=1,2,\ldots,d}\{KS(F_k, G_k)\} \tag{2.14}$$

   where F_k (G_k) is the empirical distribution function of feature k in $\mathscr{D}_t$ ($\mathscr{D}_\lambda$).

3. Determine if drift occurred:

   Drift is present at the significance level α if $\delta_{KS}(t)$ is greater than $c(\frac{\alpha}{d})\sqrt{\frac{n+m}{n\times m}}$, where $m = |\mathscr{D}_t|$, $n = |\mathscr{D}_\lambda|$, d is the dimensionality of the data and $c(\frac{\alpha}{d})$ is given in the Kolmogorov table.

   If drift is detected, it is necessary to reset $\mathscr{D}_\lambda$ by making $\mathscr{D}_\lambda = \mathscr{D}_t$ and $\lambda = t$.
   When drift is not present, $\mathscr{D}_\lambda$ must be expanded to include $\mathscr{D}_t$: $\mathscr{D}_\lambda = \{\mathscr{D}_\lambda, \mathscr{D}_t\}$.

2.4 Machine Learning System

A machine learning system can be understood as a software system that incorporates a machine learning model in one of its components. As discussed in Section 2.1, using an ML model in a system requires carrying out a series of procedures, which will be described in more detail in Section 2.4.4.

The taxonomy presented in Figure 2.2 was developed in this master's research project with the aim of providing a comprehensive overview of the main concepts involved in projects that integrate machine learning systems. The three major conceptual divisions were structured considering the areas of specialization of the professionals who commonly participate in these projects: data scientists, ML engineers, data engineers, and those responsible for data governance.

This taxonomy was inspired by the structure proposed by Schröder and Schulz (2022), who focused on monitoring approaches for ML systems. In addition, the choice to organize the concepts according to stakeholder roles draws from the work of Muccini and Vaidhyanathan (2021), where a similar classification is used to align concerns with the responsibilities of different actors in software-intensive ML projects. Taxonomies like this are useful for visually and conceptually organizing complex domains, providing a clearer understanding of how technical and organizational elements interrelate.

The following sections will discuss the concepts of this taxonomy that are most related to the objectives of this master's thesis, as well as some aspects that cut across these categories.

2.4.1 Trustworthy Machine Learning

The area of data governance has become increasingly relevant to machine learning (ML) projects, especially due to the growth of regulations and guidelines aimed at protecting user data, such as the General Data Protection Law (LGPD, from *Lei Geral de Proteção de Dados* in Portuguese) in Brazil – which is similar to GDPR (General Data Protection Regulation). Alongside data protection, another field that is gaining prominence is trustworthy machine learning.

Trustworthy machine learning involves guidelines proposed by regulatory bodies, aiming to guide the development of more ethical and responsible ML applications. According to the European Commission for Artificial Intelligence (*Ethics guidelines for trustworthy AI* 2026), the fundamental pillars for a trustworthy ML system are: legality, ethics, and robustness. To be considered legal, the system must comply with all applicable laws and regulations. To be ethical, the system must respect fundamental principles and values, such as diversity, non-discrimination, and justice. Finally, a robust system must be resilient to malicious attacks in order to guarantee its integrity, availability, and confidentiality (Xiong *et al.*, 2022).

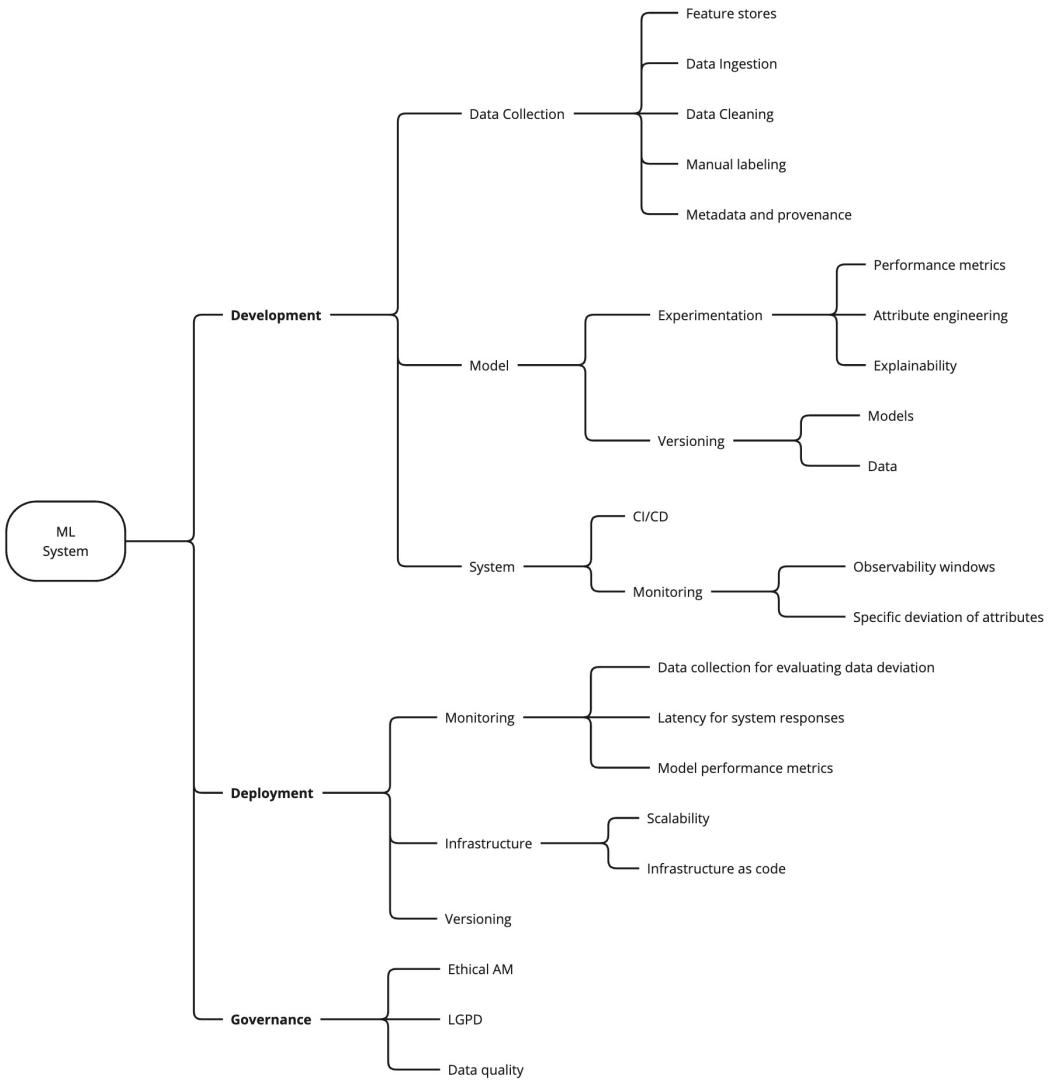


Figure 2.2: *Taxonomy of concepts related to ML projects.*

2.4.2 Explainability

The area of explainability has attracted considerable attention from the scientific community, since the ability to interpret the predictions of certain models is essential for their adoption in practical contexts. By offering justifications for the decisions made by models, explainability techniques aim to provide greater transparency to users, thereby increasing trust in machine learning-based systems (Weller, 2019).

2.4.3 Requirements for a Machine Learning System

Just like traditional software systems, machine learning systems also have requirements to meet. In addition to the functional requirements common to software systems, machine learning systems are expected to meet the fundamental pillars presented in Section 2.4.1.

Such systems must have qualities such as good accuracy and coverage, as well as being fair, transparent, secure, respectful of privacy, and stable and reliable (Horkoff, 2019). Many of these aspects guide research in the area.

Regardless of their criticality, machine learning systems must have *a priori* definitions of the characteristics of their data (Kuwajima *et al.*, 2020). By carefully defining the test distributions, it becomes possible to monitor the system's data in production in order to detect any deviations that may indicate flaws in the system's expected behavior. The ability to detect variations in the data makes the system more robust and consistent, allowing it to be maintained more assertively whenever necessary.

2.4.4 Lifecycle of a Machine Learning System

In the taxonomy represented in Figure 2.2, the development area covers both experimentation in machine learning and the development of the software system. With regard to experimentation, good practices should be considered, such as versioning test data and recording the performance metrics of the models obtained. In general, the successful development of a machine learning system involves the following steps:

1. **Data preparation**: in this step, the $\mathscr{D}$ data set to be used to obtain the model is defined. This data will also be used to assess whether, over time, the production environment is still adequately represented in the training set. After defining the data set, the attributes $(a_1, a_2, \ldots, a_n)$ of the $\mathscr{D}$ instances are refined and, if the representation of the domain proves to be insufficient to obtain a model with satisfactory performance, new attributes can be created and incorporated.

2. **Model training and validation**: in this step, the model must be trained and validated using the disjoint subsets of $\mathscr{D}$, namely $\mathscr{D}_{\text{training}}$ and $\mathscr{D}_{\text{test}}$. The model's performance on the test set must be evaluated based on the metrics defined as relevant to the project. If the results are not satisfactory, it may be necessary to create new attributes that add more knowledge about the domain or expand the training set. The reference performance measures (*baseline*) should be stored so that the stability of the model can be checked over time.

3. **Implementation of the model in production**: this stage consists of making the model available in an operational software system, able to respond to data from the real world. It is essential that the data used for prediction is stored so that the model's performance can be continuously evaluated. Deployment can be carried out using specific tools, which will be discussed in Section 3.2.

4. **System data monitoring**: this stage involves monitoring the model's input data in production, with the aim of identifying potential deviations. To carry out this monitoring, you can use the strategies presented in Sections 2.3.1 and 2.3.2. If the new data differs significantly from the original data, it may be necessary to develop a new model more suited to the new profile observed.

5. **Model monitoring**: in this step, the performance metrics of the model in production are observed, based on the stored prediction data. Significant deviations from the metrics obtained during training may indicate the occurrence of concept drift. In

such cases, the need to retrain the model should be assessed.

2.5 MLOps

MLOps (short for *Machine Learning Operations*) is a paradigm that encompasses best practices, sets of concepts, and a development culture aimed at the description, implementation, deployment, monitoring, and scalability of machine learning systems (Kreuzberger *et al.*, 2023). This paradigm aims to integrate the areas of machine learning, software development (especially the principles of *DevOps*) and data engineering, with the aim of facilitating the implementation of machine learning systems by reducing the distance between development (*Dev*) and operations (*Ops*) – in other words, between the creation of the model and its actual operation in a production environment.

Essentially, MLOps seeks to enable the construction of robust machine learning systems through principles such as continuous integration; orchestration of workflows; reproducibility; versioning of data, models, and code; collaboration between teams; continuous training and validation of models; tracking and logging of models and their metadata; and continuous monitoring and iterative feedback cycles.

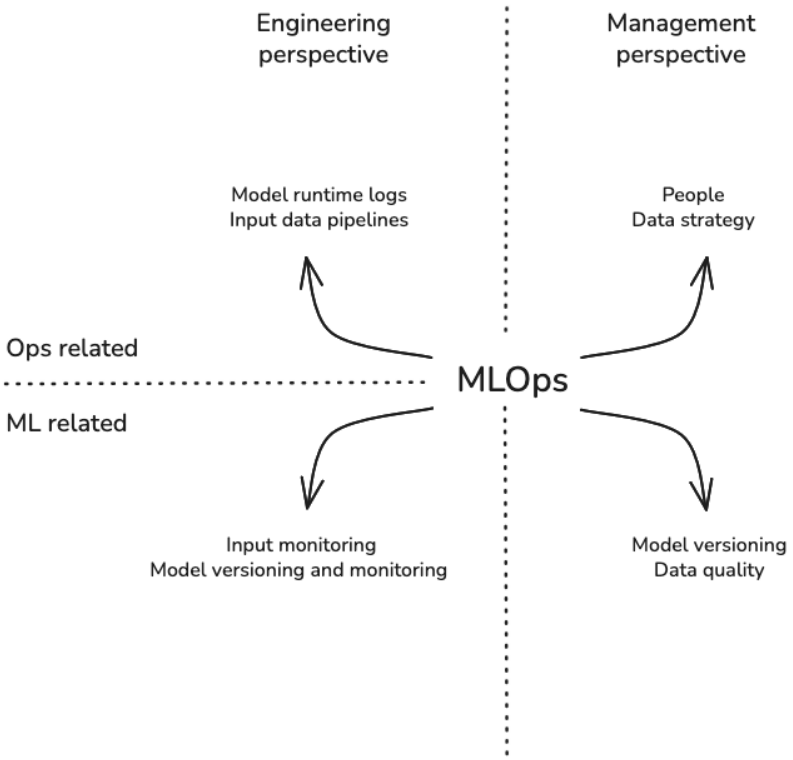


Figure 2.3: *MLOps overall conceptual map.*

Figure 2.3 summarizes the main dimensions of MLOps by organizing its concerns into four quadrants, following a structure similar to the socio-technical perspective proposed by Leite *et al.* (2023). The figure highlights how MLOps integrates practices from software engineering, machine learning, operations, and organizational management to support the full lifecycle of ML systems.

From the *engineering perspective* (upper-left), the emphasis is on the software and data infrastructure required for reliable operation, such as data pipelines and model runtime logging. The *ML perspective* (lower-left) focuses on model-centric practices including input monitoring, model versioning, and mechanisms to detect and respond to degradation.

On the *management perspective* (upper-right), MLOps involves coordinating people, defining a data strategy, and aligning technical processes with organizational goals. The *quality and governance perspective* (lower-right) encompasses data quality and model governance, addressing the need for responsible deployment, reproducibility, and auditability.

Together, these quadrants illustrate that MLOps extends beyond tooling or automation, functioning instead as an interdisciplinary framework that connects technical, organizational, and operational practices to ensure that ML systems remain reliable, maintainable, and aligned with business objectives.

2.6 MLaaS

Mauro Ribeiro *et al.* (2016) coined the term *machine learning as a service* (MLaaS) in 2015, when they developed a scalable and flexible machine learning platform based on a service component architecture. The aim of this platform was to facilitate the creation, validation, and execution of machine learning models.

Infrastructure as a service (IaaS) platforms, which have been consolidated over the last decade, have also started to incorporate tools aimed at the field of machine learning. Among the main infrastructure platforms, Google has been offering *Vertex AI* (2026) since 2016, Amazon has been offering *Sagemaker* (2026) since 2017, and Microsoft has been providing *Azure Machine Learning* (2026) since 2015.

2.7 AutoML

The growing demand for machine learning solutions has motivated researchers to investigate automated machine learning, commonly referred to as AutoML. AutoML seeks to automate the application of machine learning techniques to real-world problems, encompassing what has come to be known as the "end-to-end machine learning process" (Karmaker Santu *et al.*, 2021).

AutoML processes involve automating the experimentation and deployment stages of machine learning models, aiming to democratize their use among non-expert users. To this end, artificial intelligence techniques are employed to facilitate the various phases required for model building, thereby reducing dependence on in-depth technical knowledge.

Tools such as MLJar (Płońska and Płoński, 2021) are capable of automatically generating models from labeled tabular data, performing tasks such as attribute engineering and adjusting hyperparameters, and also providing information to help explain and interpret the generated models.

The AutoML area has also been widely explored by infrastructure as a service (IaaS) platforms, making them particularly attractive to companies and projects that lack the

interest or resources to set up specialized machine learning teams. The main platforms include: Google, with *Cloud AutoML* (2026); Amazon, with *SageMaker Autopilot* (2026); and Microsoft, with *Azure Machine Learning* (2026).

2.8 Chapter Remarks

The concepts presented in this chapter establish the necessary groundwork for understanding the broader context in which machine learning systems operate. While we have covered key notions such as classification, data and concept drift, and the lifecycle of ML systems, it is essential to emphasize that the central focus of this work lies specifically in monitoring inputs within these systems.

Input monitoring plays a crucial role in identifying distributional changes that may compromise model performance, particularly in production environments. The techniques and terminologies introduced here will serve as the basis for evaluating strategies aimed at detecting such shifts and ensuring the reliability of machine learning systems over time.

Chapter 3

Related Work

This chapter presents research related to the central theme of this thesis—detecting drift in machine learning (ML) systems – with emphasis on studies in Software Engineering for Machine Learning (SE4ML) and the tools that support the lifecycle of ML-based applications. The objective is to situate the architectural and methodological contributions of this work within the broader scientific and industrial landscape, highlighting both consolidated research findings and areas where practice continues to evolve.

Literature Search Strategy

The identification and selection of relevant literature followed a multistage process combining academic search, snowballing, and the incorporation of grey literature. This structure was necessary because SE4ML and MLOps are relatively young research areas in which many impactful contributions originate in industrial practice and only later appear in peer-reviewed venues. This motivated the adoption of a *multivocal literature review protocol*, suitable for capturing knowledge distributed across both academic and industrial sources (Kamei *et al.*, 2021).

First, foundational academic studies were located through searches on IEEE Xplore, ACM Digital Library, and Google Scholar using terms such as "software engineering for machine learning," "MLOps," "technical debt in ML," "machine learning system architecture," and related expressions such as *"MLOps tools," "machine learning experimentation," "feature store," "model deployment pipeline," "model monitoring," "data versioning," "ML system architecture,"* and *"feedback loops in ML"*. These search terms were refined iteratively throughout the review process as new tools, methodologies, and taxonomies emerged over the past several years.

Subsequently, backward and forward snowballing were employed to expand the review's coverage. This revealed an important characteristic of the field: as observed in studies such as Serban and Visser (2022), Washizaki *et al.* (2019), and Nascimento *et al.* (2020), a large portion of academic work on SE4ML has been published in conferences or workshops, particularly after 2016. Although the broader field of AI encompasses a substantial volume of recent journal publications, relatively few focus specifically on the

intersection of machine learning engineering and software architecture – precisely the domain addressed in this thesis. Conferences, therefore, serve as the primary venue for the rapid dissemination of architectural, infrastructural, and lifecycle-oriented contributions.

To complement these academic sources and capture up-to-date industry practice, the review also incorporated grey literature, consistent with established recommendations for multivocal reviews in rapidly evolving fields Kamei *et al.*, 2021. Grey literature sources included engineering blogs from major technology companies (e.g., Google, Uber, Netflix), technical articles on platforms such as Medium and Towards Data Science, documentation from open-source MLOps frameworks (e.g., MLflow, Kubeflow, Seldon), and practitioner-oriented reports from organizations such as Microsoft and McKinsey. These sources were selected because they routinely provide early descriptions of tools, architectures, and operational challenges that only later appear in academic venues.

The selection of all sources—academic and grey literature—was guided by clear inclusion and exclusion criteria. Sources were included when they (i) described tools, methodologies, or architectural components relevant to experimentation, deployment, monitoring, or feedback loops in ML systems; (ii) provided enough technical detail to understand their purpose and operation; and (iii) demonstrated either academic relevance (e.g., reputable venue, citation impact) or community validation (e.g., widespread adoption, active maintenance, or robust documentation). Promotional material, opinion pieces without substantive technical content, or documentation of deprecated tools were excluded.

The combination of peer-reviewed and practitioner-oriented literature ensures that the survey presented in this chapter reflects both the conceptual foundations of ML system design and the practical realities faced by organizations deploying ML in production. This broad evidence base also strengthens the architectural proposal developed later in this thesis, grounding it in a representative and comprehensive body of knowledge.

3.1 Software Engineering for Machine Learning

Machine learning systems differ from traditional software in that their behavior depends not only on code but also on data and models, which evolve over time. This introduces challenges related to testing, reproducibility, deployment, monitoring, and architecture that require adaptations of established software engineering practices. In recent years, both industry and academia have proposed frameworks, taxonomies, and tooling to better support the development, delivery, and operation of ML-based systems.

The following subsections summarize these contributions, covering engineering challenges, quality assurance, empirical studies of best practices, MLOps and architectural perspectives, and recent advances in explainability.

3.1.1 Challenges in Developing and Maintaining Machine Learning Systems

Machine learning applications are increasingly consolidated in the industry. However, the process of developing, deploying, and continuously improving them is more complex

than the development of traditional software, such as web services or mobile applications. Machine learning systems are subject to changes in three main components: the code, the model, and the data. Their behavior is often complex and difficult to predict, making them harder to test, explain, and improve.

Some professionals who work with machine learning have no formal training in software development and are more familiar with experimentation and model validation practices. To ensure that systems are more reliable, it is essential that their maintainers have knowledge of software engineering methodologies. Additionally, many traditional software development techniques, such as testing and quality control practices, are not directly applicable to machine learning systems.

In 2015, Sculley *et al.* (2015) highlighted the problems and technical debts faced in developing and maintaining machine learning systems within *Google*, noting that traditional software engineering best practices do not address all the challenges of these systems. The expected behavior of machine learning systems cannot be fully expressed in terms of programming logic, as it relies heavily on external data, and real-world behavior is difficult to encapsulate. The authors highlighted recurring problems caused by changes in data structure, underscoring the need for stability and versioning of data profiles used in model pipelines. Another relevant challenge was the scarcity of tools for static analysis of data dependencies, unlike in traditional systems, where compilers and build systems support such analysis. The configuration of machine learning systems also proved to be a recurring source of problems, as large systems include several configurable options, such as attribute selection, data sampling strategies, learning algorithm configurations, and pre- and post-processing steps. The authors additionally identified technical debts in areas such as data testing, reproducibility of experiments, project management, and even cultural alignment within teams, all of which compromise the long-term sustainability of ML systems.

3.1.2 Testing, Quality Control, and Continuous Delivery in ML

Years later, Breck *et al.* (2017) proposed strategies focused on testing to improve the development of machine learning systems, based on practical experiences at *Google*. In this paper, the authors highlight the main components that should be tested: data, models, and infrastructure. They also discuss how continuous monitoring of these tests should be carried out. Specifically, they advocate operationalizing a "test score" rubric into ongoing, automated checks that run in production to detect data pipeline instability (e.g., schema changes, missing or shifted features), training–serving skew, and distribution drift as data and user behavior evolve over time. Breck *et al.* (2017) further recommend tracking live model quality signals – such as performance, calibration, and slice-level metrics – alongside system health indicators (latency, throughput, resource usage), with dashboards and alerting wired into team workflows so— degradations and regressions are surfaced early and attributable to data, model, or infrastructure causes. The article also emphasizes the importance of disseminating this knowledge within internal teams and adopting suitable frameworks at each stage of system development.

Zhang *et al.* (2022) synthesized findings from 138 different studies to establish a framework for machine learning (ML) testing. A primary distinction they identify be-

tween conventional software and ML-based systems is the dynamic influence of incoming data on the latter's logic and behavior. Within this framework, a *bug* is conceptualized as any discrepancy between a system's observed performance and its intended functional requirements. Consequently, the authors define testing as any investigative process aimed at identifying these inconsistencies. Their analysis further reveals a significant imbalance in current literature; while testing methodologies for supervised learning are well-documented, there remains a notable lack of research concerning reinforcement and unsupervised learning paradigms.

In 2019, Sato *et al.* (2019) adapted the principles and practices of traditional continuous delivery to the context of machine learning systems. Their work demonstrates how to achieve better risk management when implementing changes to ML-based applications, using a sales forecasting system as a case study. To make the process more reproducible and auditable, the authors employed tools such as *MLflow* (Chen *et al.*, 2020) and *Data Version Control* (2026) for experiment tracking and data versioning. System monitoring was performed using *Elasticsearch* (2026), while continuous delivery was supported by *GoCD* (2026).

3.1.3 Empirical Studies and Taxonomies of Best Practices

Amershi *et al.* (2019) conducted an internal study at Microsoft to identify the main challenges and best practices adopted across different teams when developing machine learning systems. The study revealed a wide range of challenges throughout the ML lifecycle. Based on the experiences and obstacles identified, the authors proposed a model to assess the technical maturity of teams.

Serban, Blom, *et al.* (2020) carried out an empirical study on the ways in which teams build, operate, and sustain machine learning–based software systems. Their work included a critical examination of both academic and grey literature, noting that many industry practices are not well documented in scholarly research. From this analysis, the authors developed a taxonomy of software engineering practices tailored to ML projects. These practices were organized according to different stages of the ML lifecycle, including data-related activities, model training, deployment, programming, staffing, and governance. In addition, the study differentiated between practices that are novel to ML, those inherited from traditional software engineering, and those adapted from conventional approaches.

To evaluate how widely these practices are adopted, the authors designed a questionnaire that was answered by 313 ML practitioners from diverse professional backgrounds and regions. The results enabled comparisons across different team sizes and organizational contexts. The findings indicated that several of the most relevant practices were associated with the training phase, while others focused on data preparation. Key training-related practices involved defining measurable objectives, aligning goals across teams, maintaining version control for data and models, and monitoring model performance over time. In the data preparation stage, reusable scripts for data cleaning and integration were considered particularly important. The study also highlighted the use of continuous integration in programming, ongoing monitoring of deployed models, and the enforcement of fairness and privacy requirements as central governance practices for responsible ML use.

Following this work, Serban, Blom, *et al.* (2021) proposed a new taxonomy for developing reliable ML systems, aligning the practices with requirements defined by the European Union commission for trustworthy ML (*Ethics guidelines for trustworthy AI* 2026). Recommended practices include: testing for social bias in training data; preventing the use of discriminatory attributes as model inputs; adopting privacy-preserving ML techniques; using interpretable models when possible; and assessing or mitigating subgroup-related biases.

Alves *et al.* (2023) developed broader taxonomies aimed at ML product development, covering topics such as problem and solution definition, ML product management, data and model management, software engineering, and model delivery.

3.1.4 Architecture and Continuous Monitoring (MLOps)

In a more recent work, Serban and Visser (2022) investigated how systems can be architected to enable robust integration of machine learning components. The authors identified major architectural challenges and organized them into a comprehensive taxonomy aligned with prior studies. They also proposed directions to address each challenge. The largest category, *Design*, includes classic SE challenges (e.g., managing development components) and ML-specific challenges such as managing model uncertainty and designing systems that support automatic adaptation, as in AutoML approaches. The taxonomy proposed in this master's thesis, presented in Figure 2.2, focuses on technical processes in ML system development.

Kreuzberger *et al.* (2023) conducted a mixed-methods study that organized MLOps into four main dimensions—architecture, roles, components, and principles—leading to the proposal of a holistic definition of the field. In a related line of research, Muccini and Vaidhyanathan (2021) examined the software architecture of machine learning–based systems and identified four critical areas that require further attention, along with directions for future standardization. Furthermore, Nazir *et al.* (2024) presented a comprehensive analysis of ML-enabled system architectures, reporting 27 key decisions, 35 design challenges, and 42 best practices based on a systematic literature review and a questionnaire. While academic and industrial perspectives largely converge, important differences remain.

Schröder and Schulz (2022) emphasized the inherent fragility of ML systems compared to traditional software, identifying six major monitoring challenges. Key issues include silent model degradation, evolving data environments (e.g., concept drift), and complexities arising from high-dimensional inputs. The study highlights how failures often result from distribution shifts or adversarial data, reinforcing the need for continuous monitoring of model performance, data quality, and system infrastructure. The authors reference the CACE principle ("changing anything changes everything"), advocating for monitoring framed as continuous testing in operation. This supports upstream validation and monitoring as proactive strategies to mitigate technical debt and ensure long-term reliability.

3.1.5 Model Explainability (XAI)

Explainable Artificial Intelligence (XAI) aims to make the behavior of machine learning models comprehensible to humans, allowing stakeholders to inspect, question, and ultimately trust automated decisions. Rather than treating models as opaque black boxes, XAI techniques aim to reveal how inputs are transformed into outputs, what patterns the model relies on, and under which conditions its predictions may fail. In this subsection, the focus is on post-hoc approaches that generate explanations from trained models, laying the groundwork for the specific techniques and empirical studies discussed next.

M. T. Ribeiro *et al.* (2016) proposed a modular and extensible approach to interpretably explain the predictions made by machine learning models, and developed an open-source library to support this approach. Later, Lundberg and Lee (2017) extended this line of research using concepts from game theory, proposing *SHAP* (*SHapley Additive exPlanations*) to interpret predictions based on feature contributions. The implementation is publicly available on GitHub (*SHapley Additive exPlanations* 2026).

Recent work by Mersha *et al.* (2024) presents a comprehensive survey of explainable artificial intelligence (XAI), organizing the field according to stakeholder needs, available techniques, and open research challenges. The authors propose a taxonomy that classifies explanation methods based on their scope (local or global), stage of intervention (intrinsic or post hoc), and intended function. Their analysis highlights both the diversity of existing approaches and persistent gaps related to evaluation, standardization, and theoretical foundations, emphasizing the importance of explainability for trustworthy machine learning systems.

Bhatt *et al.* (2020) investigated how organizations apply explainability techniques in ML systems deployed in production, reporting practical outcomes and identifying future research directions. Although adoption has been increasing, the authors observed limitations that hinder the exposure of explainability to end users, especially non-technical ones. These limitations remain open research challenges.

3.2 Tools for Developing a Machine Learning System

During the model experimentation and software development stages, practitioners can leverage a variety of tools designed to support the specific requirements of machine learning systems. The following sections present a selection of such tools, identified through a multivocal literature review that includes both academic publications and posts from widely recognized blogs within the ML development community.

3.2.1 Platforms

Machine learning platforms support multiple stages in the lifecycle of ML systems, ranging from experimentation to deployment and monitoring. As discussed in Section 2.6, some of these platforms are offered as a service by large infrastructure providers, making it significantly easier to develop these systems.

On these platforms, developers can upload their data and then use pre-trained models

to make predictions or train new models with the data provided. The resulting models are available for use via APIs, and their predictions can be stored directly on the platforms. These models and their predictions can then be integrated into software systems to solve specific problems in different application contexts.

Among the platforms hosted by the companies that make them available are Amazon *Sagemaker* (2026), launched in November 2017, *Vertex AI* (2026), launched in 2019 under the name *AI Platform*, and *Azure Machine Learning* (2026), launched in 2015. All of these platforms offer functionalities such as detecting deviations in data, AutoML tools, and creating *endpoints* for prediction with the trained models.

In addition to the platforms offered by large infrastructure providers, some companies have developed open source tools aimed at managing the lifecycle of machine learning systems, either end-to-end or focusing on specific stages. Examples include *Metaflow* (2026), *MLflow* (Chen *et al.*, 2020), and *Kubeflow* (2026).

Metaflow is a Python library developed by Netflix to help data scientists and engineers develop and manage data science projects. *MLflow* is an open source platform created by Databricks that supports the experimentation, reproducibility, deployment, and model management stages. *Kubeflow*, created by Google, aims to facilitate the deployment of machine learning workflows in environments based on *Kubernetes*, promoting portability and scalability.

Table 3.1 presents some of the main platforms that support the lifecycle of machine learning systems in production, comparing them on the basis of the stages covered throughout this cycle. The *Open* column indicates whether the tool is open source, as well as the number of stars the project has on *GitHub*, a metric that reflects the level of community interest in the project.

Name	Experimentation	Implementation	Monitoring	Explainability	Open
MLflow	X	X			X (23.1k)
Kubeflow	X	X	X	X	X (15.3k)
Metaflow	X	X			X (9.6k)
Vertex AI	X	X	X	X	–
SageMaker	X	X	X	X	–
Azure ML	X	X	X		–
Seldon		X	X	X	X (4.7k)

Table 3.1: *Tools and their support in the stages of the machine learning lifecycle*

Table 3.1 shows that two of the three tools that support all stages of the machine learning model lifecycle are closed-source solutions developed by major technology companies. The only open-source alternative that covers the entire lifecycle is Kubeflow, which, however, requires a significant initial setup effort, as it is fully designed for Kubernetes-based environments. Moreover, the monitoring and explainability functionalities in Kubeflow are enabled through integration with *Seldon Core* (2026), an open-source tool developed by Seldon Technologies, a company that specializes in building infrastructure and tooling for machine learning systems.

Regarding explainability support, some tools listed in Table 3.1 do not provide native,

built-in mechanisms for generating model explanations. Instead, explainability is typically enabled through integrations with external libraries or services that offer post-hoc interpretation techniques, such as feature importance, model-agnostic explanations, or visualization tools. For example, in the case of Kubeflow, explainability functionalities are commonly achieved through its integration with Seldon, which supports the deployment of models together with explanation services. Similarly, commercial platforms such as Vertex AI and SageMaker offer explainability features as part of their managed services, often relying on proprietary or tightly integrated explanation frameworks rather than open, standalone XAI tools.

3.2.2 Experimentation Tools

With the growing attention paid to machine learning in the last decade, the experimentation phase has become one of the most contemplated stages in the development of tools and libraries to support the life cycle of ML systems. An experimentation tool assists the developer in carrying out experiments by keeping execution logs, performance metrics, data, and parameters used. To speed up the creation of experiments, the developer can also use automated machine learning approaches, as presented in Section 2.7.

Experiment version management can be done using tools such as *DVC - Data Version Control* (2026). DVC is a command-line tool for version control of machine learning data and experiments, operating in a similar way to Git. It creates a hidden directory in the project to maintain an index and stores different versions of artifacts in user-defined locations, which can be on local disk or on remote storage services such as Amazon S3 and Google Cloud Storage. *MLflow*, already mentioned in Section 3.2.1, can also be used in its open source version by developers who are not Databricks customers, allowing them to manage experiment metadata.

Companies such as *Neptune* (2026) offer online platforms as a service, aimed at managing all the metadata related to model building. *Verta AI* (2026), on the other hand, offers a platform that covers model experimentation, deployment, and monitoring. It also maintains the development of *ModelDB* (2026), an open-source system for managing machine learning model metadata.

In addition to the experimentation tools available on online platforms or in their respective open-source versions, there are also other tools widely used in the development of machine learning models. One example is *Jupyter Notebook* (Kluyver *et al.*, 2016), which provides an interactive environment for the step-by-step development of models, making it easier to visualize results and reproduce the steps taken.

Another example is *PyCaret* (2026), a Python library that encapsulates various machine learning libraries, such as *scikit-learn*, *XGBoost*, *LightGBM*, *CatBoost*, *spaCy*, *Hyperopt*, *Ray*, among others. With *PyCaret*, you can build models with just a few lines of code and store the *pipelines* and models generated. However, the tool does not offer support for the deployment and monitoring stages of the models. This is an example of an AutoML framework, as is *MLJar*, mentioned in Section 2.7.

Kedro (2026) is another open source Python framework for creating reproducible and modular data science experiments. An experiment code developed with Kedro adheres to

software architecture standards, which facilitates the future maintenance of ML systems. In practice, Kedro encourages a clear separation between configuration, data access, and business logic, organizing projects into pipelines composed of reusable nodes that can be tested and executed independently. In addition, the framework provides a data catalog to centralize the definition of data sources and sinks, support for versioning datasets, and a standardized project template, which together help teams collaborate, track experiments, and deploy pipelines to different environments in a more systematic way.

There are tools that function as centralized data repositories, known as *feature stores*, which feed multiple machine learning systems within a company, sharing and versioning data from different models (Patel, 2020). Among the open-source projects, one that has gained significant relevance in the machine learning developer community is *Feast* (2026), which has more than 6.5k stars on GitHub. The *Feast* project is maintained by the company *Tecton*, which also offers, as a service, a platform for centralizing machine learning attributes, enabling transformations on these attributes and supplying them in a resilient manner.

3.3 Drift Detection

In production machine learning systems, monitoring for drift is an operational requirement on the same level as deployment, retraining, and resource provisioning, because changes in data distributions can silently and progressively degrade model performance over time (Sculley *et al.*, 2015). From a systems perspective, data is a first-class artifact whose quality and stability must be tracked throughout the pipeline rather than only during training. This motivates the development of explicit mechanisms to detect and react to both data drift (changes in input or label distributions) and concept drift (changes in the relationship between inputs and outputs) throughout the software lifecycle.

The academic literature commonly distinguishes between data drift and concept drift, often assuming online or streaming-learning settings. Ogasawara *et al.* (2025) provide a unified view connecting concept drift detection with event detection in time series, emphasizing metrics such as detection probability and detection delay. Complementarily, J. Lu *et al.* (2018) survey core drift types (sudden, gradual, incremental, recurring), summarize three main detection families (error-rate monitoring, window-based statistical tests, and ensemble-based methods), and discuss how detectors integrate with adaptive learners. Although many of these methods were designed for streams, their underlying principles—distributional comparison, error monitoring, and temporal segmentation—remain relevant in batch-oriented MLOps environments.

Within this line of research, Gama and Castillo (2006) propose learning with local drift detection, in which drift is identified within localized regions of the feature space rather than across the entire data stream. This regional perspective enables targeted adaptation when different subpopulations evolve at different rates, a scenario increasingly common in production pipelines involving heterogeneous data sources or user groups. Local detection methods, therefore, complement global detectors and align well with MLOps architectures that must respond to fine-grained drift signals.

Data drift is often monitored independently of model error, especially when labels

are scarce or delayed. In this context, Ditzler and Polikar (2011) introduce a drift detector based on the Hellinger distance between reference and current input distributions, computed via histograms. This technique supports both abrupt and gradual changes and operates effectively in low-label environments, making it suitable for the early-warning layer of a monitoring subsystem. Its reliance on statistical divergence rather than classifier behavior allows practitioners to trigger model retraining or deeper investigation even before downstream performance degrades.

An alternative approach utilizes information-theoretic summaries for the scalable monitoring of high-dimensional data. Dasu *et al.* (2006) propose change detection in multidimensional streams by tracking statistics derived from information measures over time. These approaches emphasize computational efficiency and robustness, aligning with modern ML system architectures in which numerous features and data sources must be tracked simultaneously. Their applicability to batch monitoring makes them compatible with data-intensive observability pipelines.

Finally, Rabanser *et al.* (2019) offer an extensive empirical comparison of data drift detection methods under controlled perturbations of covariates and labels. Their study shows that two-sample statistical tests applied to low-dimensional representations – often obtained from pre-trained models – tend to be among the most reliable and efficient detectors across a broad range of shift scenarios. They also evaluate domain-discriminator approaches, where a classifier is trained to distinguish between training and test data. Their findings underscore that practical drift is often detectable with relatively few samples when the representation and test statistic are chosen appropriately; however, detection quality remains highly dependent on the type and magnitude of the shift. This reinforces the need for problem-specific detector selection in real-world MLOps workflows.

3.4 Chapter Remarks

In addition to the academic literature on software architectures for machine learning and data drift detection, we analyzed open-source tools currently adopted in both industry and academia for building robust machine learning systems. This analysis revealed that, although mature tools exist to support individual stages of the machine learning lifecycle, there is a notable lack of consolidated architectural references that guide their integration into a comprehensive and systematic monitoring solution.

Based on this review, a research gap was identified at the intersection of software architecture and machine learning: few studies propose a holistic architecture for ML systems that includes a detailed decomposition of components specifically aimed at observability.

In this context, the master's project presented in this document leverages insights from related works to design a software architecture that enables the monitoring of relevant stages of the ML lifecycle. Additionally, to address the specific challenge of active data monitoring, this work presents experiments conducted in a simplified environment. These experiments replicate a batch prediction scenario to evaluate monitoring techniques, simulating the data input conditions found in real-world systems.

Chapter 4

Software Architecture for Supervised Machine Learning Systems

The first contribution of this master's thesis is to define a software architecture for a supervised machine learning system that enables monitoring all stages and their respective components involved in its development cycle. The aim of this architecture is to make it easier for developers to implement end-to-end machine learning systems, thereby reducing technical debt that may arise in this type of system and promoting an understanding of the various stages that need to be monitored.

Through the monitoring made possible by the proposed architecture, the system will have the necessary information to adopt MLOps practices, as discussed in Section 2.5. In particular, the system will facilitate feedback loops, which consist of mechanisms for propagating information between development stages, as detailed in Section 4.1.

4.1 Feedback Loop

A feedback loop consists of making information from a machine learning system's production environment available to the environment where the models were built. The information that must be available across system development stages can be found in any of the three layers of machine learning systems: data, source code, or model.

Figure 4.1 is a representation of a feedback loop in machine learning systems, showing how information from different layers is needed in the system's development stages. The figure illustrates the stages involved in developing a model until it is implemented in an ML system, along with the monitoring required to maintain the model's reliability.

During the monitoring stage, information from previous model development stages must be available, including the performance measures of the deployed model and the training data used. In this way, it is possible to compare the data from the system in production with the training data to assess whether a new model needs to be developed. With external feedback data, it is possible to determine whether the model's performance in the production environment is satisfactory by comparing it with the measurements

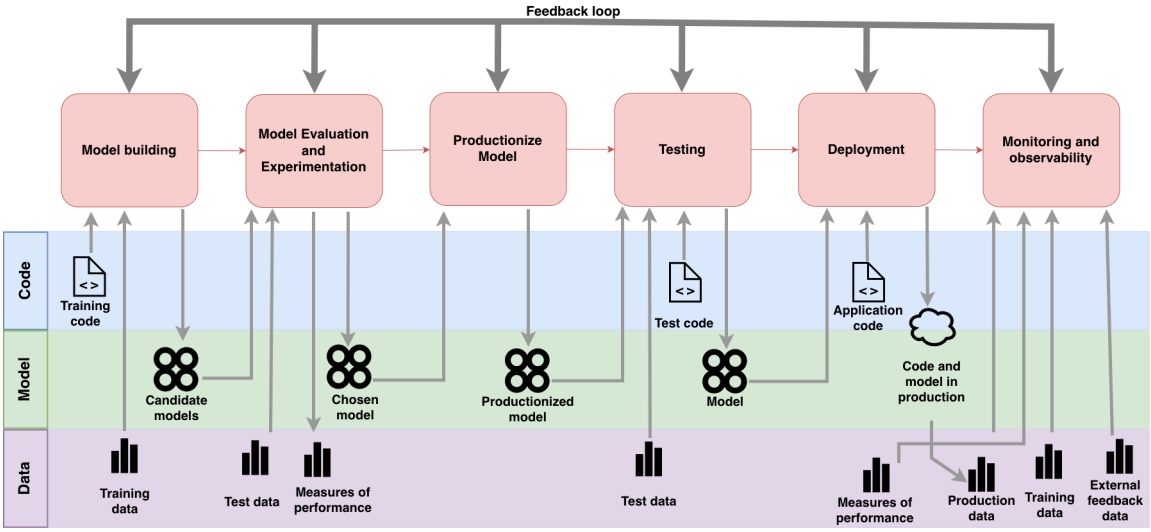


Figure 4.1: *Representation of a feedback loop for ML, adapted from the work of* Sato et al. *(2019).*

obtained during the model's evaluation and experimentation stages.

If the model's performance is below the expected level or the production data differs significantly from the training data, a time window of the production data can be used to build a new version of the model. The way this data is used in the construction and validation of a new version is the developer's responsibility, and the best decisions must be made to maintain the reliability of the system that depends on the model developed.

Lastly, it is important to emphasize that the feedback loop is not strictly sequential: information can flow into any stage of the system development lifecycle whenever necessary. Monitoring may reveal issues that do not require model rebuilding – for example, an overload in the deployed model may indicate the need for infrastructure adjustments rather than changes to the model itself. Thus, the feedback loop enables all stages to exchange data, code, and model information flexibly, supporting corrective actions ranging from retraining to operational adjustments, depending on the nature of the problem observed.

The next sections show how the software architecture of a machine learning system can be enriched with subsystems to manage the system's data and support feedback loops. The architectural diagrams presented in this work are defined using *Unified Modeling Language* (UML) Component Diagrams (*OMG Unified Modeling Language (UML) Specification, Version 2.5.1* 2017). Table 4.1 shows the UML graphical elements used in the diagrams, together with their respective functions.

4.2 Simple Architecture

The architecture shown in Figure 4.2 has only the subsystems needed for a simple model deployment. There is a training component using a data set from a source other than the production data. Deployment is also as minimalist as possible, using the model only to generate predictions and collecting information only about the predictions made.

The architecture in Figure 4.2 represents machine learning systems that focus solely

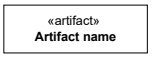
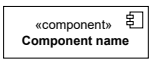
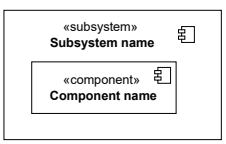
Component name	Description	Graphical representation
Port	A port is used to specify a point of interaction through which a component can communicate with the environment, with other components, or with its internal parts.	
Provided Interface	The provided interface is an element that defines the operations offered by a component.	
Required Interface	The required interface is an element that defines the operations requested by a component.	
Dependency	A dependency starts from the component that depends and points to the one it depends on.	
Delegation Connector	A connector that maps a component's external interface, defined by its ports, to the internal elements that realize its behavior.	
Artifact	An element that represents a physical piece of information used or produced during system development, deployment, or operation.	
Component	A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.	
Subsystem	Subsystems are a specialized type of component that represent independent behavioral units within a system. They are typically used to model large-scale functional elements of complex systems.	

Table 4.1: *Table with the UML graphical elements used in this work and their respective meanings.*

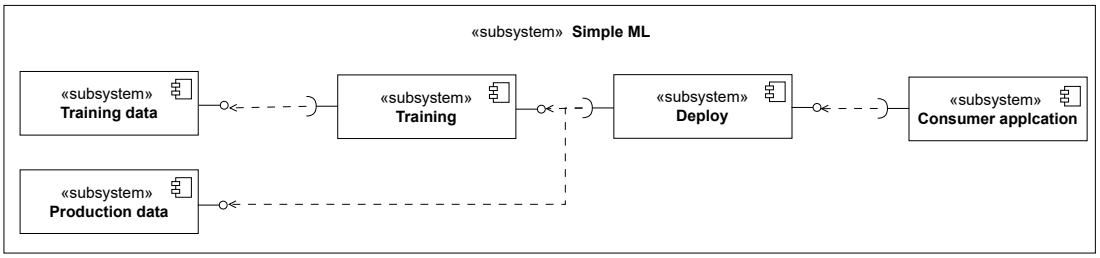


Figure 4.2: *Architecture for a simple ML system.*

on training and using a machine learning component to generate predictions, without concern for the model monitoring stage.

4.3 The Architecture for Enabling MLOps

The architecture proposed in this work (Figure 4.3) takes into account the *FAIR* data principles elaborated by Wilkinson *et al.* (2016), ensuring that all data created within the framework is findable, accessible, interoperable, and reusable. The design of the components also takes into account an ontology that schematizes machine learning concepts, proposed by Publio *et al.* (2018), which provides a set of classes, properties, and restrictions for representing and exchanging information about machine learning algorithms, data sets, and experiments.

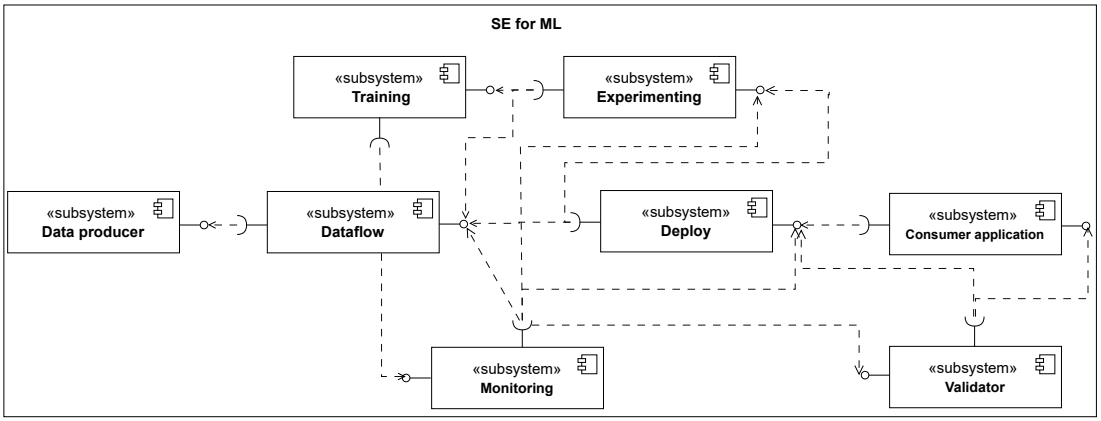


Figure 4.3: *Generic architecture of the subsystems interactions.*

The architecture is described using the UML 2.5 subsystems concept to represent the major areas of activity in the development of an ML system, in an attempt to overcome a shortcoming of current models, which do not capture all the information needed by the various stakeholders. The architecture also accounts for the heterogeneity of tools used to implement each stage of the machine learning system's development, using subsystems to segment each tool's domain of activity. Each subsystem consists of components that represent the entities needed to develop an ML system, including all the data required to execute the feedback loops.

The architecture diagram in Figure 4.3 shows the interactions among the subsystems involved in the development of an AM system, which aims to maintain adequate observability to ensure robustness and reliability. The following sections detail the components that build the subsystems and their interactions. An overview of the architecture, with more detailed component interactions, is presented after the subsystem sections, in Figure 4.9.

4.3.1 Data Flow

The data flow subsystem (Figure 4.4) is the gateway to the data that will serve as inputs for the system, and is responsible for carrying out all the relevant transformations that this data may require. The organization of the components of this subsystem was designed to maintain the provenance of the data and the record of all the transformations it undergoes.

This subsystem incorporates the *Feature Store*, introduced in Section 3.2.2, which is responsible for storing and providing the data required by experimentation workflows and production models. Tools such as *Feast* (2026) exemplify this functionality.

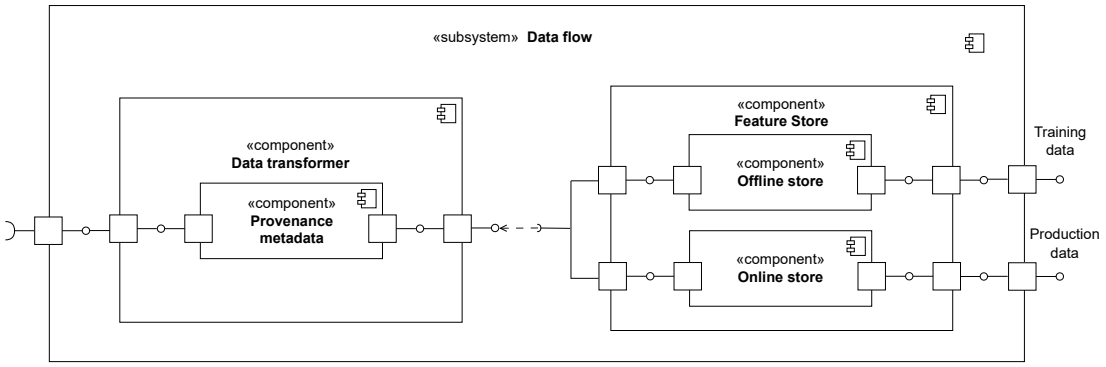


Figure 4.4: *Dataflow subsystem.*

4.3.2 Training and Experimentation

The training subsystem (Figure 4.5) serves as an interface for triggering the experimentation subsystem, determining which data to consume from the data flow system, how much to consume, and which parameters to use in the experiments. These parameters can be defined by the system's developers however they prefer, but they can also be better selected based on information from the monitoring components. If a model performs poorly on a production data set, the developer can use this information to design new experiments. To increase the system's robustness, monitoring data can be used to align the training data with the production context. Typical tools supporting these activities include interactive environments, workflow frameworks, and versioning systems, such as Kluyver *et al.* (2016), *Kedro* (2026), and *Data Version Control* (2026).

The experimentation subsystem (Figure 4.6) is responsible for storing all the data related to the experiments carried out until a production model is obtained. Its components have been depicted with ports and interfaces to enable the necessary interactions throughout the model's life cycle. The data stored in this subsystem serves as input for subsequent

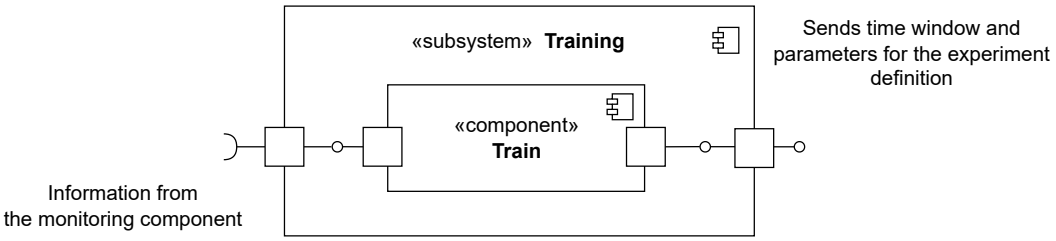


Figure 4.5: *Training subsystem.*

decision-making in the feedback loop and contributes to the system's overall robustness and reliability once deployed in production. Tools such as MLFlow (Chen *et al.*, 2020) can support this process by tracking experiments, organizing artifacts, and centralizing the information generated throughout the model development cycle.

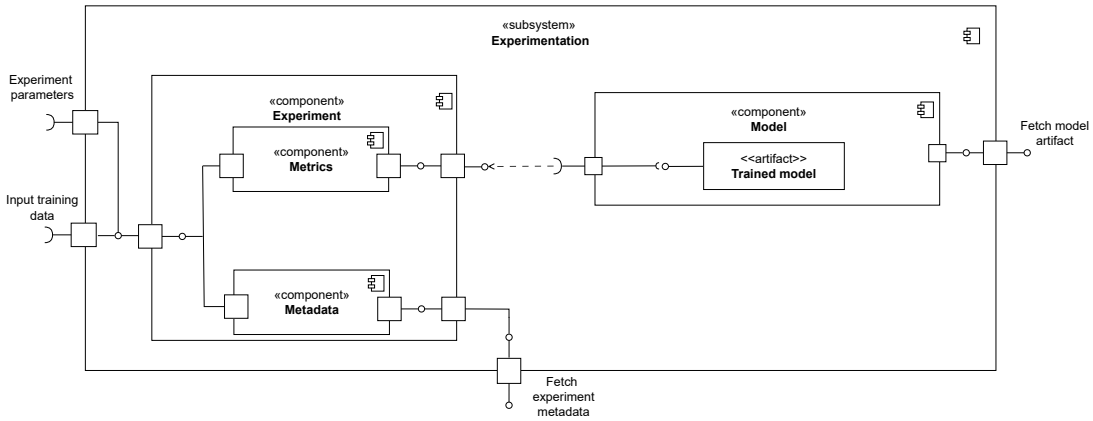


Figure 4.6: *Experimentation subsystem.*

4.3.3 Deployment, Consumption and Validation

The model deployment subsystem (Figure 4.7) includes a generic component that represents the producers of predictions and the consumers of models. The communications between components aim to highlight each component's responsibilities and its interactions with external subsystems, enabling the feedback loop.

The external consumer subsystem was added generically to the diagram to represent its role in the feedback loop dynamics, since the entire model was designed to consume from this subsystem. Separately, the validation subsystem is also very important because it provides feedback, enabling the system to be better monitored and improved. Validation has been left in a separate component because it will not necessarily be carried out by those responsible for the consuming application.

Tools such as *Kubeflow* (2026) and *Seldon Core* (2026), as well as cloud-provider platforms like *Vertex AI* (2026), *Sagemaker* (2026), and *Azure Machine Learning* (2026), can support both this subsystem and the one described in the next section by managing model deployment, orchestration, and validation workflows.

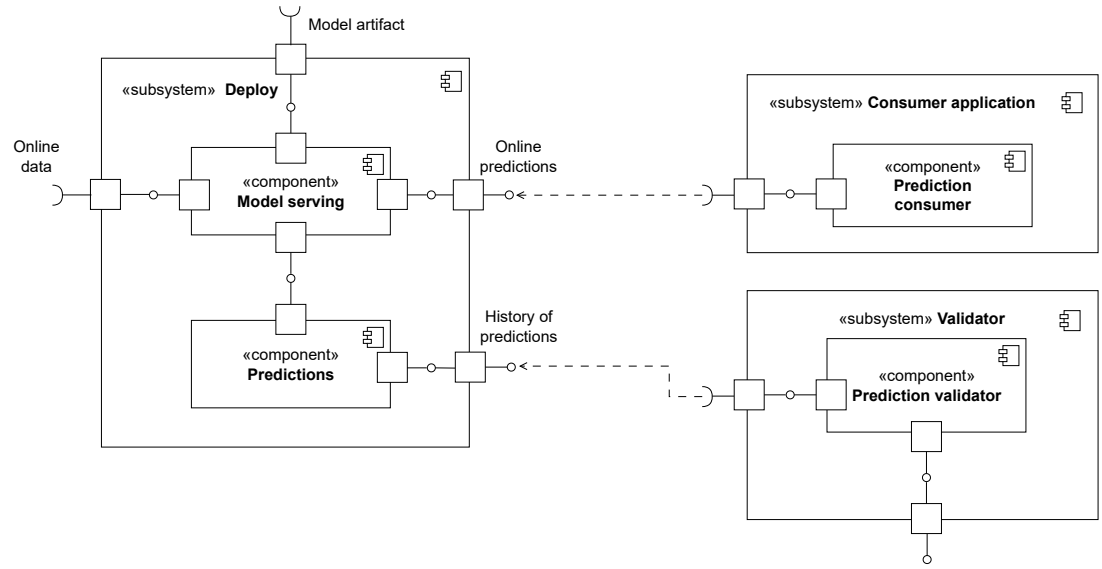


Figure 4.7: *Deploy subsystem and its near interactions.*

4.3.4 Monitoring

The monitoring subsystem (Figure 4.8) is the central component that communicates with all the other subsystems in order to carry out the feedback loop. It includes various types of monitoring, such as detecting potential deviations in input attributes, in the model's prediction profiles, and even in users' computing load.

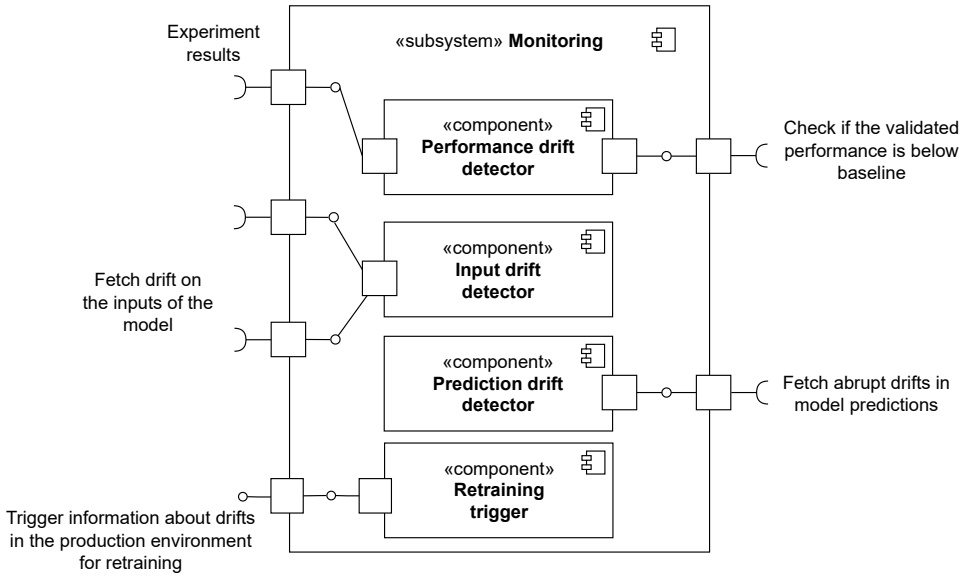


Figure 4.8: *Monitoring subsystem.*

In the framework's monitoring subsystem, there is a component that enables concept drift detection based on monitoring input data distributions (step 1 of the concept drift detection process described in Section 2.3.2), and there is also a component for error rate-based validation, which involves developing a classifier (Section 2.3.2).

The information from monitoring the models may be used to automatically construct new models using AutoML (Section 2.7), or it may simply make the information readily available so that the system maintainers can decide whether to retrain the deployed models.

4.4 Data Model

The logical data model supporting the MLOps architecture proposed in this work is illustrated in Figure 4.10. In this model, all data used and generated by the system is uniquely identified through a combination of identifiers. This approach enables information from different stages of the model development and deployment cycle to be cross-referenced, facilitating effective feedback loops.

The attributes that serve as model inputs are represented generically in the `attribute` table. This abstraction ensures that the system remains agnostic to the data source – whether it is a relational database, a non-relational database, or a file – thereby preserving the system's flexibility and robustness.

Attributes can be combined to create new attributes, which can then be included in new *datasets*. These operations are captured in the `transformation` table, which records the relationship between each transformation, its input attributes, and the resulting output attribute. For attributes generated by transformations, the `source_id` is set to null.

A `dataset` is defined as a collection of attributes and may have multiple versions, tracked in the `version_dataset` table. Each version represents an evolution of the dataset as data is updated or expanded. Each experiment is associated with a specific `dataset`, and an identifier specifies which attribute serves as the target variable.

Each `experiment` consists of one or more runs, recorded in the `execution` table. These runs generate both performance metrics and trained model artifacts. The resulting metrics and models are directly linked to the metadata defined at the time of the experiment, such as algorithm hyperparameters and the computing infrastructure used. Metrics and models are stored in the `experiment_metric` and `model` tables, respectively.

Models produced from experiments can be deployed to generate predictions. Each prediction is linked to the deployed model, allowing tracing the model back to the experiment that produced it. This linkage enables a comparison between the real-world performance of the deployed model and the performance metrics recorded during experimentation. Information regarding model deployment, generated predictions, and their subsequent validation is stored in the `deployment`, `prediction`, and `validation` tables, respectively.

4.5 Chapter Remarks

This chapter presented an architecture that enables feedback loops between components of an ML system. The discussion integrated relevant tools introduced in the previous chapter, illustrating how they can be combined to form a cohesive and practical scenario for a hypothetical ML pipeline. A data model was also proposed to support the storage of key information required for enabling the feedback mechanisms. By following the

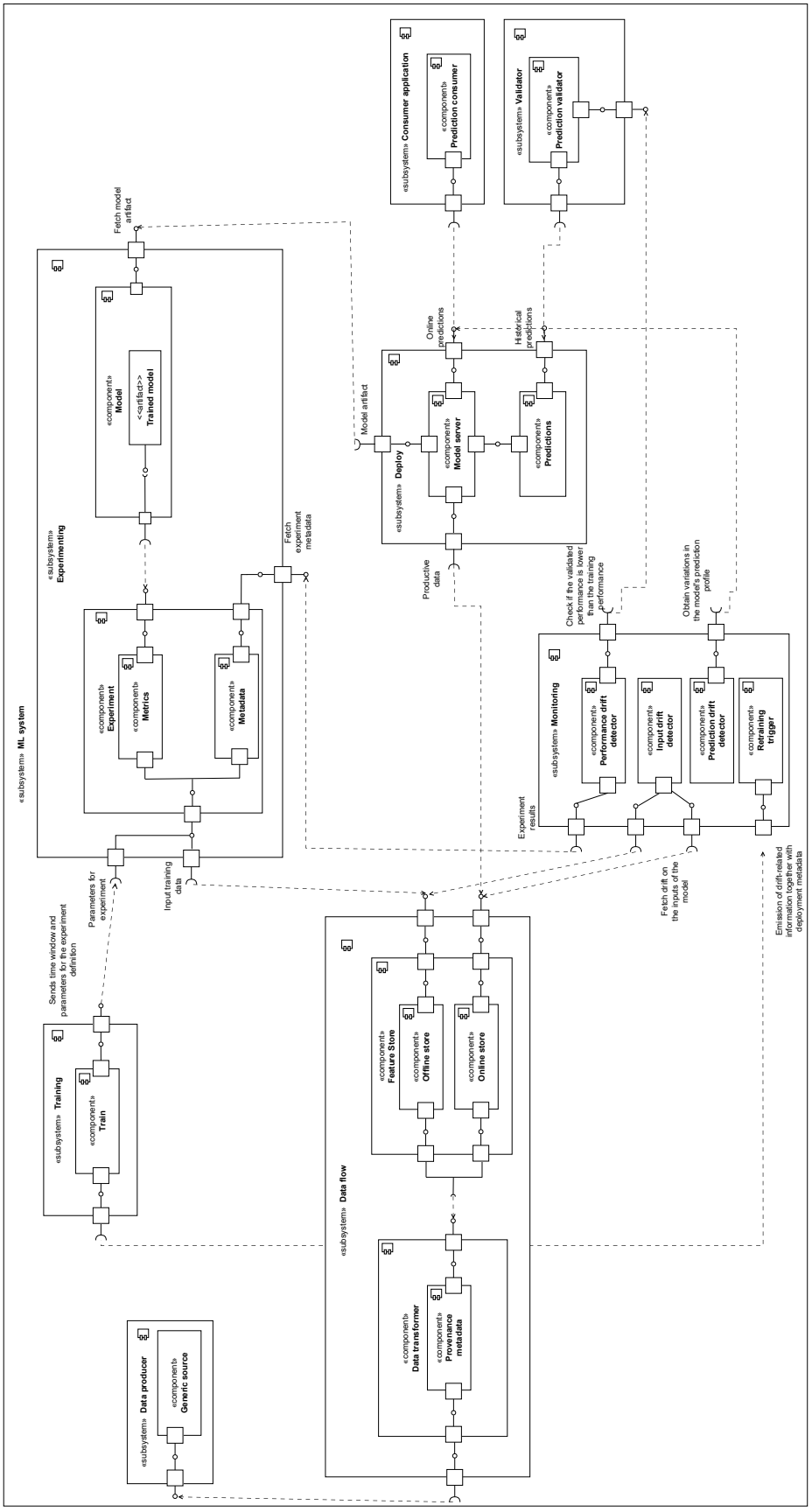


Figure 4.9: *Architecture overview with all its interactions.*

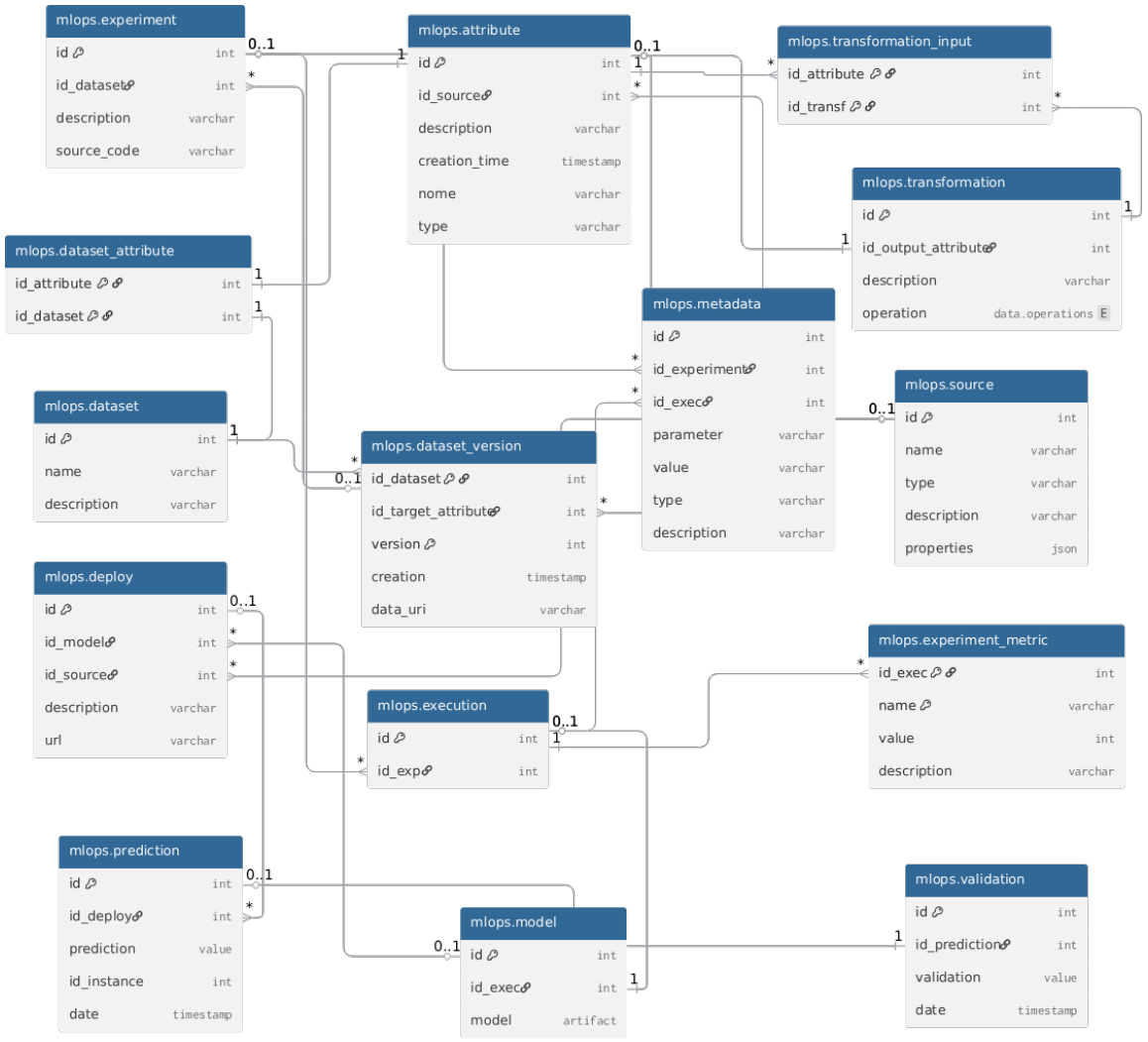


Figure 4.10: *Data model for the architecture.*

principles outlined in this chapter, an ML system would be better equipped to monitor changes across multiple components and enhance its overall robustness.

Chapter 5

Detecting Drift in Datasets

This chapter investigates how different data drift detection techniques behave when applied to a variety of datasets. The goal is to evaluate their ability to identify distributional changes that may affect model performance in real-world scenarios.

A diverse set of datasets was employed, including real-world, synthetic, and benchmark datasets commonly used in concept drift research. These datasets are introduced and described in Section 5.1. Section 5.2 details the application of different data drift detection techniques, including visualizations and quantitative metrics used to assess their performance.

5.1 Datasets

In our experiments, we used datasets cited by J. Lu *et al.* (2018) in their literature review on learning under concept drift. The objective of utilizing these datasets is to examine how data drift detection methods respond in a concept drift scenario. Specifically, we employ two datasets in which the distributions remain stable while only the concept drifts. Additionally, we utilized synthetic datasets with data drift to further investigate the behavior of the proposed techniques.

5.1.1 Insects Datasets

The Insects Datasets (Souza *et al.*, 2020) are derived from a real-world streaming application that uses optical sensors to capture data and, in real time, recognize flying insect species with a smart trap. The temperature is the main environmental factor influencing the insect behavior in the trap and, consequently, the data captured. Observations were ordered over time based on temperature patterns, and examples were uniformly sampled within each temperature to isolate temperature-induced drifts. Then, the data were split into 11 datasets with 33 features, showcasing different patterns of change (incremental, abrupt, gradual, recurring), balanced and unbalanced classes, and changes in class distribution.

5.1.2 SEA Datasets

The Streaming Ensemble Algorithm (SEA) synthetic dataset (Street and Kim, 2001) simulates a data stream with abrupt concept drift. Each instance is described by three features, of which only the first two are relevant. The target variable is binary and takes a positive value when the sum of the relevant features exceeds a predefined threshold. Concept drift is induced by altering this threshold, effectively changing the underlying classification function.

We used an implementation of the SEA dataset provided by River (Montiel *et al.*, 2021), a library to build online machine-learning models. In the library, there are four different variants to be chosen as threshold configurations: (0) threshold = 8, (1) threshold = 9, (2) threshold = 7, and (3) threshold = 9.5. Using them, we built two different experimental setups:

1. **SEA**: a stream that starts with variant 0, has variant 3 in the middle, then returns to variant 0, simulating a deviation in concept for a period;
2. **MULTISEA**: a scenario that has 12500 items from each of the four variants.

5.1.3 STAGGER Datasets

The STAGGER synthetic datasets introduced by Schlimmer and Granger (1986), like SEA, simulate data with abrupt concept drift. In STAGGER, objects are described by three features – size (small, medium, and large), shape (circle, square, and triangle), and color (red, blue, and green) – and the target is a boolean value given by a function f. The River library provides three variants for f: (0) True if the size is small and the color is red, (1) True if the color is green or the shape is a circle, and (2) True if the size is medium or large. Changing f causes concept drift. Using River, we built two experimental setups:

1. **STAGGER**: a stream that starts with variant 0, has variant 1 in the middle, then returns to variant 0, simulating a deviation in concept for a period;
2. **MULTISTAGGER**: a scenario with 16666 items from each of the three variants.

5.1.4 Electricity

The Electricity Pricing dataset (Harries, 1999) is used in our experiments to simulate an environment with concept drift and class imbalance. The dataset originally contains 45312 instances collected at half-hour intervals from the New South Wales electricity market, spanning May 7, 1996, to December 5, 1998.

5.1.5 Magic Gamma Telescope

The MAGIC Gamma Telescope dataset(Bock, 2007) is generated via Monte Carlo simulations to model the registration of high-energy gamma particles by a ground-based atmospheric Cherenkov telescope using imaging techniques. The images captured by the telescope enable discrimination between primary gamma events (signal) and hadronic showers induced by cosmic rays (background). As the original dataset exhibits minimal

drift, it was modified by sorting the `fConc1` feature in ascending order and constructing incremental data batches that introduce meaningful distributional drift, as described in (Ditzler and Polikar, 2011).

5.1.6 Synthetic Datasets

To enable a more precise and rigorous evaluation of data drift detection techniques, this study also utilizes controlled synthetic datasets. These datasets enable complete control over drift occurrences, allowing for objective assessments of detection performance under known conditions. The datasets represent a random binary classification problem and were generated using the `make_classification` function from the scikit-learn library, following the methodology described by Guyon (2003).

Base Dataset Configuration

Each synthetic dataset contains 80000 samples and five features, of which four are informative, and one is redundant. The datasets were generated with a fixed random seed to ensure reproducibility. All feature values were normalized to positive ranges through an additive transformation, and feature names follow a consistent naming convention (`feature1` through `feature5`). The target variable is binary and stored in a dedicated `class` column.

Drift Scenarios and Feature Selection

The synthetic datasets model multiple types of concept drift, inspired by real-world scenarios and grounded in two theoretical patterns: *parallel drift* and *switching drift*. In the parallel pattern, multiple features drift simultaneously, modeling global changes across the system. In contrast, the switching pattern involves sequential drift across individual features, representing cascading or localized changes.

Drifts were applied to a subset of features: `feature1`, `feature3`, and `feature5`. The remaining features, `feature2` and `feature4`, were left unchanged to serve as controls. Each drift-affected feature undergoes two drift events. All drift windows are aligned with batch boundaries to simplify interpretation and reduce noise in performance analysis.

Drift Types and Mathematical Transformations

Two types of drift were implemented:

- **Abrupt drift:** A fixed value of +5.0 was added to feature values within the drift window, causing an instantaneous distributional shift.
- **Incremental drift:** Feature values were modified progressively, starting with an additive value of 0.05 and increasing by 0.001 per sample throughout the drift window.

These transformations were applied either in parallel (affecting all drifted features simultaneously) or in switching mode (affecting one feature at a time in a staggered sequence).

Drift Timing and Batch Configuration

Drift events are positioned to occur after the second processing batch, ensuring a stable baseline. Each dataset contains 20000 drifted samples, spread over two events per affected feature. Experiments were conducted with batch sizes of 1000, 1500, 2000, and 2500, allowing for the assessment of how processing granularity influences detection performance.

Dataset Variants

Five synthetic datasets were used, each modeling a specific drift scenario:

- **SYN:** Control dataset with no drift.
- **SYN-PA:** Parallel abrupt drifts.
- **SYN-PI:** Parallel incremental drifts.
- **SYN-SA:** Switching abrupt drifts.
- **SYN-SI:** Switching incremental drifts.

The distributions for SYN-PA and SYN-SI are illustrated in Section 5.2 through heatmaps that show changes in the data distribution across the feature space.

5.2 Analysis of Drift Detection

The following sections present visualizations of the detected drifts and the performance metrics for the detection techniques. To evaluate these techniques, the datasets were segmented into batches of 1000, 1500, 2000, and 2500 instances. These batch sizes were selected to balance dataset size constraints while ensuring a sufficient number of batches for a robust evaluation. Additionally, this segmentation enables an analysis of how the sensitivity of each technique varies with batch size. The following drift detection techniques were tested: Base (no drift detection), KS95 (KSDDM with $\alpha = 5\%$), KS90 (KSDDM with $\alpha = 10\%$), HD (HDDDM with $\gamma = 1$, as specified in (Ditzler and Polikar, 2011)), and JS (JSDDM with $\gamma = 1$).

5.2.1 Visualization of the Detected Data Drifts

Figures 5.1 and 5.2 shows how the drift detection techniques behaved for the MULTI-STAGGER and MULTISEA datasets, and Figures 5.3 and 5.4 shows the behavior for the Insects' abrupt balanced and imbalanced datasets, which exhibit concept drift. Figures 5.5 and 5.7 provide examples of drift detection in datasets with synthetic data drift.

In these figures, each point represents a detected drift in a given batch, with colors distinguishing the detection techniques. The vertical dashed lines indicate the locations of known drifts (concept or data drifts, depending on the dataset). At the top of each plot, for the synthetic datasets, the affected features and the corresponding batch intervals where data drifts occur are indicated.

In the concept drift scenario, even though data drifts are being detected, they are not being detected specifically at the datasets' concept-changing points. For the case of the data drift scenarios, the techniques are more prone to detect the drifts near the regions where drifts were added, behaving expectedly.

Heatmap plots can be used to visualize the drifts of all the features individually, as shown in Figures 5.8 and 5.6. This type of plotting provides valuable insights into the drift patterns detected from the batches and the reference set. On these plots, darker color shades indicate a greater degree of divergence between specific batches and the reference batch.

In the incremental scenario depicted in Figure 5.5, the KS test effectively detects changes within the drift interval, whereas JS and HD exhibit a slight delay in identifying variations in the feature distribution as data drift. Furthermore, Figure 5.8 illustrates how less significant differences are smoothed as the reference dataset expands, thereby enhancing the technique's adaptability to new data.

5.2.2 Metrics

To evaluate the performance of the drift detection techniques on the new synthetic datasets, the batches containing drift in the feature distributions can be used to assess whether the techniques are detecting drift accurately. The evaluation framework defines a true positive (TP) as a correctly identified drift in a batch. A false positive (FP) occurs when drift is detected, but none is actually present. True negatives (TN) correspond to cases where no drift is detected, and none occurs. False negatives (FN) represent batches in which drift is present but goes undetected.

This can be interpreted in Figures 5.5 and 5.7, where the dots within the intervals between the dashed lines represent true positives (TP), while the dots outside these intervals represent false positives (FP). Batches that do not contain detections within the dashed lines are classified as false negatives (FN), while batches outside the dashed lines that also lack drift detections are likewise considered false negatives (FN).

Table 5.1 presents the performance metrics of the drift detection techniques when applied to the synthetic dataset variations with batch sizes of 1000 and 2500. The tables include the metrics precision (P), recall (R), and F1-score (F1) calculated as presented in Section 2.2.

The results in Table 5.1 reveal how different drift patterns influence detection performance and, consequently, the retraining strategy of a classifier. Our primary interest lies in achieving higher recall values, as recall provides insight into the proportion of correctly identified instances within batches containing drift. Upon examination, it is evident that recall values tend to be higher for larger batch sizes, suggesting that smaller batch sizes may lead to less precise and assertive drift detection. Regarding precision, a lower precision score implies the potential for over-detection, which could introduce instability in a continuous learning system by triggering unnecessary retraining.

The techniques tend to detect drift less frequently in the case of abrupt drifts. This may be due to the inherent design of the drift detection methods, which may not be as responsive to the sudden changes characteristic of abrupt drift patterns. In contrast, for

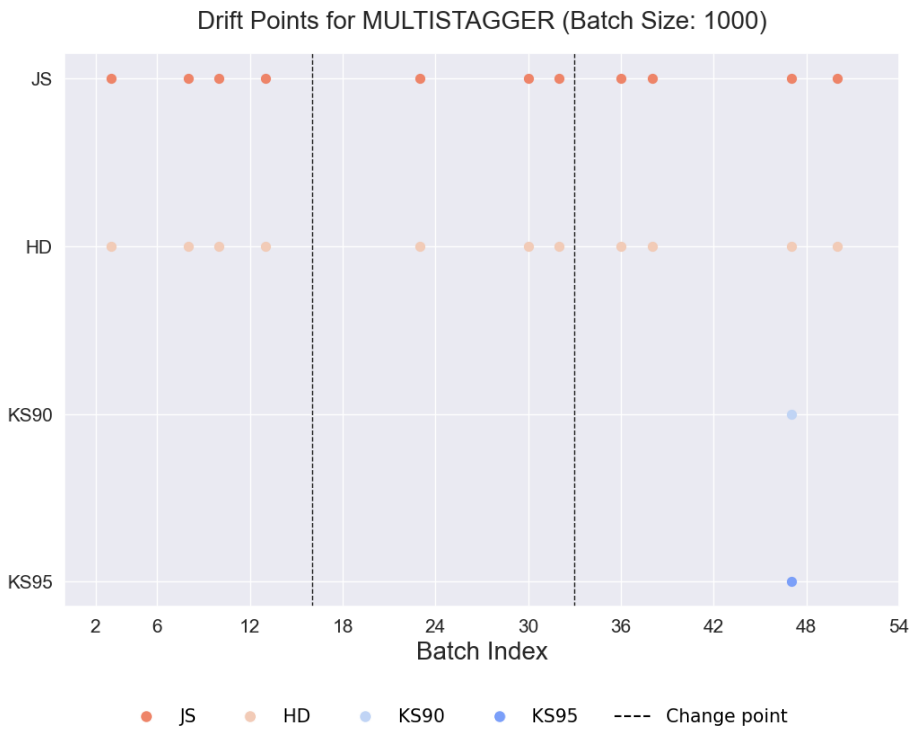
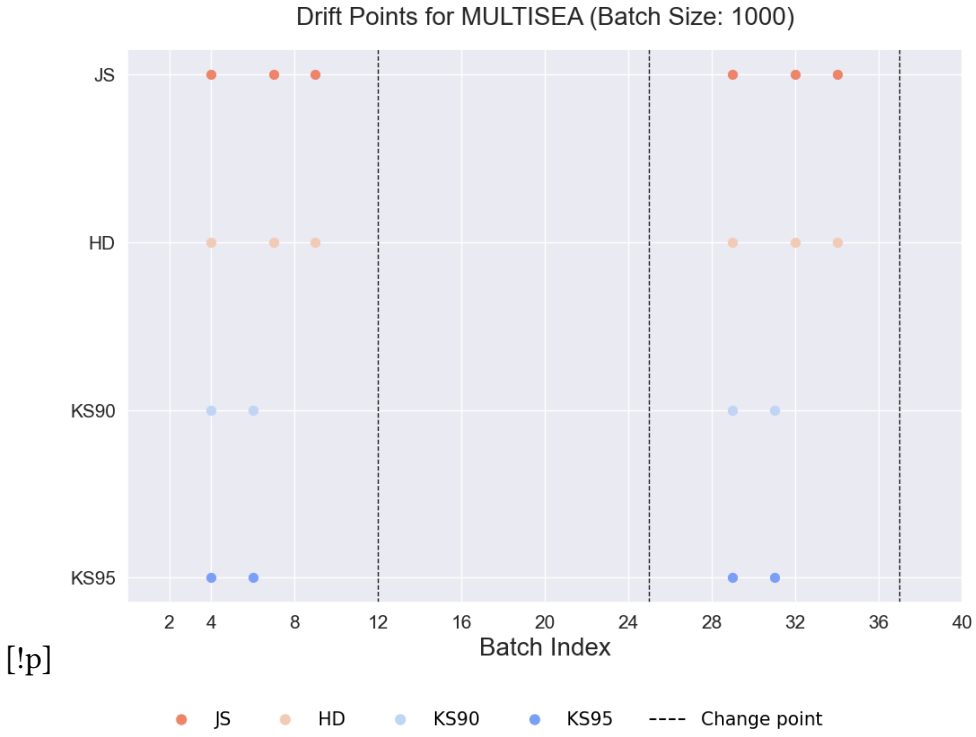


Figure 5.1: *Detected drifts for the MULTISTAGGER dataset (Batch size: 1000).*



[!p]

Figure 5.2: *Detected drifts for the MULTISEA dataset (Batch size: 1000).*

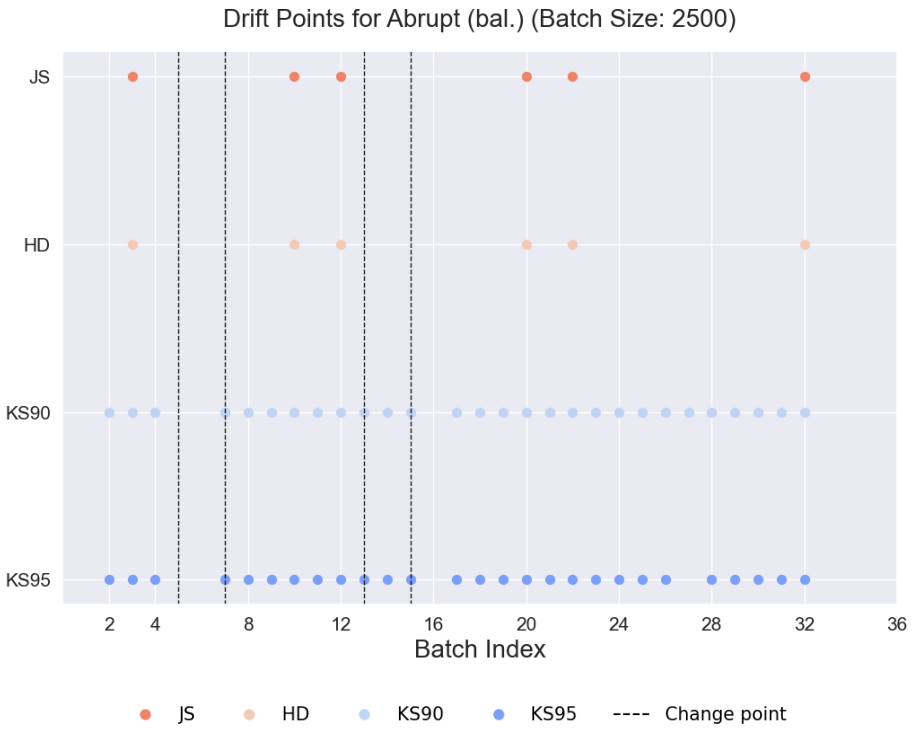


Figure 5.3: *Detected drifts for the Insects' Abrupt balanced dataset (Batch size: 2500).*

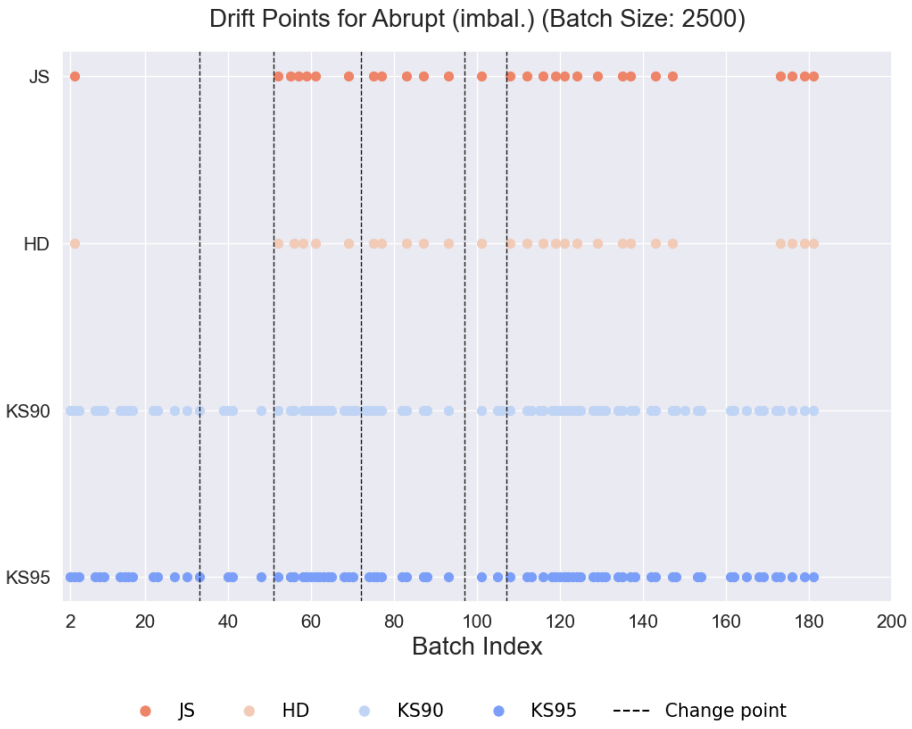


Figure 5.4: *Detected drifts for the Insects' Abrupt imbalanced dataset (Batch size: 2500).*

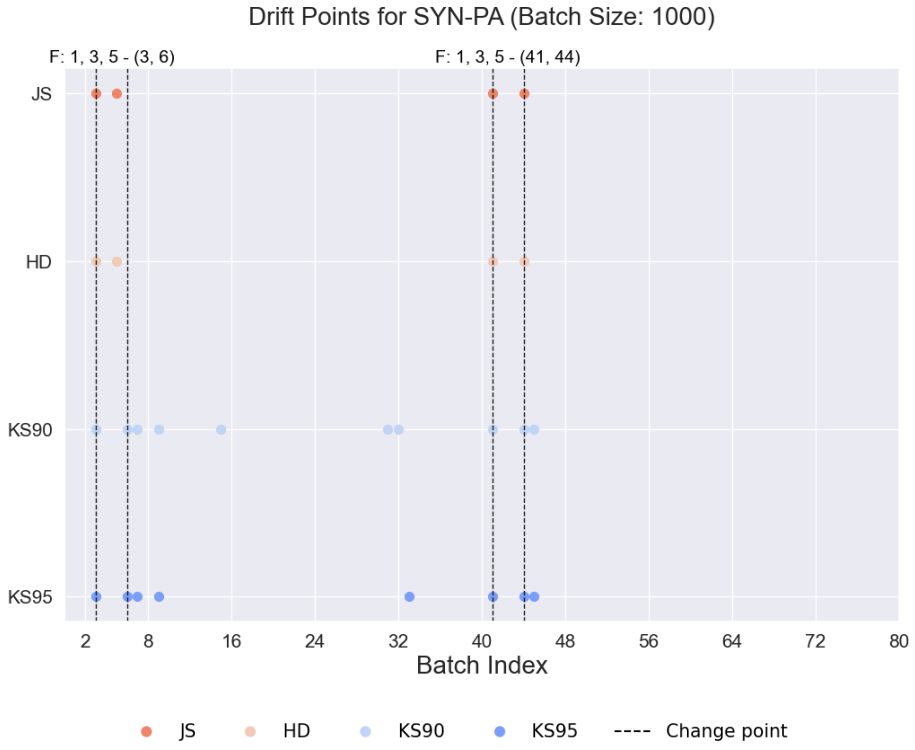


Figure 5.5: *Detected drifts for the SYN-PA dataset (Batch size: 1000).*

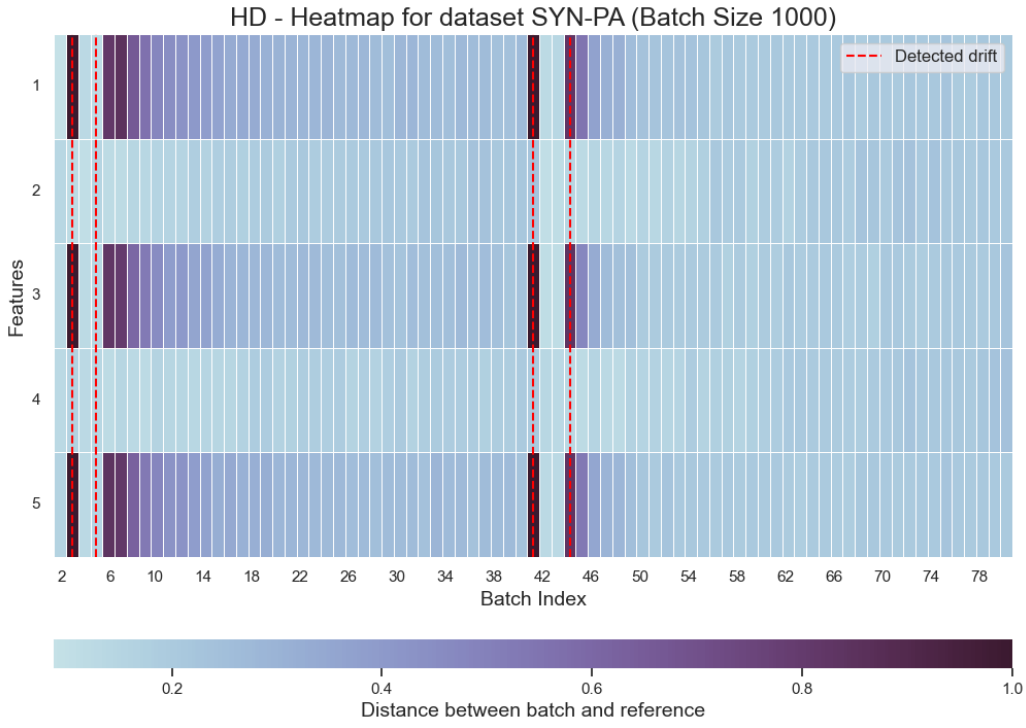


Figure 5.6: *Heatmap for the drifts detected using HDDDM for SYN-PA.*

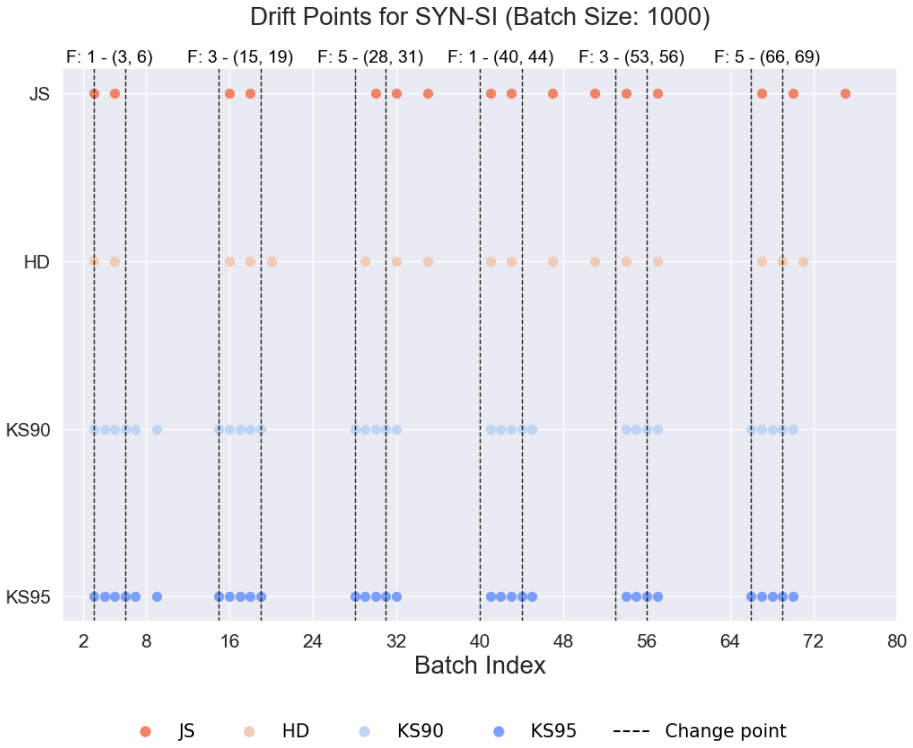


Figure 5.7: *Detected drifts for the SYN-SI dataset (Batch size: 1000).*

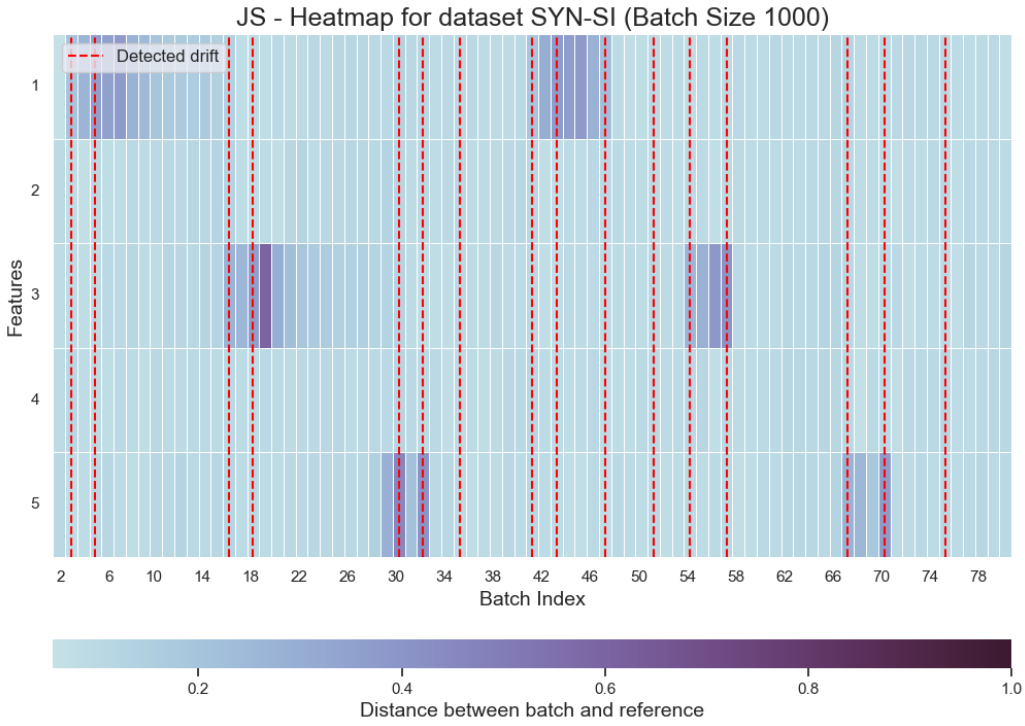


Figure 5.8: *Heatmap for the drifts detected using JSDDM for SYN-SI.*

Dataset	Technique	Batch Size = 1000			Batch Size = 2500		
		P	R	F1	P	R	F1
SYN-PA	KS95	0.5	0.5	0.5	0.571	1.0	0.727
	KS90	0.4	0.5	0.444	0.571	1.0	0.727
	HD	1.0	0.5	0.667	1.0	1.0	1.0
	JS	1.0	0.5	0.667	1.0	1.0	1.0
SYN-PI	KS95	0.667	1.0	0.8	0.571	1.0	0.727
	KS90	0.571	1.0	0.727	0.571	1.0	0.727
	HD	1.0	0.5	0.667	1.0	0.5	0.667
	JS	1.0	0.5	0.667	1.0	0.5	0.667
SYN-SA	KS95	0.714	0.577	0.638	0.706	0.857	0.774
	KS90	0.714	0.577	0.638	0.706	0.857	0.774
	HD	0.688	0.423	0.524	1.0	0.286	0.444
	JS	0.688	0.423	0.524	1.0	0.286	0.444
SYN-SI	KS95	0.8	0.923	0.857	0.722	0.929	0.813
	KS90	0.8	0.923	0.857	0.722	0.929	0.813
	HD	0.588	0.385	0.465	0.6	0.429	0.5
	JS	0.562	0.346	0.429	0.6	0.429	0.5

Table 5.1: *Precision (P), Recall (R), and F1-score (F1) of Drift Detection techniques for batch sizes* 1000 *and* 2500*.*

incremental drifts, the techniques can more consistently detect the drift over time. This intermittent detection of abrupt drifts could be attributed to the gradual nature of drift detection algorithms, which are often tuned to perform better with slower, incremental shifts in data.

From the perspective of the techniques, KS-based methods (KS95, KS90) generally performed robustly across datasets, maintaining high recall and improving precision with larger batch sizes. This suggests that they provide reliable drift detection, although careful tuning may be necessary to balance them, particularly for switching drifts. HDDDM and JSDDM, which rely on distance-based measures, show varying effectiveness depending on the dataset. While they struggle with recall in certain incremental scenarios (e.g., SYN-SA at batch 2500), they maintain reasonable precision in other cases, which helps reduce unnecessary retraining. These results suggest that no single technique is universally superior; rather, selection should be guided by the dataset's drift characteristics and the trade-off between detection sensitivity and retraining stability.

Chapter 6

Using Drift Detection within ML Systems

This study aims to provide an empirical evaluation of how statistical and distance-based methods for detecting data drift in input streams can be used to improve the performance of machine learning classifiers operating under non-stationary conditions. In particular, the experiments analyze whether retraining models based on detected distributional changes leads to better predictive performance compared to a baseline scenario in which no retraining is performed after the initial training phase.

The datasets described in Section 5.1 were used to (re)train classifiers and compute performance metrics under different drift scenarios. The experimental pipeline was implemented using the *scikit-learn*, *scipy*, *river*, and *Menelaus* libraries.[1]

This chapter presents the experimental evaluation designed to assess the impact of incorporating drift detection into the retraining process of machine learning systems. Building on the drift detection approaches analyzed in Chapter 5, Section 6.2 describes the experimental protocol adopted to evaluate how detected drifts can be leveraged to trigger model retraining. The subsequent sections analyze the resulting performance metrics and discuss the implications of continuous input monitoring for the deployment of robust and adaptive learning systems.

6.1 Using Drift Detection to Improve ML System's Performance

The classifiers used in the experiments were evaluated using a prequential (test-then-train) approach, in which predictions are generated for each batch before the models are updated. When a drift is detected by the selected technique, the drift-aware classifier is reset and retrained, following a strategy similar to that adopted by Souza *et al.* (2020). To implement this evaluation framework, we followed the algorithm described in Section 6.1.1,

[1] scikit-learn.org/, scipy.org/, riverml.xyz/, pypi.org/project/menelaus/

which maintains both a baseline classifier and a drift-aware classifier for performance comparison.

6.1.1 Prequential Algorithm

1. **Input**: Labeled data batches and a drift detection technique.
2. Use batch 1 to train a Naive Bayes classifier C_B – the baseline model.
3. Use batch 1 to train a Naive Bayes classifier C_D – the model that benefits from drift detection.
4. Set batch 1 as the reference batch.
5. **From batch 2 onwards:**
   (a) Store the predictions of both classifiers C_B and C_D for the current batch.
   (b) Check for drift between the reference set and the current batch using the drift detection technique.
   (c) Update C_B classifier with the current batch.
   (d) **If no drift is detected:**
      - Update C_D classifier with the current batch.
      - Update the reference set by merging it with the current batch.
   (e) **If drift is detected:**
      - Set the reference set to the current batch.
      - Reset classifier C_D training only with the new reference set.
6. **At the end of all batches:** Compute the performance metrics.

6.1.2 Learning and Retraining Strategy

In all experiments, a Naive Bayes classifier was adopted as the learning algorithm. Two models were maintained throughout the data stream: a baseline classifier C_B, which was continuously updated with incoming batches, and a drift-aware classifier C_D, whose updates depended on the outcome of the drift detection mechanism.

When no drift was detected, C_D was incrementally updated using the current batch, allowing it to accumulate knowledge over time. In contrast, when a drift was detected, the classifier was reinitialized and retrained exclusively on the most recent batch, simulating the replacement of outdated knowledge with a model aligned to the current data distribution.

Because true class labels were available for all batches, performance metrics were computed after each update or retraining step. This enabled a continuous assessment of classifier behavior under both drift-aware and non-drift-aware strategies across the entire data stream.

6.2 Experimental Protocol

The experimental evaluation was designed to assess whether retraining classifiers based on detected drifts leads to improved predictive performance under non-stationary data conditions. The drift detection techniques described in Section 2.3.3 — HDDDM, JSDDM, and KSDDM — were applied to the datasets presented in Section 5.1, and classifier performance was evaluated throughout the learning process.

Given the heterogeneous nature of the datasets and the presence of class imbalance in several scenarios, the Area Under the Curve (AUC) and F1-score (Section 2.2) were selected as the primary evaluation metrics. These measures provide a robust assessment of classification performance in drift-affected environments.

To evaluate these techniques, the datasets were segmented into batches of 1000, 1500, 2000, and 2500 instances. These batch sizes were selected to balance dataset size constraints while ensuring a sufficient number of batches for a robust evaluation.

Since the datasets vary considerably in scale, smaller and standardized batch configurations were adopted to enable consistent comparisons across multiple scenarios and to preserve a sufficient number of evaluation points. As batch sizes were fixed, the total number of batches naturally varies across datasets according to their original sizes. The same batch sizes were maintained in the multi-run experiments to ensure methodological consistency between both analyses.

The experimental evaluation was structured into two main experiments. Both experiments analyze whether retraining a classifier when drift is detected leads to improved predictive performance compared to a strategy that does not incorporate drift-aware retraining. The first experiment uses a combination of real-world datasets with concept drift and selected synthetic datasets with controlled drift scenarios. The second experiment focuses exclusively on synthetic datasets and consists of 30 independent repetitions with different random seeds, enabling a statistically robust assessment of the observed results.

6.2.1 Multi-Dataset Experiment

The objective of this experiment is to evaluate whether monitoring input data and retraining classifiers when drift is detected can improve performance in environments affected by both data drift and concept drift.

This experiment uses a combination of real-world and synthetic datasets, all described in Section 5.1. The evaluated datasets are:

- **Insects**: Real-world dataset containing multiple concept drift scenarios.
- **SEA**: Synthetic dataset designed to simulate concept drift.
- **STAGGER**: Synthetic dataset with controlled concept drift patterns.
- **Electricity**: Real-world dataset exhibiting concept drift over time.
- **Magic Gamma Telescope**: Dataset modified to simulate data drift by sorting the `fConc1` feature.

- **Synthetic datasets**: Artificially generated datasets modeling different data drift scenarios.

With the exception of the Magic Gamma Telescope and the synthetic datasets, all real-world datasets exhibit concept drift. The Magic Gamma Telescope dataset was originally generated using Monte Carlo simulations to model gamma particle detection in atmospheric Cherenkov telescopes (Bock, 2007). Since the original dataset exhibits minimal drift, it was modified by sorting the `fConc1` feature in ascending order and creating incremental batches with meaningful data drift, following the methodology proposed by Ditzler and Polikar (2011).

6.2.2 Multi-Run Synthetic Experiments

The second experiment focuses exclusively on synthetic datasets and aims to provide statistically robust evidence of the impact of drift-aware retraining strategies.

Five synthetic datasets were used, each modeling a specific drift scenario:

- **SYN:** Control dataset with no drift.
- **SYN-PA:** Parallel abrupt drifts.
- **SYN-PI:** Parallel incremental drifts.
- **SYN-SA:** Switching abrupt drifts.
- **SYN-SI:** Switching incremental drifts.

Each dataset was generated 30 times using different random seeds, resulting in 30 independent runs per drift scenario. This design enables a statistically meaningful comparison of classifier performance under varying drift conditions while controlling for randomness in data generation.

In addition to evaluating predictive performance, this experiment also provides insights into the computational behavior of the drift detection techniques, including execution time and stability across repeated runs.

6.3 Results and Discussion

This section presents and discusses the experimental results obtained from the (re)training of Naive Bayes classifiers using the datasets introduced in Section 5.1 and the evaluated drift detection techniques. It analyzes the performance of the classifiers and the characteristics of the drift detection techniques that may impact them. The discussion is organized into two main parts: Section 6.3.1, which examines the results from experiments across various datasets, and Section 6.3.2, which presents insights derived from multiple runs of experiments on synthetic datasets with varying random seeds. The detailed results are reported in Tables 6.1-6.8.

6.3.1 Multi-Dataset Experiment

Tables 6.1-6.4 present the performance metrics obtained in the Multi-Dataset Experiment. Dataset names in these tables use the following abbreviations: **Abr** (Abrupt), **Grad** (Gradual), **Inc** (Incremental), **Rec** (Reoccurring), **Bal** (Balanced), and **Imbal** (Imbalanced), which correspond to different drift and class-balance scenarios of the *Insects* dataset. The remaining abbreviations refer to synthetic datasets: **SYN-PA** (Synthetic Parallel Abrupt), **SYN-PI** (Synthetic Parallel Incremental), **SYN-SA** (Synthetic Switching Abrupt), and **SYN-SI** (Synthetic Switching Incremental).

Regarding the table formatting, the best F1 metrics for each dataset are shown in **bold**, while the best AUC values are in *italics*. Each dataset varies in size, consequently having a specific number of batches. The **Drift** column indicates the number of drifts detected by a technique in that particular dataset scenario. The **Base** column provides the values for classifier C_B, which is not retrained at any batch.

This improvement can be clearly observed in datasets such as *Magic*, where the baseline classifier achieves an F1-score close to 0.50, while drift-aware approaches reach values as high as 0.84 when using smaller batch sizes. Similarly, in the *MULTISTAGGER* dataset, the F1-score increases from approximately 0.49 with the base classifier to about 0.74 when drift detection is applied. These substantial gains highlight the impact of timely drift detection and model retraining on classification performance, particularly in scenarios with frequent or abrupt distribution changes.

Across all techniques, the best overall performance is consistently obtained with a batch size of 1000. For example, the KS90 method achieves a mean F1-score of 0.615 at this batch size, compared to 0.567 when the batch size is increased to 2500. A similar trend is observed for the HD and JS methods, whose mean F1-scores decrease from approximately 0.61 to around 0.56 as the batch size grows. This suggests that smaller batches allow faster detection of distribution changes and more effective adaptation of the classifier.

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
MULTISTAGGER	50	0	0.502	0.555	1	0.557	0.587	1	0.557	0.587	11	**0.841**	*0.837*	11	**0.841**	*0.837*
MULTISEA	50	0	0.687	0.651	4	0.691	0.653	4	0.691	0.653	6	**0.692**	*0.655*	6	**0.692**	*0.655*
SEA	50	0	**0.690**	0.659	4	**0.690**	*0.662*	6	**0.690**	0.661	4	**0.690**	0.660	4	**0.690**	0.660
STAGGER	50	0	0.576	0.529	0	0.576	0.529	1	0.781	0.713	12	**0.864**	*0.836*	12	**0.864**	*0.836*
Electricity	45	0	0.452	0.508	45	**0.559**	*0.556*	45	**0.559**	*0.556*	5	0.481	0.518	7	0.485	0.520
Magic	19	0	0.518	0.505	19	**0.964**	*0.950*	19	**0.964**	*0.950*	3	0.907	0.873	3	0.907	0.873
Inc (Bal)	57	0	0.500	0.487	15	**0.581**	0.454	17	**0.581**	0.454	1	0.500	*0.487*	1	0.500	*0.487*
Inc (Imbal)	452	0	0.344	*0.552*	47	**0.520**	0.514	68	**0.520**	0.513	22	0.519	0.521	23	0.519	0.520
Abr (Bal)	79	0	0.458	0.564	43	0.522	0.571	46	**0.527**	0.558	19	0.464	*0.593*	19	0.464	*0.593*
Abr (Imbal)	452	0	0.383	*0.585*	93	**0.501**	0.530	102	**0.501**	0.528	50	0.493	0.532	51	0.492	0.533
Inc-Grad (Bal)	24	0	0.325	*0.473*	10	**0.546**	0.349	10	**0.546**	0.349	5	0.517	0.354	5	0.517	0.354
Inc-Grad (Imbal)	143	0	0.466	*0.474*	21	0.550	0.440	26	**0.552**	0.437	10	0.548	0.437	11	0.548	0.436
Inc-Abr-Rec (Bal)	52	0	0.498	0.438	19	**0.581**	0.404	21	0.579	0.406	9	0.525	0.442	10	0.512	*0.464*
Inc-Abr-Rec (Imbal)	355	0	0.522	*0.544*	41	0.548	0.513	53	0.547	0.512	19	**0.552**	0.512	19	**0.552**	0.511
Inc-Rec (Bal)	79	0	0.335	*0.571*	42	0.519	0.527	45	**0.523**	0.520	17	0.481	0.568	19	0.475	0.566
Inc-Rec (Imbal)	452	0	0.419	*0.584*	90	0.500	0.528	103	**0.501**	0.527	52	0.499	0.532	52	0.498	0.532
SYN	80	0	**0.663**	*0.665*	3	0.661	0.663	5	0.661	0.664	7	0.662	0.664	3	0.659	0.662
SYN-PA	80	0	0.634	0.642	8	0.650	0.653	10	**0.653**	*0.655*	4	0.636	0.646	4	0.636	0.646
SYN-PI	80	0	0.653	0.657	12	0.659	0.662	14	**0.662**	*0.665*	4	0.653	0.660	4	0.653	0.660
SYN-SA	80	0	0.645	0.649	21	**0.663**	*0.664*	21	**0.663**	*0.664*	16	0.657	0.661	16	0.657	0.661
SYN-SI	80	0	0.660	0.663	30	**0.666**	*0.669*	30	**0.666**	*0.669*	17	0.665	0.668	16	0.663	0.666

Table 6.1: *F1 and AUC metrics for the classifier with a batch size of 1000.*

According to the experimental results, employing drift detection techniques to guide

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
MULTISTAGGER	33	0	0.502	0.555	0	0.502	0.555	0	0.502	0.555	4	**0.767**	*0.758*	4	**0.767**	*0.758*
MULTISEA	33	0	0.687	0.651	3	0.685	0.651	5	0.686	0.656	6	**0.693**	*0.658*	6	**0.693**	*0.658*
SEA	33	0	**0.691**	0.660	2	0.690	0.659	4	0.689	0.661	7	0.690	*0.662*	7	0.690	*0.662*
STAGGER	33	0	0.574	0.528	0	0.574	0.528	0	0.574	0.528	5	**0.840**	*0.800*	5	**0.840**	*0.800*
Electricity	30	0	0.438	0.508	30	**0.526**	*0.536*	30	**0.526**	*0.536*	4	0.450	0.510	10	0.462	0.512
Magic	12	0	0.517	0.504	12	**0.937**	*0.913*	12	**0.937**	*0.913*	1	0.860	0.814	1	0.860	0.814
Inc (Bal)	38	0	0.497	*0.488*	11	0.574	0.468	12	**0.575**	0.469	9	**0.575**	0.468	9	**0.575**	0.468
Inc (Imbal)	301	0	0.345	*0.552*	33	**0.521**	0.517	45	0.518	0.515	16	0.512	0.527	16	0.512	0.527
Abr (Bal)	53	0	0.445	0.570	38	**0.464**	0.591	39	**0.464**	0.591	13	0.402	*0.608*	13	0.402	*0.608*
Abr (Imbal)	301	0	0.382	*0.585*	89	**0.490**	0.534	103	**0.490**	0.533	46	0.487	0.544	46	0.482	0.542
Inc-Grad (Bal)	16	0	0.317	*0.476*	8	**0.531**	0.369	8	**0.531**	0.369	3	0.439	0.335	3	0.439	0.335
Inc-Grad (Imbal)	95	0	0.464	*0.475*	18	0.543	0.437	21	0.542	0.437	10	**0.545**	0.442	11	**0.545**	0.440
Inc-Abr-Rec (Bal)	35	0	0.492	0.440	17	0.550	0.439	21	**0.554**	0.437	6	0.496	*0.466*	6	0.496	*0.466*
Inc-Abr-Rec (Imbal)	236	0	0.521	*0.545*	36	**0.546**	0.512	46	0.543	0.513	16	0.543	0.515	14	0.543	0.513
Inc-Rec (Bal)	53	0	0.321	0.578	36	**0.478**	0.544	39	0.476	0.543	10	0.402	*0.585*	9	0.393	0.585
Inc-Rec (Imbal)	301	0	0.417	*0.585*	83	0.491	0.539	97	**0.491**	0.536	41	0.480	0.543	43	0.481	0.544
SYN	53	0	0.663	0.665	2	**0.665**	*0.667*	4	0.664	0.666	5	0.664	0.666	4	0.662	0.664
SYN-PA	53	0	0.634	0.642	8	0.644	0.649	10	**0.649**	*0.653*	8	0.639	0.647	7	0.639	0.646
SYN-PI	53	0	0.652	0.656	9	0.657	0.661	11	**0.661**	*0.664*	7	0.654	0.659	6	0.654	0.659
SYN-SA	53	0	0.644	0.648	23	0.656	0.658	24	**0.657**	*0.659*	13	0.655	0.657	13	0.655	0.657
SYN-SI	53	0	0.660	0.663	24	0.664	*0.667*	25	**0.665**	*0.667*	15	0.664	0.666	15	0.664	0.666

Table 6.2: *F1 and AUC metrics for the classifier with a batch size of 1500.*

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
MULTISTAGGER	25	0	0.493	0.549	0	0.493	0.549	0	0.493	0.549	1	**0.550**	*0.582*	1	**0.550**	*0.582*
MULTISEA	25	0	0.687	0.651	1	0.687	0.651	7	**0.693**	*0.658*	1	0.686	0.651	1	0.686	0.651
SEA	25	0	**0.690**	0.659	1	**0.690**	0.659	5	0.689	*0.661*	1	**0.690**	*0.661*	1	**0.690**	*0.661*
STAGGER	25	0	0.573	0.528	0	0.573	0.528	0	0.573	0.528	3	**0.795**	*0.731*	3	**0.795**	*0.731*
Electricity	22	0	0.438	0.507	22	**0.519**	*0.535*	22	**0.519**	*0.535*	6	0.449	0.512	7	0.480	0.520
Magic	9	0	0.517	0.504	9	**0.909**	*0.875*	9	**0.909**	*0.875*	2	0.814	0.760	1	0.708	0.650
Inc (Bal)	28	0	0.496	0.489	12	0.574	0.468	13	**0.575**	0.468	2	0.502	*0.491*	7	0.571	0.472
Inc (Imbal)	226	0	0.344	*0.553*	41	0.515	0.522	53	**0.516**	0.521	18	0.508	0.525	17	0.508	0.529
Abr (Bal)	39	0	**0.435**	0.578	31	0.418	0.642	31	0.418	0.642	14	0.348	*0.643*	14	0.348	*0.643*
Abr (Imbal)	226	0	0.382	*0.585*	84	0.481	0.536	93	**0.484**	0.537	40	0.476	0.547	44	0.475	0.545
Inc-Grad (Bal)	12	0	0.296	*0.484*	8	**0.454**	0.355	8	**0.454**	0.355	3	0.425	0.328	3	0.425	0.328
Inc-Grad (Imbal)	71	0	0.463	*0.476*	19	0.541	0.437	21	**0.546**	0.441	11	0.528	0.446	11	0.528	0.446
Inc-Abr-Rec (Bal)	26	0	**0.486**	0.440	13	0.477	*0.452*	13	0.477	*0.452*	6	0.478	0.449	6	0.478	0.449
Inc-Abr-Rec (Imbal)	177	0	0.521	*0.544*	35	**0.545**	0.512	45	0.543	0.511	15	0.531	0.518	15	0.531	0.518
Inc-Rec (Bal)	39	0	0.312	0.588	33	0.469	0.560	35	**0.471**	0.547	10	0.391	0.606	15	0.412	*0.615*
Inc-Rec (Imbal)	226	0	0.417	*0.585*	85	**0.484**	0.541	89	**0.484**	0.540	47	0.479	0.545	47	0.479	0.546
SYN	40	0	0.662	0.665	2	**0.665**	*0.667*	2	**0.665**	0.666	1	0.663	0.665	1	0.663	0.665
SYN-PA	40	0	0.634	0.642	7	0.644	0.649	8	**0.648**	*0.651*	5	0.638	0.644	5	0.638	0.644
SYN-PI	40	0	0.653	0.657	7	0.659	0.662	8	**0.661**	*0.664*	5	0.653	0.658	5	0.653	0.658
SYN-SA	40	0	0.642	0.645	19	**0.650**	*0.652*	20	**0.650**	*0.652*	13	**0.650**	*0.652*	13	**0.650**	*0.652*
SYN-SI	40	0	0.660	0.662	19	**0.663**	*0.666*	20	**0.663**	*0.666*	11	0.662	0.665	10	0.662	0.664

Table 6.3: *F1 and AUC metrics for the classifier with a batch size of 2000.*

Dataset	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
MULTISTAGGER	20	0	0.493	0.549	0	0.493	0.549	0	0.493	0.549	2	**0.744**	*0.736*	2	**0.744**	*0.736*
MULTISEA	20	0	0.687	0.651	0	0.687	0.651	8	**0.693**	*0.659*	4	0.690	0.655	4	0.690	0.655
SEA	20	0	**0.690**	0.659	2	0.689	0.661	4	0.689	0.660	3	0.689	*0.661*	3	0.689	*0.661*
STAGGER	20	0	0.574	0.529	0	0.574	0.529	0	0.574	0.529	3	**0.829**	*0.785*	3	**0.829**	*0.785*
Electricity	18	0	0.435	0.506	18	**0.520**	*0.535*	18	**0.520**	*0.535*	7	0.495	0.525	7	0.487	0.521
Magic	7	0	0.514	0.502	7	**0.849**	*0.801*	7	**0.849**	*0.801*	2	**0.849**	*0.801*	1	0.660	0.608
Inc (Bal)	22	0	0.494	*0.486*	13	**0.572**	0.470	13	**0.572**	0.470	0	0.494	*0.486*	0	0.494	*0.486*
Inc (Imbal)	180	0	0.343	*0.553*	41	**0.509**	0.523	47	**0.509**	0.520	19	0.498	0.534	19	0.498	0.534
Abr (Bal)	31	0	**0.424**	0.587	27	0.374	*0.646*	28	0.373	0.645	6	0.288	0.627	6	0.288	0.627
Abr (Imbal)	180	0	0.380	*0.585*	79	**0.474**	0.545	85	0.473	0.543	27	0.449	0.538	28	0.449	0.538
Inc-Grad (Bal)	9	0	0.286	*0.489*	6	**0.463**	0.347	6	**0.463**	0.347	3	0.442	0.410	3	0.442	0.410
Inc-Grad (Imbal)	57	0	0.461	*0.477*	15	0.540	0.441	18	**0.545**	0.434	8	0.529	0.463	8	0.529	0.463
Inc-Abr-Rec (Bal)	21	0	**0.483**	0.439	14	0.442	*0.471*	14	0.442	*0.471*	7	0.451	0.452	7	0.451	0.452
Inc-Abr-Rec (Imbal)	142	0	0.520	*0.545*	31	**0.533**	0.522	32	**0.533**	0.522	15	0.514	0.525	16	0.516	0.530
Inc-Rec (Bal)	31	0	0.309	*0.596*	26	**0.429**	0.594	26	**0.429**	0.594	6	0.375	0.580	6	0.375	0.580
Inc-Rec (Imbal)	180	0	0.415	*0.585*	79	0.471	0.546	85	**0.474**	0.546	36	0.467	0.552	36	0.467	0.552
SYN	32	0	**0.662**	*0.664*	1	0.661	0.663	1	0.661	0.663	1	0.660	0.663	2	0.660	0.662
SYN-PA	32	0	0.633	0.641	7	**0.638**	*0.644*	7	**0.638**	*0.644*	4	0.620	0.636	4	0.620	0.636
SYN-PI	32	0	0.653	0.656	7	**0.654**	*0.658*	7	**0.654**	*0.658*	2	0.650	0.656	2	0.650	0.656
SYN-SA	32	0	0.640	0.644	17	**0.650**	*0.652*	17	**0.650**	*0.652*	4	0.643	0.647	4	0.643	0.647
SYN-SI	32	0	0.660	0.662	18	**0.664**	*0.666*	18	**0.664**	*0.666*	10	0.661	0.663	10	0.663	0.665

Table 6.4: *F1 and AUC metrics for the classifier with a batch size of 2500.*

retraining decisions can lead to improved classifier performance in several scenarios. In most datasets, the highest F1-scores are achieved when drift detection mechanisms are used, rather than by the base classifier without monitoring. This indicates that incorporating input monitoring and drift-aware retraining strategies often contributes to better adaptation under non-stationary conditions.

However, the results also show that such improvements are not consistent across all datasets and configurations. In some cases, the base classifier achieves comparable or superior performance, suggesting that the effectiveness of drift-based retraining depends on factors such as the type of drift, dataset characteristics, and batch configuration. Therefore, while drift detection can be a valuable tool for enhancing model performance, its benefits are context-dependent rather than universal.

The use of smaller batches consistently provided the best results in terms of the F1 and AUC metrics. In terms of the number of detected drifts for each technique across the batches of each experiment, the KS90 technique triggered more resets to the classifiers than the other techniques, closely followed by KS95. The HDDDM and JSDDM techniques presented similar results, triggering considerably fewer classifier resets than the KS techniques.

The KS Test is notably sensitive to data shifts, as highlighted in Sections 5.2.1 and 5.2.2. While this sensitivity can, in some contexts (e.g., highly stable distributions like MULTISTAGGER), lead to an overfitted classifier with potentially lower generalization, our aggregate findings across diverse datasets suggest an overall superior performance. Conversely, the adaptive threshold of HDDDM and JSDDM techniques makes them more sensitive to drifts over time, which proved beneficial for datasets with more stable distributions, such as MULTISTAGGER. Their detected drifts and consequent full retraining of the model resulted in F1 and AUC metrics that were better than those of the KSDDM method in

this specific case. It is also worth noting that the replacement of the Hellinger Distance by the JS Divergence in the HDDDM method, which resulted in the JSDDM framework, did not yield improvements in HDDDM performance.

For the SYN datasets, the concepts exhibit greater stability, leading to more consistent and accurate predictions by the classifier. This stability, coupled with the performance of the drift detection techniques, results in balanced and non-excessive model retraining, thereby enhancing the overall robustness of the system. Additionally, for the SYN datasets with drifts, the best performance was observed when using the KS techniques for monitoring.

Our experimental results demonstrate that the analyzed drift detection techniques contribute to improving the system's robustness, even in the presence of concept drift or data drift. Despite using prequential evaluation, which trains the classifier incrementally on past data and tests it on the current batch, the strategy of resetting the classifier was found to consistently improve performance when drifts occurred.

Overall Performance Summary

To summarize the overall performance, Figures 6.1 and 6.2 present the mean F1 and AUC scores, respectively, aggregated across all datasets and batch sizes. The error bars indicate the variability observed across different batch sizes, providing a notion of dispersion in the results.

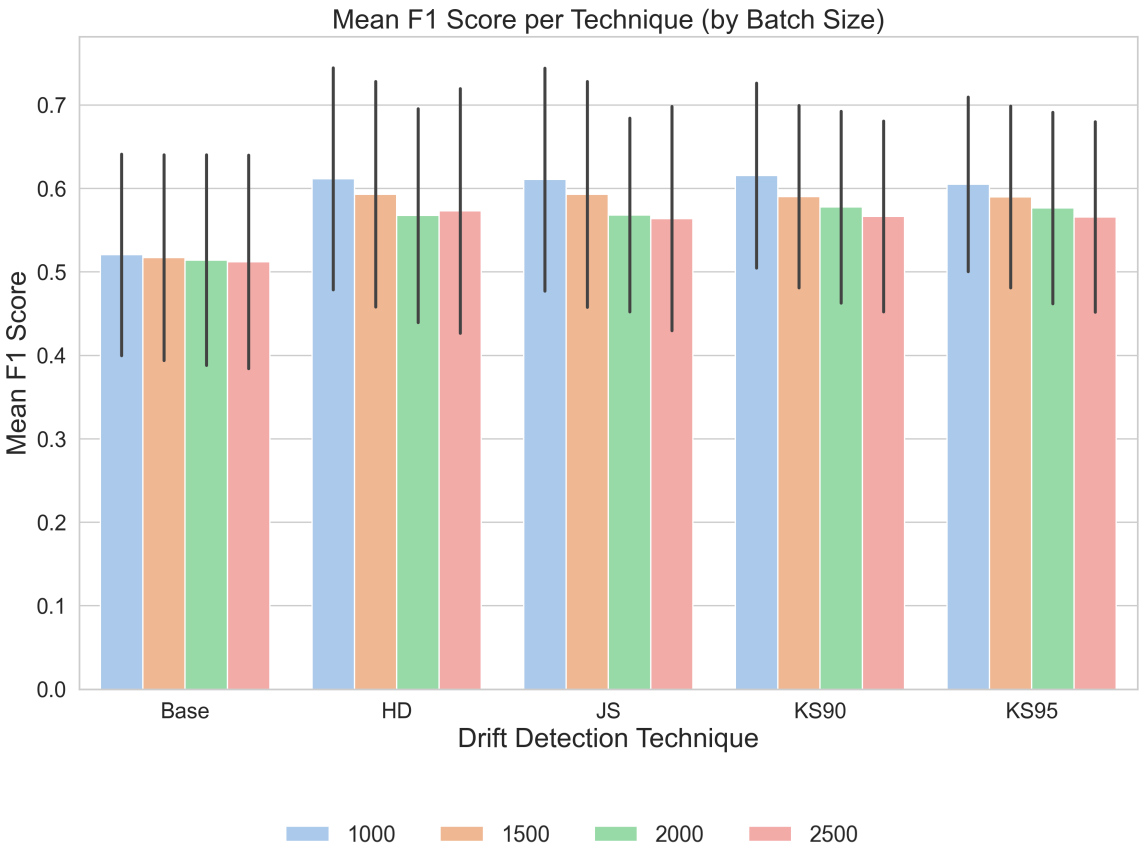


Figure 6.1: *Mean classisifier F1-score for all batch sizes across all datasets.*

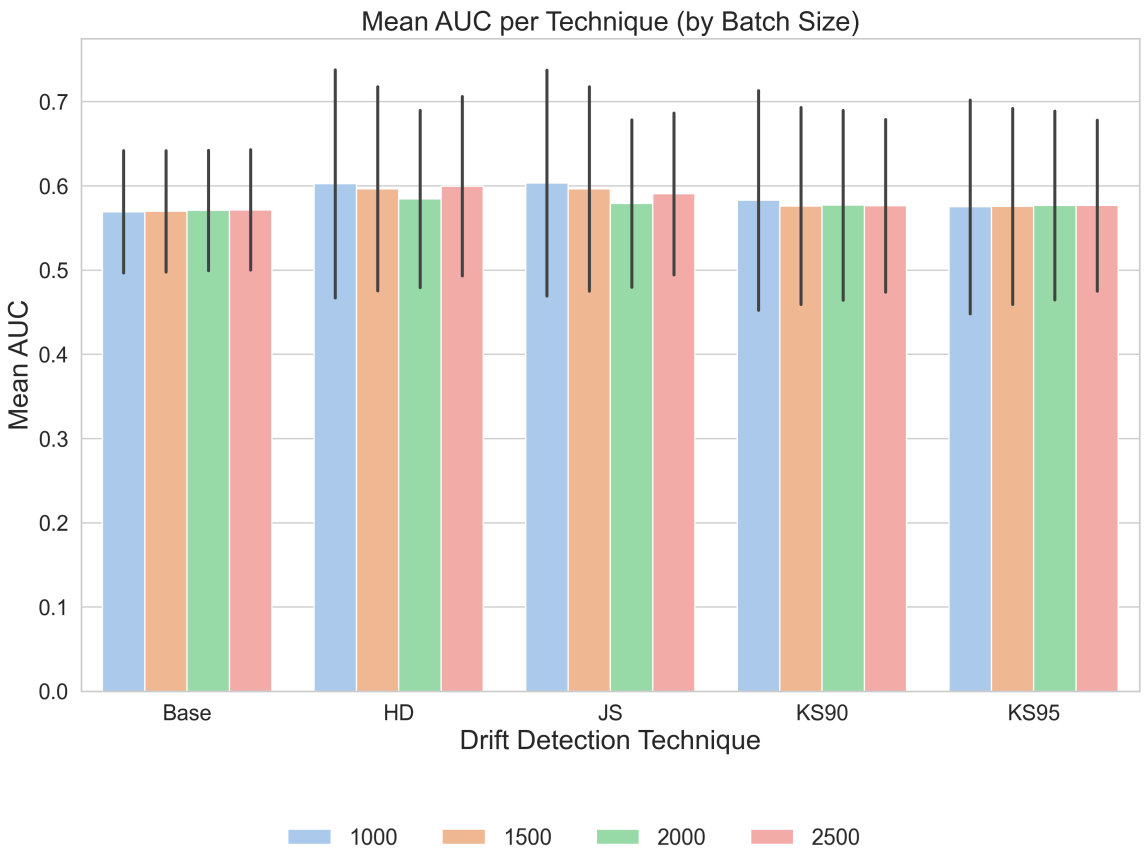


Figure 6.2: *Mean classifier AUC for all batch sizes across all datasets.*

For the F1-score, all drift-aware approaches (HD, JS, KS90, and KS95) consistently outperform the *Base* classifier, which does not employ drift detection. The *Base* method achieves mean F1 values around 0.51–0.52 across all batch sizes, whereas the drift-aware techniques reach values between approximately 0.56 and 0.62. The highest mean F1-score is observed for KS90 with a batch size of 1000 (0.615), closely followed by HD (0.612) and JS (0.611). As the batch size increases, a gradual decline in F1 performance is observed for all techniques, suggesting that larger batches may delay drift detection and adaptation.

Regarding AUC, the improvements over the *Base* classifier are more modest. The *Base* approach achieves stable AUC values around 0.57 across all batch sizes. HD and JS yield slightly higher AUC scores, particularly for smaller batch sizes, reaching approximately 0.60–0.61. In contrast, the KS-based methods (KS90 and KS95) exhibit AUC values closer to the baseline, with mean scores around 0.57–0.58. This indicates that, while KS-based techniques provide noticeable gains in F1, their advantage in terms of AUC is limited under the evaluated conditions.

Overall, no single drift detection method consistently dominates across all metrics and batch sizes. HD and JS tend to provide stronger AUC improvements, whereas KS90 achieves the highest F1-score for smaller batches. The relatively small differences between techniques suggest that their effectiveness is broadly comparable, and that performance is influenced more strongly by batch size and dataset characteristics than by the specific

statistical test used for drift detection.

6.3.2 Multi-Run Synthetic Experiments

The experiments for synthetic datasets were executed 30 times to gather more robust insights into the performance of the techniques. As the results for real-world datasets are deterministic, it was important to validate the stability of results by switching random seeds for the synthetic datasets.

Figure 6.4 summarizes the distribution of the mean F1-scores obtained across these repetitions. Each point corresponds to the average F1-score computed over the 30 executions performed with the same experimental parameters, changing the random seed for the dataset generation. The colors indicate the batch size, and each boxplot represents a different drift detection technique. The narrower dispersion observed for the KS-based methods suggests a more consistent performance across dataset variants, indicating that these techniques tend to be less sensitive to variations in the data generation process.

Figure 6.3 depicts the relationship between the average F1-score and the average number of drift detections. It is evident that a higher number of drift detections does not necessarily lead to an improved F1-score. However, the figure clearly shows that the KS-based techniques are indeed the most sensitive for drift detection.

Regarding the Tables 6.5-6.8, the formatting follows the pattern established in previous sections: **bold** values represent the best average F1 metrics for each dataset, while *italic* values denote the best average AUC. Each dataset has a different size and, consequently, a specific number of batches. This time, the **Drift** column indicates the average number of drifts a technique detected within that specific dataset scenario. The **Base** column provides the values for the classifier C_B, which remains untrained across all batches.

Both images demonstrate that using smaller batch sizes can yield favorable results, though often accompanied by a higher number of drift detections. We can infer that this outcome is highly dependent on the nature of the datasets. Based on the table results, for the Abrupt datasets, KS techniques performed well with smaller batch sizes but were outperformed by the base classifier on larger datasets. This suggests that the selection of a drift detection technique should be carefully aligned with the inherent characteristics of the datasets used by the system.

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
NO-DRIFTS	80	0.00	**0.752**	*0.754*	4.52	0.751	0.753	8.74	0.750	0.752	4.71	0.750	0.752	4.65	0.751	0.753
PARALLEL-ABRUPT	80	0.00	0.734	0.736	9.87	**0.741**	*0.743*	13.68	0.741	0.743	4.06	0.736	0.738	4.03	0.736	0.738
PARALLEL-INCREMENTAL	80	0.00	**0.748**	*0.750*	13.71	0.748	0.750	17.32	0.747	0.749	5.97	0.747	0.749	5.52	0.747	0.749
SWITCHING-ABRUPT	80	0.00	0.717	0.719	22.94	**0.729**	*0.731*	25.90	0.728	0.730	16.39	0.723	0.725	16.29	0.723	0.725
SWITCHING-INCREMENTAL	80	0.00	**0.743**	*0.745*	30.42	0.742	0.744	33.45	0.742	0.744	13.52	0.741	0.743	13.48	0.741	0.743

Table 6.5: *Average F1 and AUC metrics for the classifier with a batch size of 1000.*

6.3.3 Execution Time Analysis

Figures 6.5 and 6.6 present the box-plots of the drift detection execution times per batch size in the performed experiments. The figures show a consistent trend across

Figure 6.3: *Scatter plot of the average F1-scores and the average number of detected drifts by each technique and batch size.*

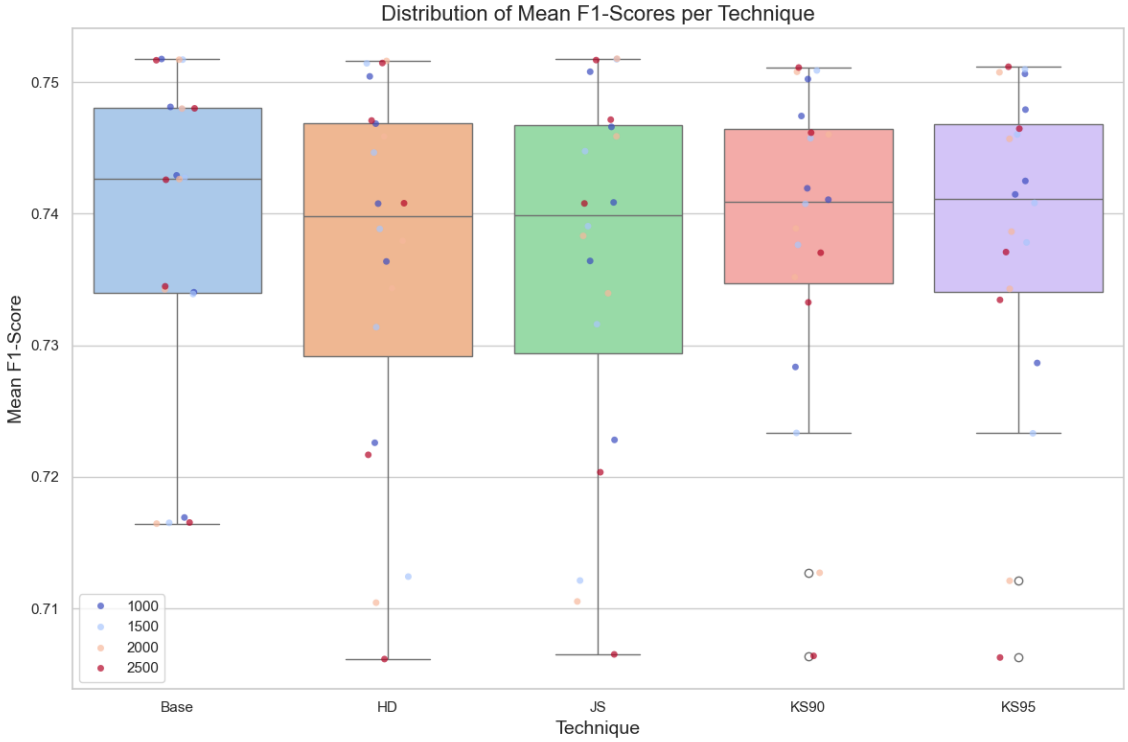


Figure 6.4: *Distribution of mean F1-scores per technique from the multi-run experiment.*

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
NO-DRIFTS	53	0.00	0.752	0.754	2.68	0.751	0.753	4.97	0.751	0.753	3.90	0.751	0.753	3.90	**0.752**	*0.754*
PARALLEL-ABRUPT	53	0.00	0.734	0.736	9.23	**0.738**	*0.740*	11.68	0.738	0.740	6.19	0.731	0.733	5.81	0.732	0.734
PARALLEL-INCREMENTAL	53	0.00	**0.748**	*0.750*	9.94	0.746	0.748	12.45	0.746	0.748	6.26	0.745	0.747	6.26	0.745	0.747
SWITCHING-ABRUPT	53	0.00	0.717	0.719	24.42	0.723	0.725	25.90	**0.723**	*0.725*	14.23	0.712	0.714	13.97	0.712	0.714
SWITCHING-INCREMENTAL	53	0.00	**0.743**	*0.745*	24.68	0.741	0.743	26.06	0.741	0.743	13.23	0.739	0.741	13.32	0.739	0.741

Table 6.6: *Average F1 and AUC metrics for the classifier with a batch size of 1500.*

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
NO-DRIFTS	40	0.00	0.752	0.754	2.74	0.751	0.753	4.77	0.751	0.753	2.06	0.752	0.754	2.03	**0.752**	*0.754*
PARALLEL-ABRUPT	40	0.00	0.734	0.736	8.03	0.734	0.736	10.26	**0.735**	*0.737*	6.00	0.734	0.736	5.61	0.734	0.736
PARALLEL-INCREMENTAL	40	0.00	**0.748**	*0.750*	8.03	0.746	0.748	10.26	0.746	0.748	5.58	0.746	0.748	5.32	0.746	0.748
SWITCHING-ABRUPT	40	0.00	**0.716**	*0.718*	19.29	0.712	0.714	20.71	0.713	0.715	12.81	0.710	0.712	12.71	0.711	0.713
SWITCHING-INCREMENTAL	40	0.00	**0.743**	*0.745*	19.29	0.739	0.741	20.71	0.739	0.741	11.74	0.738	0.740	11.42	0.738	0.740

Table 6.7: *Average F1 and AUC metrics for the classifier with a batch size of 2000.*

Dataset(*)	Batches	Base			KS95			KS90			HD			JS		
		Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC	Drifts	F1	AUC
NO-DRIFTS	32	0.00	0.752	0.754	1.48	0.751	0.753	3.10	0.751	0.753	1.13	0.751	0.753	1.06	**0.752**	*0.754*
PARALLEL-ABRUPT	32	0.00	**0.734**	*0.736*	6.94	0.733	0.735	8.16	0.733	0.735	4.45	0.722	0.724	4.00	0.720	0.722
PARALLEL-INCREMENTAL	32	0.00	**0.748**	*0.750*	6.94	0.746	0.748	8.16	0.746	0.748	2.32	0.747	0.749	2.16	0.747	0.749
SWITCHING-ABRUPT	32	0.00	**0.717**	*0.719*	16.48	0.706	0.708	17.19	0.706	0.708	6.55	0.706	0.708	6.06	0.707	0.709
SWITCHING-INCREMENTAL	32	0.00	**0.743**	*0.745*	18.26	0.737	0.739	18.97	0.737	0.739	9.52	0.741	0.743	9.52	0.741	0.743

Table 6.8: *Average F1 and AUC metrics for the classifier with a batch size of 2500.*

multiple drift detection techniques: total execution time decreases as batch size increases. Figures 6.7 and 6.8 present the same data but as a violin plot, offering additional insight into the distribution of execution times. Although the observed behavior may appear counterintuitive when considering the per-batch computational complexity, it underscores a critical trade-off inherent to data stream processing. The total execution time of an experiment comprises both the algorithmic processing time and the fixed overhead incurred for each batch. With smaller batch sizes, the greater number of batches required to process the dataset amplifies the cumulative overhead associated with data handling, loop control, and memory operations. As a result, this overhead can dominate the overall execution time, effectively diminishing the influence of per-batch complexity and rendering larger batch sizes more efficient in practice.

Figures 6.9 and 6.10 further substantiate this trade-off by showing a similar performance pattern across a range of drift detection techniques, including HDDDM, JSDDM, and two variants of the Kolmogorov-Smirnov test, KS90 and KS95. The uniformity in observed performance across these distinct statistical methods indicates that the phenomenon is not specific to a single algorithm's core computation but rather a fundamental characteristic of the overarching batch-by-batch data processing framework. This consistent behavior underscores that the efficiency of data stream analysis is often governed by a delicate balance between the algorithmic workload and the logistical overheads of managing a large number of discrete batches, suggesting that optimizing for total processing time may require larger batch sizes despite the higher per-batch cost.

Figure 6.11 provides a summary of the average execution times for the different

drift detection techniques, consolidating a trend observed across all batch sizes. The bar chart shows that JSDDM consistently outperforms HDDDM in average execution speed, albeit by a small margin. While JSDDM is designed to be more statistically stable by mitigating the issue of null bins (where a probability of zero in one distribution can lead to an undefined divergence), the connection between this statistical robustness and a performance advantage is not a direct one. The observed difference in execution speed is more plausibly attributed to the distinct computational primitives involved in the two metrics. JSDDM's core calculation relies on logarithms (from the Kullback-Leibler divergence), whereas HDDDM's calculation involves square root operations. The marginal performance difference is likely a consequence of the relative efficiency of these specific mathematical operations and their implementation within the given software environment, rather than a direct benefit derived from JSDDM's statistical stability.

While this runtime analysis provides an initial perspective on computational costs, further investigation is needed to examine how these factors affect large-scale streaming systems and whether algorithmic optimizations can mitigate them.

Figure 6.5: *Execution time for drift detection as a function of batch size, aggregated across all datasets.*

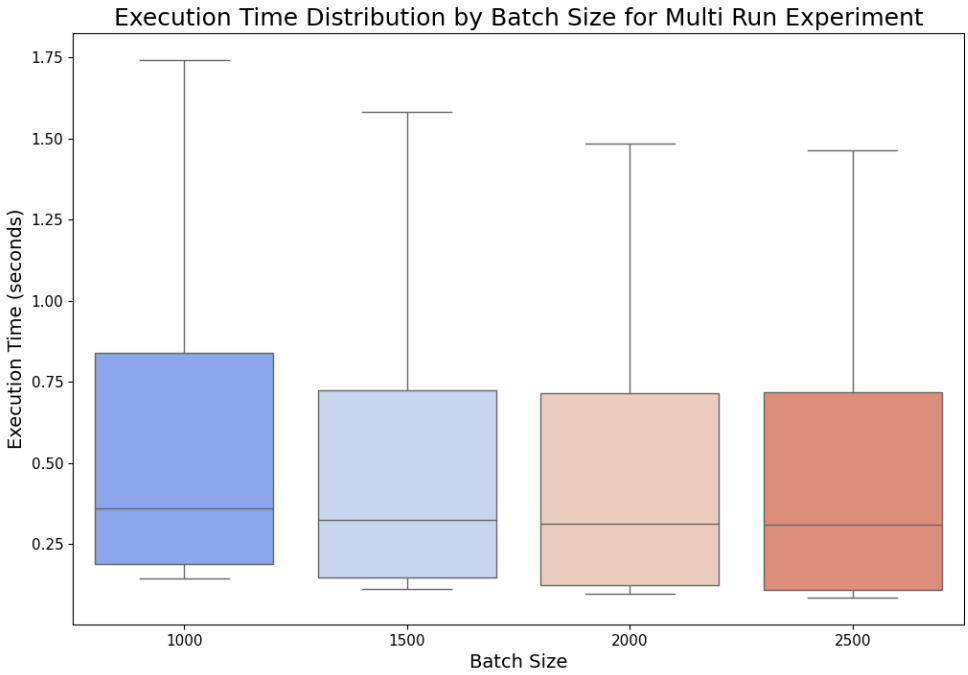


Figure 6.6: *Execution time for drift detection as a function of batch size, for multi run of synthetic datasets.*

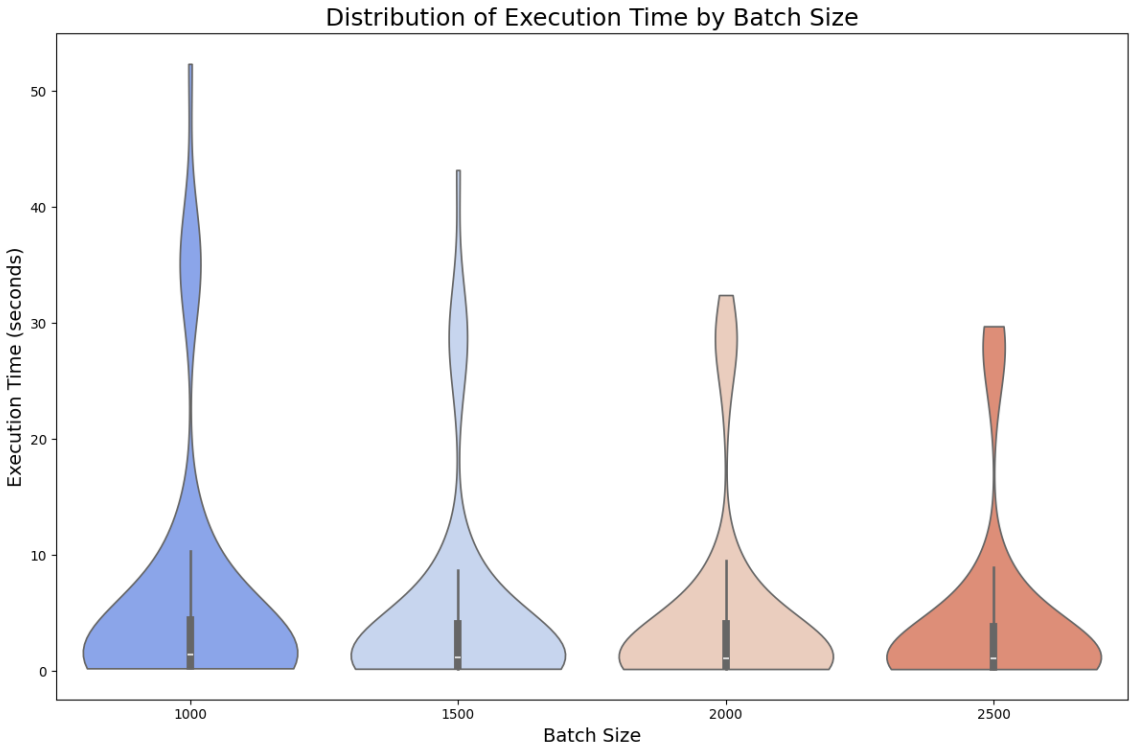


Figure 6.7: *Violin plot of execution time for drift detection as a function of batch size, aggregated across all datasets.*

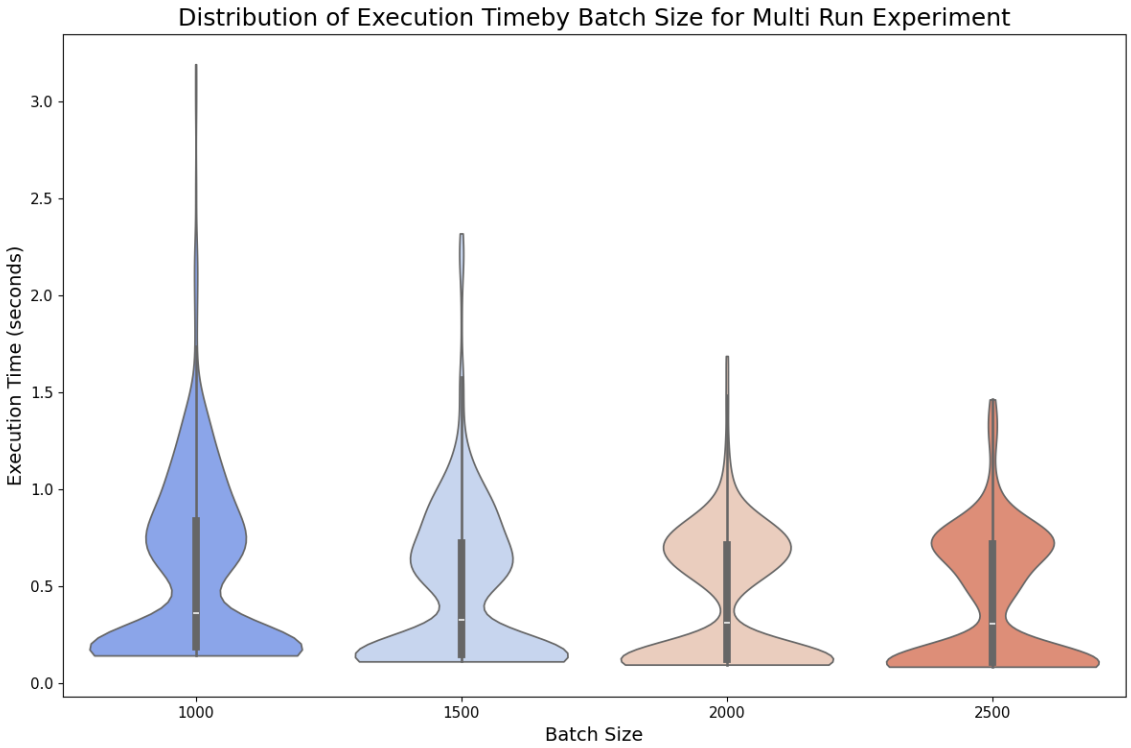


Figure 6.8: *Violin plot of execution time for drift detection as a function of batch size, for multiple runs of synthetic datasets.*

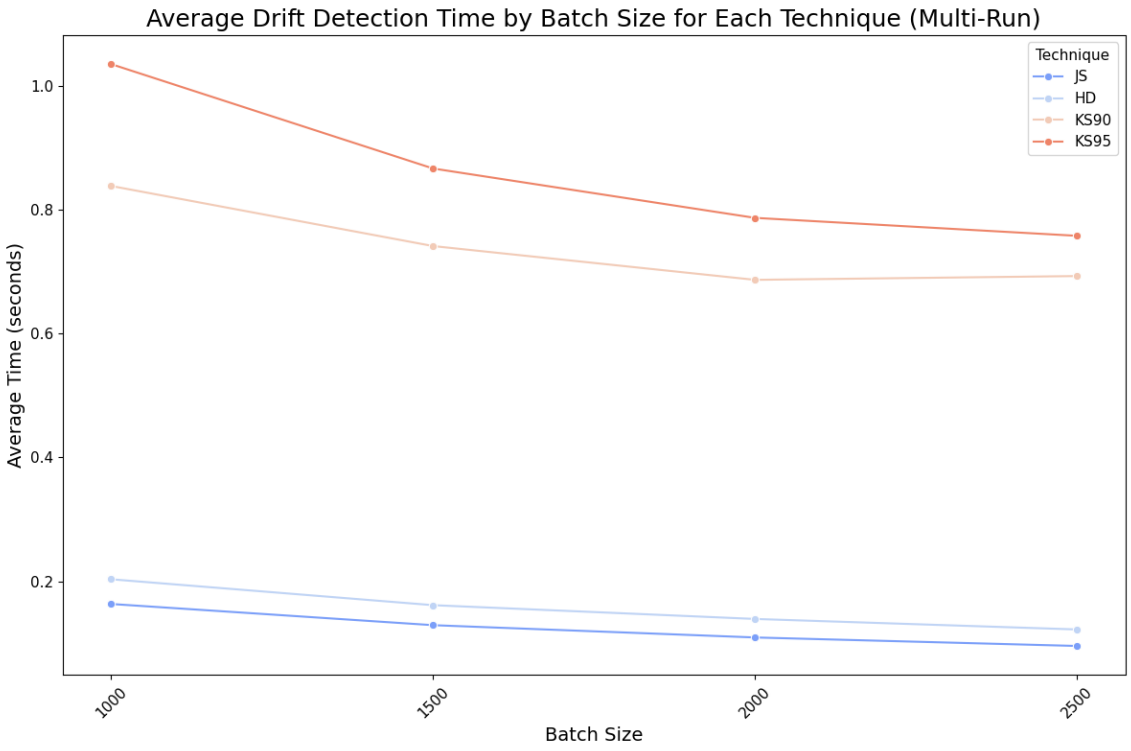


Figure 6.9: *Average drift detection time by batch size for each technique for the multi-run experiment.*

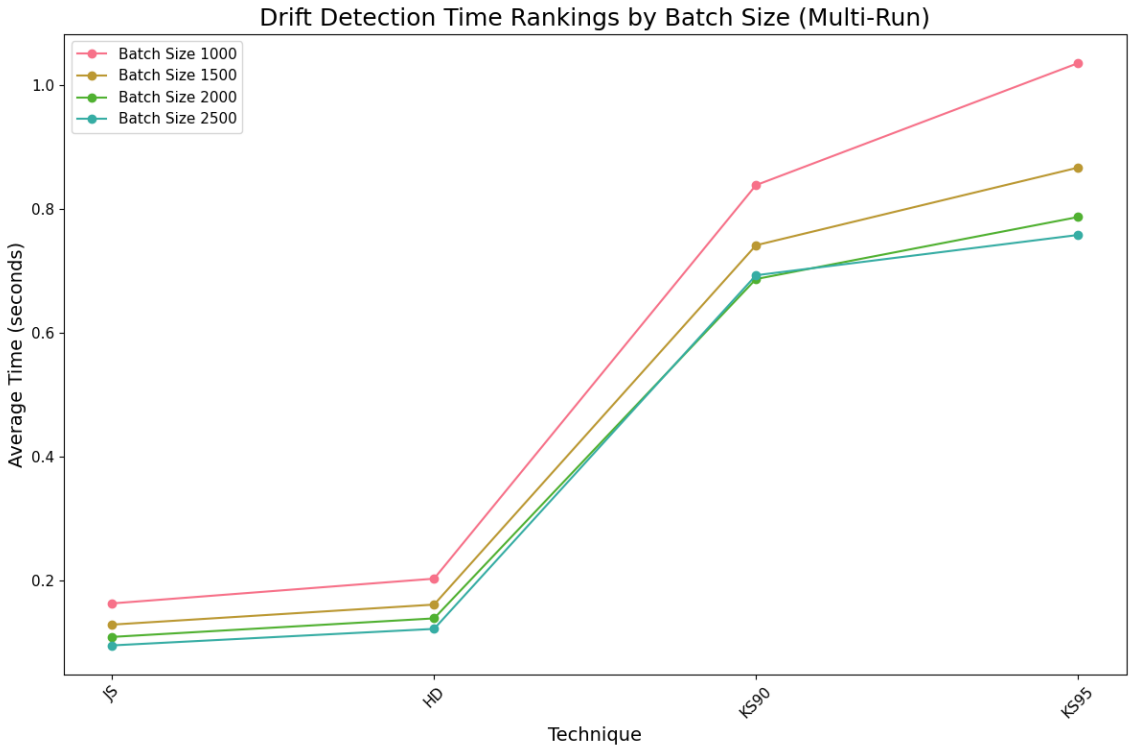


Figure 6.10: *Alternate visualization of the average drift detection time by batch size for each technique for the multi-run experiment.*

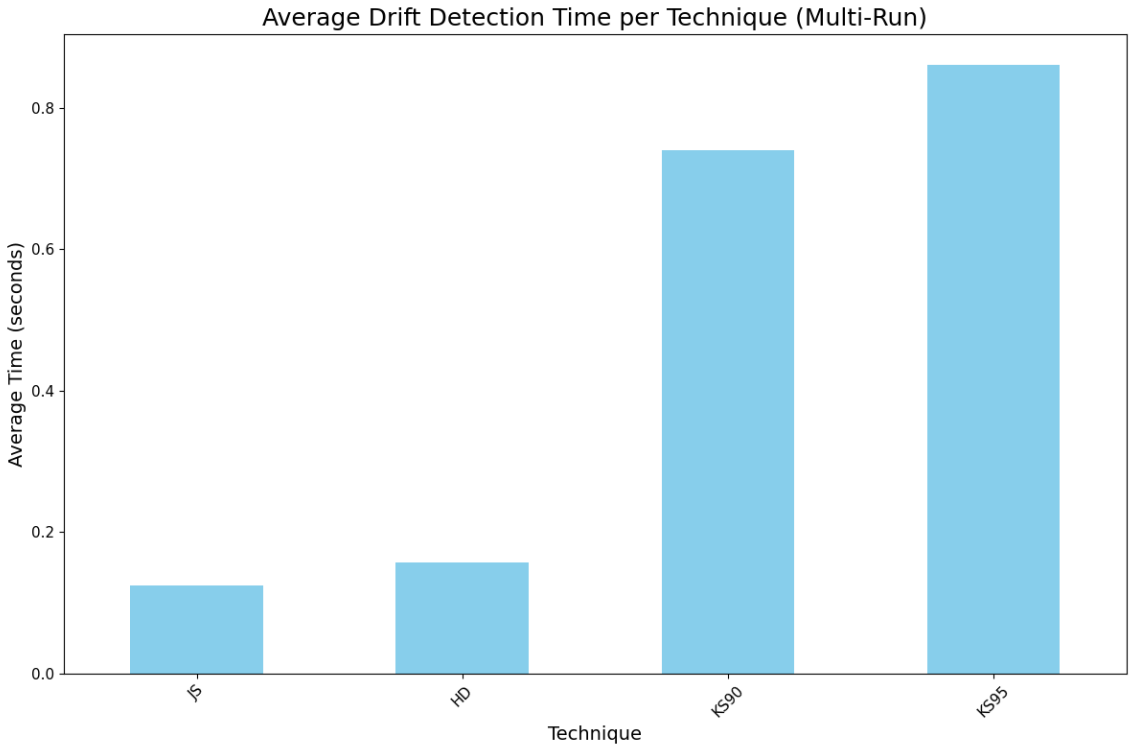


Figure 6.11: *Average drift detection time per technique for the multi-run experiment.*

Chapter 7

Conclusion

This master's thesis investigated the problem of detecting data drift in machine learning systems, focusing on how different drift scenarios impact model behavior and how software engineering principles can support the construction of robust, maintainable, and reliable ML solutions. The work combined an architectural perspective with empirical experimentation to examine both conceptual and practical aspects of drift detection within the machine learning lifecycle.

The first contribution of this thesis is the definition of a theoretical software architecture tailored to the needs of machine learning systems. This architecture organizes the ML lifecycle into interconnected subsystems—data flow, experimentation, training, deployment, monitoring, and feedback—clarifying their responsibilities and communication patterns. It highlights the importance of data versioning, reproducible experimentation, monitoring of input distributions, and explicit feedback loops for model retraining. Accompanying this architecture, a data model was proposed to capture essential metadata used throughout the lifecycle, including dataset versions, model configurations, monitoring signals, and detected drifts.

The second contribution is an empirical analysis of drift detection techniques, conducted using a diverse collection of real, synthetic, and benchmark datasets. The study evaluated a set of distance-based and statistical techniques—such as Jensen–Shannon distance, Kullback–Leibler divergence, Kolmogorov–Smirnov tests, and others—applied to batch windows of data. The experiments compared the sensitivity, stability, and computational implications of each measure under different drift types (e.g., sudden, gradual, incremental). The results demonstrated that although all techniques can identify distributional changes, their responsiveness and robustness vary substantially depending on dataset characteristics. Notably, monitoring input data distributions proved to be an effective and practical strategy for identifying drifts early, strengthening system stability and improving developer confidence when deploying models in dynamic environments.

These findings were validated through peer review and published in the proceedings of the Brazilian Symposium on Databases (SBBD 2024). An extended version of the paper was submitted and accepted for publication in the Journal of Information and Data Management, reinforcing both the scientific relevance and practical applicability of the

conducted analysis.

7.1 Final Remarks

During the development of this thesis, new contributions in the field of Software Engineering for Machine Learning emerged—such as the work of Kreuzberger *et al.* (2023)—which highlight challenges that intersect with those addressed in this research. These publications reinforce the growing importance of architectural thinking, monitoring, and lifecycle management in modern ML systems, supporting the relevance and timeliness of the approach proposed in this master's thesis.

The results presented throughout this thesis emphasize that the reliability of ML systems depends not only on the quality of the trained models but also on the engineering practices surrounding them. As shown in the experiments, data drift can significantly affect model performance, often in ways that are difficult to identify without structured monitoring. Likewise, the proposed architecture demonstrates that integrating experimentation tracking, data lineage, monitoring, and retraining mechanisms contributes to more transparent, maintainable, and resilient ML solutions.

Ultimately, this thesis argues that the intersection of machine learning and software engineering requires holistic solutions: systems must be designed not only to learn from data but also to evolve reliably as data, requirements, and operating environments change.

7.2 Future Work

Several opportunities for extending this research remain open. Future investigations may focus on improving and automating drift detection workflows, creating benchmark suites for evaluating drift detectors under controlled and realistic scenarios, or designing adaptive retraining pipelines that incorporate drift evidence into model selection strategies. Another promising direction is evaluating how architectures similar to the one proposed here behave when supporting more complex ML paradigms—such as multimodal models or systems with stricter latency or safety requirements.

Empirical studies involving practitioners—through surveys, interviews, or field observations—may also provide valuable insights into how drift detection and monitoring tools are adopted in practice, which challenges remain unsolved, and how organizations incorporate these mechanisms into their development processes. Such studies could help guide future research in Software Engineering for Machine Learning toward addressing the most pressing needs of real-world ML teams.

References

[Abu-Mostafa *et al.* 2012] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from Data*. New York: AMLBook, 2012. isbn: 978-1-60049-006-4 (cit. on pp. 5, 6).

[Alves *et al.* 2023] Isaque Alves, Leonardo AF Leite, Paulo Meirelles, Fabio Kon, and Carla Silva Rocha Aguiar. "Practices for managing machine learning products: a multivocal literature review". *IEEE Transactions on Engineering Management* 71 (2023), pp. 7425–7455 (cit. on p. 25).

[Amershi *et al.* 2019] Saleema Amershi *et al.* "Software engineering for machine learning: a case study". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 291–300 (cit. on p. 24).

[*Autopilot* 2026] *Autopilot*. https://aws.amazon.com/sagemaker/autopilot/. (Visited on 01/19/2026) (cit. on p. 20).

[*Azure Machine Learning* 2026] *Azure Machine Learning*. https://azure.microsoft.com/services/machine-learning/. (Visited on 01/19/2026) (cit. on pp. 19, 20, 27, 36).

[Baena-Garcia *et al.* 2006] Manuel Baena-Garcia *et al.* "Early drift detection method". In: *Fourth international workshop on knowledge discovery from data streams*. Vol. 6. Citeseer. 2006, pp. 77–86 (cit. on p. 11).

[Balakrishnan *et al.* 2020] Tara Balakrishnan, Micahel Chui, Bryce Hall, and Nicolaus Henke. "Global survey: The state of AI in 2020". *Mckinsey Analytics* November (2020), p. 13. url: https://www.mckinsey.com/capabilities/quantumblack/our-insights/global-survey-the-state-of-ai-in-2020 (cit. on p. 1).

[Bhatt *et al.* 2020] Umang Bhatt *et al.* "Explainable machine learning in deployment". *FAT* 2020 - Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency* (2020), pp. 648–657. doi: 10.1145/3351095.3375624. arXiv: 1909.06342 (cit. on p. 26).

[Bifet and Gavalda 2007] Albert Bifet and Ricard Gavalda. "Learning from time-changing data with adaptive windowing". In: *Proceedings of the 2007 SIAM international conference on data mining*. SIAM. 2007, pp. 443–448 (cit. on p. 11).

[Bland and Altman 1995] J Martin Bland and Douglas G Altman. "Multiple significance tests: the Bonferroni method". *Bmj* 310.6973 (1995), p. 170 (cit. on p. 11).

[Bock 2007] R. Bock. *MAGIC Gamma Telescope*. https://doi.org/10.24432/C52C8B. 2007 (cit. on pp. 42, 54).

[Breck *et al.* 2017] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. "The ml test score: a rubric for ml production readiness and technical debt reduction". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 1123–1132 (cit. on p. 23).

[Cam *et al.* 2019] Arif Cam, Michael Chui, and Bryce Hall. "Global AI Survey: AI proves its worth, but few scale impact". *McKinsey* November (2019), pp. 1–16 (cit. on p. 1).

[Chen *et al.* 2020] Andrew Chen *et al.* "Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle". *Proceedings of the 4th Workshop on Data Management for End-To-End Machine Learning, DEEM 2020 - In conjunction with the 2020 ACM SIGMOD/PODS Conference* (2020). doi: 10.1145/3399579.3399867 (cit. on pp. 24, 27, 36).

[*Cloud AutoML* 2026] *Cloud AutoML*. https://cloud.google.com/automl. (Visited on 01/19/2026) (cit. on p. 20).

[Dasu *et al.* 2006] Tamraparni Dasu, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi. "An information-theoretic approach to detecting changes in multi-dimensional data streams". In: *Symposium on the Interface of Statistics, Computing Science, and Applications (Interface)*. 2006 (cit. on pp. 11–13, 30).

[*Data Version Control* 2026] *Data Version Control*. https://dvc.org/. (Visited on 01/19/2026) (cit. on pp. 24, 28, 35).

[Ditzler and Polikar 2011] Gregory Ditzler and Robi Polikar. "Hellinger distance based drift detection for nonstationary environments". In: *2011 IEEE symposium on computational intelligence in dynamic and uncertain environments (CIDUE)*. 2011, pp. 41–48 (cit. on pp. 13, 30, 43, 44, 54).

[Domingos 2012] Pedro Domingos. "A few useful things to know about machine learning". *Communications of the ACM* 55.10 (2012), pp. 78–87. doi: 10.1145/2347736.2347755 (cit. on p. 6).

[Dries and Rückert 2009] Anton Dries and Ulrich Rückert. "Adaptive concept drift detection". *Statistical Analysis and Data Mining: The ASA Data Science Journal* 2.5-6 (2009), pp. 311–327 (cit. on p. 10).

[*Elasticsearch* 2026] *Elasticsearch*. https://www.elastic.co/elasticsearch/. (Visited on 01/19/2026) (cit. on p. 24).

[*Ethics guidelines for trustworthy AI* 2026] *Ethics guidelines for trustworthy AI*. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai. (Visited on 01/19/2026) (cit. on pp. 15, 25).

[*Feast* 2026] *Feast*. https://github.com/feast-dev/feast. (Visited on 01/19/2026) (cit. on pp. 29, 35).

[Gama and Castillo 2006] Joao Gama and Gladys Castillo. "Learning with local drift detection". In: *Advanced Data Mining and Applications (ADMA 2006)*. Vol. 4093. Lecture Notes in Computer Science. Springer, 2006, pp. 42–55. doi: 10.1007/11811305_4 (cit. on pp. 1, 9, 29).

[Gama, Medas, *et al.* 2004] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. "Learning with drift detection". In: *Advances in Artificial Intelligence – SBIA 2004*. Vol. 3171. Lecture Notes in Computer Science. Springer, 2004, pp. 286–295 (cit. on p. 11).

[*GoCD* 2026] *GoCD*. https://www.gocd.org/. (Visited on 01/19/2026) (cit. on p. 24).

[Guyon 2003] Isabelle Guyon. "Design of experiments of the nips 2003 variable selection benchmark". In: *NIPS 2003 workshop on feature extraction and feature selection*. Vol. 253. 2003, p. 40 (cit. on p. 43).

[Han *et al.* 2022] Jiawei Han, Jian Pei, and Hanghang Tong. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2022. doi: 10.1016/C2013-0-18660-6 (cit. on p. 7).

[Harries 1999] M Harries. *Splice-2 comparative evaluation: electricity pricing*. Tech. rep. The University of New South Wales, Sydney, 1999 (cit. on p. 42).

[Helfstein and Braghetto 2024] Lucas Helfstein and Kelly Rosa Braghetto. "An empirical analysis of data drift detection techniques in machine learning systems". In: *Simpósio Brasileiro de Banco de Dados (SBBD)*. SBC. 2024, pp. 40–52 (cit. on p. 3).

[Helfstein and Braghetto n.d.] Lucas Helfstein and Kelly Rosa Braghetto. "Evaluating data drift detection and its effects on machine learning system performance". *Journal of Information and Data Management* (). To appear (cit. on p. 3).

[Hodges Jr 1958] JL Hodges Jr. "The significance probability of the smirnov two-sample test". *Arkiv för matematik* 3.5 (1958), pp. 469–486 (cit. on p. 11).

[Horkoff 2019] Jennifer Horkoff. "Non-functional requirements for machine learning: challenges and new directions". In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE. 2019, pp. 386–391 (cit. on p. 17).

[Kamei *et al.* 2021] Fernando Kamei *et al.* "Grey literature in software engineering: a critical review". *Information and Software Technology* 138 (2021), p. 106609 (cit. on pp. 21, 22).

[Karmaker Santu *et al.* 2021] Shubhra Kanti Karmaker Santu *et al.* "Automl to date and beyond: challenges and opportunities". *ACM Computing Surveys* 54.8 (2021), 175:1–175:36. DOI: 10.1145/3470918 (cit. on p. 19).

[*Kedro* 2026] *Kedro*. https://kedro.readthedocs.io/en/stable/index.html. (Visited on 01/19/2026) (cit. on pp. 28, 35).

[Kluyver *et al.* 2016] Thomas Kluyver *et al.* "Jupyter notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by Fernando Loizides and Birgit Scmidt. Netherlands: IOS Press, 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/ (cit. on pp. 28, 35).

[Kreuzberger *et al.* 2023] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. "Machine learning operations (mlops): overview, definition, and architecture". *IEEE access* 11 (2023), pp. 31866–31879 (cit. on pp. 18, 25, 68).

[*Kubeflow* 2026] *Kubeflow*. https://www.kubeflow.org/. (Visited on 01/19/2026) (cit. on pp. 27, 36).

[Kuwajima *et al.* 2020] Hiroshi Kuwajima, Hirotoshi Yasuoka, and Toshihiro Nakae. "Engineering problems in machine learning systems". *Machine Learning* 109.5 (2020), pp. 1103–1126 (cit. on p. 17).

[Leite *et al.* 2023] Leonardo Leite, Fabio Kon, and Paulo Meirelles. "A grounded theory of organizational structures for development and infrastructure professionals in software-producing organizations". In: *Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*. SBC. 2023, pp. 29–39 (cit. on p. 18).

[Liu *et al.* 2017] Anjin Liu, Yiliao Song, Guangquan Zhang, and Jie Lu. "Regional concept drift detection and density synchronized drift adaptation". In: *IJCAI International Joint Conference on Artificial Intelligence*. 2017 (cit. on p. 10).

[J. Lu *et al.* 2018] Jie Lu *et al.* "Learning under concept drift: a review". *IEEE transactions on knowledge and data engineering* 31.12 (2018), pp. 2346–2363 (cit. on pp. 9, 10, 29, 41).

[N. Lu *et al.* 2014] Ning Lu, Guangquan Zhang, and Jie Lu. "Concept drift detection via competence models". *Artificial Intelligence* 209 (2014), pp. 11–28 (cit. on p. 10).

[Lundberg and Lee 2017] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon *et al.* Curran Associates, Inc., 2017, pp. 4765–4774. url: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf (cit. on p. 26).

[Mersha *et al.* 2024] Melkamu Abay Mersha, Khang Lam, Joseph Wood, Ali AlShami, and Jugal Kalita. "Explainable artificial intelligence: a survey of needs, techniques, applications, and future direction". *Neurocomputing* 599 (2024). Preprint available as arXiv:2409.00265, p. 128111. doi: 10.1016/j.neucom.2024.128111 (cit. on p. 26).

[*Metaflow* 2026] *Metaflow*. https://github.com/Netflix/metaflow. (Visited on 01/19/2026) (cit. on p. 27).

[*ModelDB* 2026] *ModelDB*. https://github.com/VertaAI/modeldb. (Visited on 01/19/2026) (cit. on p. 28).

[Montiel *et al.* 2021] Jacob Montiel *et al.* "River: machine learning for streaming data in python" (2021) (cit. on p. 42).

[Muccini and Vaidhyanathan 2021] Henry Muccini and Karthik Vaidhyanathan. "Software architecture for ml-based systems: what exists and what lies ahead". In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE. 2021, pp. 121–128 (cit. on pp. 15, 25).

[Nascimento *et al.* 2020] Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø, and Tayana Conte. "Software engineering for artificial intelligence and machine learning software: a systematic literature review". *arXiv preprint arXiv:2011.03751* (2020). arXiv: 2011.03751 (cit. on pp. 2, 21).

[Nazir *et al.* 2024] Roger Nazir, Alessio Bucaioni, and Patrizio Pelliccione. "Architecting ml-enabled systems: challenges, best practices, and design decisions". *Journal of Systems and Software* 207 (2024), p. 111860 (cit. on p. 25).

[*Neptune* 2026] *Neptune*. https://neptune.ai/. (Visited on 01/19/2026) (cit. on p. 28).

[Ogasawara *et al.* 2025] Eduardo Ogasawara, Rebecca Salles, Fabio Porto, and Esther Pacitti. *Event detection in time series*. Springer, 2025 (cit. on p. 29).

[*OMG Unified Modeling Language (UML) Specification, Version 2.5.1* 2017] *OMG Unified Modeling Language (UML) Specification, Version 2.5.1*. Tech. rep. formal/2017-12-05. Object Management Group, 2017. url: https://www.omg.org/spec/UML/2.5.1/ (cit. on p. 32).

[Patel 2020] Jayesh Patel. "The democratization of machine learning features". In: *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE. 2020, pp. 136–141 (cit. on p. 29).

[Płońska and Płoński 2021] Aleksandra Płońska and Piotr Płoński. *MLJAR: State-of-the-art Automated Machine Learning Framework for Tabular Data. Version 0.10.3.* https://github.com/mljar/mljar-supervised. MLJAR Sp. z o.o., Łapy, Poland. 2021 (cit. on p. 19).

[Publio *et al.* 2018] Gustavo Correa Publio *et al.* "Ml-schema: exposing the semantics of machine learning with schemas and ontologies". *arXiv preprint arXiv:1807.05351* (2018) (cit. on p. 34).

[*PyCaret* 2026] *PyCaret*. https://github.com/pycaret/pycaret. (Visited on 01/19/2026) (cit. on p. 28).

[Rabanser *et al.* 2019] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. "Failing loudly: an empirical study of methods for detecting dataset shift". *Advances in Neural Information Processing Systems* 32 (2019) (cit. on pp. 11, 30).

[Ramírez-Gallego *et al.* 2017] Sergio Ramírez-Gallego, Bartosz Krawczyk, Salvador García, Michał Woźniak, and Francisco Herrera. "A survey on data preprocessing for data stream mining: current status and future directions". *Neurocomputing* 239 (2017), pp. 39–57 (cit. on p. 10).

[M. T. Ribeiro *et al.* 2016] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why should I trust you?" Explaining the predictions of any classifier". *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 13-17-Augu (2016), pp. 1135–1144. doi: 10.1145/2939672.2939778. arXiv: 1602.04938 (cit. on p. 26).

[Mauro Ribeiro *et al.* 2016] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. "Mlaas: machine learning as a service". *Proceedings - 2015 IEEE 14th International Conference on Machine Learning and Applications, ICMLA 2015* (2016), pp. 896–902. doi: 10.1109/ICMLA.2015.152 (cit. on p. 19).

[*Sagemaker* 2026] *Sagemaker*. https://aws.amazon.com/pt/sagemaker/. (Visited on 01/19/2026) (cit. on pp. 19, 27, 36).

[Sato *et al.* 2019] Danilo Sato, Arif Wider, and Christoph Windheuser. "Continuous delivery for machine learning". *Martin Fowler* 9 (2019) (cit. on pp. 24, 32).

[Schlimmer and Granger 1986] Jeffrey C Schlimmer and Richard H Granger. "Incremental learning from noisy data". *Machine learning* 1 (1986), pp. 317–354 (cit. on p. 42).

[Schröder and Schulz 2022] Tim Schröder and Michael Schulz. "Monitoring machine learning models: a categorization of challenges and methods". *Data Science and Management* 5.3 (2022), pp. 105–116 (cit. on pp. 1, 15, 25).

[Sculley *et al.* 2015] D. Sculley *et al.* "Hidden technical debt in machine learning systems". In: *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*. 2015, pp. 2503–2511. URL: https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems (cit. on pp. 1, 23, 29).

[*Seldon Core* 2026] *Seldon Core*. https://github.com/SeldonIO/seldon-core. (Visited on 01/19/2026) (cit. on pp. 27, 36).

[Serban, Blom, *et al.* 2020] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. "Adoption and effects of software engineering best practices in machine learning". In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–12 (cit. on p. 24).

[Serban, Blom, *et al.* 2021] Alex Serban, Koen van der Blom, Holger Hoos, and Joost Visser. "Practices for engineering trustworthy machine learning applications". In: *2021 IEEE/ACM 1st Workshop on AI engineering-software engineering for AI (WAIN)*. IEEE. 2021, pp. 97–100 (cit. on p. 25).

[Serban and Visser 2022] Alex Serban and Joost Visser. "Adapting software architectures to machine learning challenges". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 152–163 (cit. on pp. 2, 21, 25).

[Shalev-Shwartz and Ben-David 2014] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781107298019 (cit. on pp. 5, 6).

[*SHapley Additive exPlanations* 2026] *SHapley Additive exPlanations*. https://github.com/slundberg/shap. (Visited on 01/19/2026) (cit. on p. 26).

[Silva *et al.* 2013] Jonathan A Silva *et al.* "Data stream clustering: a survey". *ACM Computing Surveys (CSUR)* 46.1 (2013), pp. 1–31 (cit. on p. 10).

[Sokolova and Lapalme 2009] Marina Sokolova and Guy Lapalme. "A systematic analysis of performance measures for classification tasks". *Information Processing and Management* 45.4 (2009), pp. 427–437. ISSN: 03064573. DOI: 10.1016/j.ipm.2009.03.002. URL: http://dx.doi.org/10.1016/j.ipm.2009.03.002 (cit. on p. 7).

[Souza *et al.* 2020] V. M. A. Souza, D. M. Reis, A. G. Maletzke, and G. E. A. P. A. Batista. "Challenges in benchmarking stream learning algorithms with real-world data". *Data Mining and Knowledge Discovery* 34 (2020), pp. 1805–1858. DOI: 10.1007/s10618-020-00698-5 (cit. on pp. 41, 51).

[Street and Kim 2001] W Nick Street and YongSeog Kim. "A streaming ensemble algorithm (sea) for large-scale classification". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, pp. 377–382 (cit. on p. 42).

[*Verta AI* 2026] *Verta AI*. https://www.verta.ai/. (Visited on 01/19/2026) (cit. on p. 28).

[*Vertex AI* 2026] *Vertex AI*. https://cloud.google.com/vertex-ai. (Visited on 01/19/2026) (cit. on pp. 19, 27, 36).

[Washizaki *et al.* 2019] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. "Studying software engineering patterns for designing machine learning systems". In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2019, pp. 49–54. doi: 10.1109/IWESEP49350.2019.00017 (cit. on pp. 2, 21).

[Webb *et al.* 2016] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. "Characterizing concept drift". *Data Mining and Knowledge Discovery* 30.4 (2016), pp. 964–994 (cit. on p. 8).

[Weller 2019] Adrian Weller. "Transparency: motivations and challenges". In: *Explainable AI: interpreting, explaining and visualizing deep learning*. Springer, 2019, pp. 23–40 (cit. on p. 16).

[Wilkinson *et al.* 2016] Mark D. Wilkinson *et al.* "Comment: the fair guiding principles for scientific data management and stewardship". *Scientific Data* 3 (Mar. 2016). issn: 20524463. doi: 10.1038/sdata.2016.18 (cit. on p. 34).

[Xiong *et al.* 2022] Pulei Xiong *et al.* "Towards a robust and trustworthy machine learning system development: an engineering perspective". *Journal of Information Security and Applications* 65 (2022), p. 103121. doi: 10.1016/j.jisa.2022.103121 (cit. on p. 15).

[Zhang *et al.* 2022] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. "Machine learning testing: survey, landscapes and horizons". *IEEE Transactions on Software Engineering* 48.1 (2022), pp. 1–36. doi: 10.1109/TSE.2019.2962027 (cit. on p. 23).