TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

**ACTA TECHNICA NAPOCENSIS**

**Series: Applied Mathematics, Mechanics, and Engineering**
**Vol. 66, Issue III, August, 2023**

# STREAMLINING MACHINE LEARNING WORKFLOWS IN INDUSTRIAL APPLICATIONS WITH CLI'S AND CI/CD PIPELINES

**Adrian-Ioan ARGESANU, Gheorghe-Daniel ANDREESCU**

***Abstract:*** *The integration of machine learning (ML) in various organizations has become an essential aspect with a wide range of applications. However, the development and deployment of machine learning models can be time-consuming and prone to errors due to the iterative nature of the process and the constant testing and retraining of models. As ML becomes more integrated with industrial systems, the demand for controlled, reproducible and repeatable processes rises. This paper proposes a novel approach for the automation of various workflows of the ML model lifecycle via custom Command Line Interfaces (CLI) and Continuous Integration/Continuous Deployment (CI/CD) pipelines. We discuss the challenges and pitfalls of non-automated ML workflows, as well as the benefits of using the proposed toolset. We introduce our bespoke approach to CLI and CI/CD automation, highlighting timesaving as well as consistency-improvement aspects for Process & Pipeline Services' use-case of detecting mechanically induced stress cracking in pipelines. This paper adds to the limited literature on ML lifecycle automation.*

***Key words:*** *machine learning, workflow automation, CLI, CI/CD, glue code, automated deployment.*

## 1. INTRODUCTION

Machine learning has become a critical tool in many organizations. Development and productionization of ML applications requires coordinated efforts from machine learning experts, data scientists, data engineers, software engineers, and DevOps personnel. A variety of tools, open source or licensed, exists and are continuously developed to support the cross-disciplinary team in the various stages of the ML lifecycle. Orchestration of these tools however is an aspect which is typically neglected. Systems coordinated via glue code and scripts suffer from high technical debt [1] and therefore system fragility.

ML systems furthermore are rarely production-ready after the first model training. With every iteration of the development cycle, a variety of steps are repeated, and artefacts generated and moved around. When these operations are executed manually, in a non-predefined order, or lack the necessary guardrails to suppress subjective interpretation, the risk of human error is greatly increased, and consistency and reproducibility potentially compromised.

The growing complexity of ML systems can also not be neglected. Abiding to the software development paradigm, each module, component, or service of the ML application must be first tested in isolation, before being integrated at various levels to iteratively validate the complete solution. Failing to do so might result in unexpected results and system instability.

Despite the vast array of challenges and importance of efficient solutions for operational excellence, there is limited literature discussing ML workflow automation. [9] discusses Software Engineering challenges in ML solutions but offers but a theoretical checklist not yet applied in the industry. Studies such as [6] and [7] have identified a knowledge gap among machine learning developers in building automation pipelines. [8] notes that "unifying and automating the day-to-day workflow of […] engineers reduces overhead and facilitates progress in the field". [10] investigates the use of DevOps practices in ML applications in an

**Received**: 28.06.23; **Similarities**: 04.07.23: **Reviewed**: 14.07./24.07.23: **Accepted**:20.08.23.

effort to improve operational functions for real-world applications – a concept this paper expands and improves on.

Expanding on the introductory concepts, section 2 lays out automation challenges as well as potential solutions. Section 3 presents our proposed approach, developed as part of Process & Pipeline Services' use-case of detecting mechanically induced stress cracking in pipelines, addressing presented risks and showcasing benefits. We conclude in section 4, outlining directions for future work.

## 2. AREAS OF AUTOMATION

In this section, we focus on three often overlooked pitfalls of ML systems. We discuss their risks and provide solutions.

### 2.1 Glue Code

The main purpose of glue code is to make the components work together seamlessly, and provide a single, cohesive system. It is often found in the following components:

- Data ingress and egress, to handle reading/writing data from/to various stores. Examples include files and databases, and often involve format conversions.
- Preprocessing, to perform data cleaning, normalization, and other transformations, preparing the data for ML operations. We consider preprocessing the prototype of the data pipeline of a ML model, consisting of ad-hoc operations executed as part of exploratory model development.
- Model training and evaluation, as a harness to run the ML model training, as well as evaluate its performance.
- Deployment, to integrate the trained model into a larger system and make it accessible to consumers.

The use of glue code in ML systems however poses some risks and challenges:

- Maintainability. Glue code can become complex and hard to maintain over time, especially as the system evolves and grows. Poorly written glue code can result in bugs and errors that can impact the performance and stability of the system.
- Lack of scalability. The code may not be designed to handle large-scale data processing and may become a bottleneck. This can result in degraded performance and longer processing times, leading to decreased efficiency and user frustration.
- Integration issues. Glue code is responsible for integrating different components of the system, and if not designed correctly, it can yield interfacing issues and data loss.
- Lack of modularity. Glue code can sometimes become monolithic, making it difficult to change or update components without affecting the rest of the system.
- Lack of reusability. Glue code is typically highly customized for a specific task or scenario and its inputs and outputs. The lack of modularity and impaired maintainability inhibits the reuse of glue code, with development teams often rewriting it even for similar problems.

System incohesion as well as the undefined impact on inference performance through buggy behavior or data loss have an exceptional bearing on safety-critical or industrial applications. To mitigate these, as well as all other risks listed above, it is important to invest in good software engineering practices when developing glue code. This includes using well-established design patterns, rigorous testing processes, and keeping code modular and scalable.

One way to achieve this is to abstract common functionality into a command line interface (CLI). A well-designed CLI can provide a consistent and automated method for performing frequent operations:

- Consolidation of data connectors can facilitate re-use throughout the system. In addition, store schema can be abstracted and configured via e.g., JSON to increase flexibility.
- Applying good software engineering practices when developing preprocessing routines facilitates their immediate reuse in production pipelines.

- Abstract training, evaluation and deployment scaffolds, as well as use-case independent utilities can be recycled for subsequent ML applications.

In addition, a custom CLI can also provide a user-friendly ingress route for personnel attempting to perform common tasks they are not specialized in – e.g., a data scientist wanting to deploy a ML model in a sandbox environment would not need to be aware of the DevOps specifics.

## 2.2 Workflows

Production-ready ML products are rarely the result of the first training iteration. The machine learning development cycle involves repeating various steps, producing artifacts, and moving them around with each change made to the input set, data pipeline, model architecture and parameters, or deployment strategy. When done at scale, this increases the risk of errors unless standardized and automated.

One way of addressing this aspect is to implement often repeated operations in the CLI, aggregate these to workflows, and add CLI commands as hooks for the newly created workflows. The benefits of this approach include:

- Improved reproducibility, ensuring that the same steps are followed every time a model is developed or deployed.
- Reduced risk of human error, minimizing the need for manual intervention. This is particularly important in complex ML projects, where this risk can be significant.
- Consistent execution, guaranteeing that all steps are executed consistently and in the correct order, reducing the risk of missed steps or incorrect configurations.
- Faster deployment times. Automated workflows can greatly reduce the time required to deploy machine learning models, as they eliminate the need for user interaction and ensure that all steps are executed quickly and efficiently.
- Improved collaboration through integration with version control systems and other collaboration tools, making it easier for teams to work together on a project. This helps to reduce the risk of conflicting changes and improves communication between team members.
- Increased efficiency, freeing engineers from performing repetitive tasks, and thus allowing them to focus on more important aspects of the development cycle.
- Auditability. Workflows can be configured to generate and automatically upload audit artefacts for applications where extensive recordkeeping is required.

## 2.3 Testing and Integration

Similar to traditional software development, from which ML system engineering inherits its coding-specific challenges, testing and integration represent critical aspects for ML development.

We define testing as the process of evaluating the code against a set of test cases to verify its behavior and performance. The main purpose is to identify bugs, issues, and errors in the code as early as possible. Similar to traditional code, ML test coverage should include unit- as well as integration tests, ensuring low-level operations and the modules that aggregate them perform to specification.

In the context of this paper, integration is defined as the process of aggregating the machine learning code with other components and (sub-)systems to create a complete solution. Functional tests should be performed at this level to assert the success of the integration and identify any issues or conflicts.

Testing and integration efforts can scale proportionally to the overall complexity of the solution. Regular execution of these operations might imply disruption of up to several hours for the user executing these. In addition, functional tests might require components or services of the broader system that are difficult to deploy or emulate locally.

Failing to test and integrate code, modules and services might result in uncontrolled program behavior, undefined results and service incompatibility. Relying on manual integration workflows adds several of the risks presented in section 2.2. When executed too rarely, testing and integration efforts might aggregate significant amounts of updates, making root cause analysis of failures increasingly difficult.

Inspired by the software development paradigm, Continuous Integration / Continuous Deployment (CI/CD) systems can be set up to continuously run ML-specific testing and integration tasks. In addition to running unit and integration tests, CI/CD pipelines can be used to:

- Simulate the ML training process and assert the stability of the training workflow.
- Act as a controlled-environment sandbox for functional testing, validating the integration of readily trained models with other components of the broader system.
- Build and deploy ML models.

## 3. CASE STUDY

For a practical implementation of the presented concepts and considerations, we focus on our Platform for End-to-End Lifecycle Management of Batch-Prediction Machine Learning Models, as introduced in [2] and employed for Process & Pipeline Services' use-case of detecting mechanically induced stress cracking in data recorded during pipeline in-line inspections.

The solutions and components presented in this section have been production-hardened for several years. Although bespoke to our platform, these can be translated to any ML system and domain. The safety-critical nature of the use-case these have been developed for increases overall robustness and resiliency.

In our platform, we employ containerization to wrap individual services in HTTP servers. Employing the "separation of concerns" software design principle, services are never collocated in the same Docker container – ML models are separated from data ingress connectors, platform orchestration modules, and other auxiliary components. A complete system is therefore composed of at least three containers: ML model, data access, and orchestration. Each service is developed, tested, and deployed in isolation.

Similar to source code version control, we implement a versioning system for all services, in which each service can be uniquely identified by the tuple `(<service_name>; <build_uuid>)`, where `service_name` represents the name of the service and `build_uuid` is a universally unique identifier (UUID) generated at build time.

To automate the majority of the glue code operations, a novel and dedicated CLI with customizable contexts has been implemented. CLI commands are available anywhere inside the checked-out copy of the source code. On the file system, the source code is organized in folders; each folder stores the code and configuration of one service. The behavior of the CLI commands is custom to the context – or working directory – of their execution. When processing any command, the name of the service is implied from the context.

Use-cases of the CLI include integration and testing of code modules, building and testing of Docker images, training and validation of ML models, and deployment and rollback of services. To list a few examples:

- Interaction with the service versioning system, including logging in and out, running queries and checks, and pulling and pushing of artefacts:
  `cli pull {build_uuid}`: pulls the Docker image and auxiliary artefacts of the provided `build_uuid` to the local machine. The service is implied from the context.
  `cli push {build_uuid}`: the counterpart of `pull`; pushes the artefacts associated with the provided `build_uuid` to the versioning system.
  `cli ls`: lists all available build UUIDs for a service.
- Deployment of services in the local Docker environment, including debugging capabilities:
  `cli up`: deploys/starts the service.
  `cli down`: stops the service.
- Deployment of services on remote Docker systems:
  `cli deploy [{scm_ref} {build_uuid}]`: either builds the service from the provided source code management (SCM) revision `scm_ref`, or pulls the readily available build with `build_uuid`

from the versioning system and deploys it on the remote system.

- Automation around model training, including building of model images, running, monitoring and validation of the training, and capturing of the training artefacts:
  `cli build {data} {configuration}`: builds a Docker image for the ML model and trains the model on the specified `data` as per `configuration`. Once trained, the model is saved to disk.
  `cli predict {data}`: deploys a trained model locally and predicts on the provided `data`.
- Automation around unit and integration testing of various components, including environment setup and teardown:
  `cli test`: runs the tests – unit, integration, functional – available for the current context.

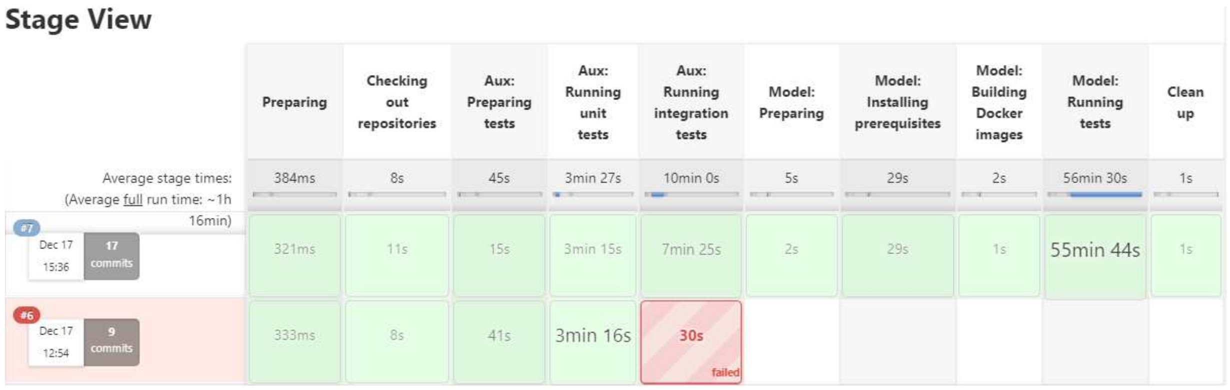
The CLI does not only cover individual operations, but also aggregates these to automated workflows. One of the most prominent examples is the `cli deploy` command, which, when called with the `scm_ref` argument, initiates a five-step workflow: 1) run SCM checks; 2) build the service – ML model or auxiliary; 3) push the artefacts to the versioning system; 4) deploy the product on the client system; and 5) verify deployment success.

The CLI also acts as the onboarding platform for novel functionality intended to be shared across ML models or services, such as data accessors and preprocessing operations. As code and functionality is especially volatile in early ML development phases, the CLI can be used to rapidly prototype new functions, with the stricter development standards imposed by the CLI enabling subsequent promotion of these to the individual services and modules as they mature. Once available in the CLI, functionality can be immediately reused by other users.

Our CLI facilitates but does not impose regular integration and testing efforts. To ensure the stability of the codebase across commits, a custom CI/CD setup is employed. We configure Jenkins pipelines, to build, test and deploy code, images and services. The Jenkins pipelines leverage CLI functionality wherever possible – e.g., via the `build`, `test`, `up` hooks, and in turn the CLI leverages the Jenkins deploy pipeline for remote rollout of services.

To build and test all code check-ins – including production simulation, a multi-stage build pipeline has been set up. The following stages are executed for every build, and are depicted in Figure 1:

- Environment reset. The Jenkins instance builds several images and runs several containers from these images as part of every build. The “Preparing” step ensures previous artefacts do not interfere with the active build.
- Checkout of the latest code from the SCM system, as part of the “Checking out repositories” step.
- Build of all non-model / auxiliary components using the CLI’s `build` hook, followed by unit and integration testing via `cli test`. The unit tests use mock-services



Stage View

| | Preparing | Checking out repositories | Aux: Preparing tests | Aux: Running unit tests | Aux: Running integration tests | Model: Preparing | Model: Installing prerequisites | Model: Building Docker images | Model: Running tests | Clean up |
|---|---|---|---|---|---|---|---|---|---|---|
| Average stage times: (Average full run time: ~1h 16min) | 384ms | 8s | 45s | 3min 27s | 10min 0s | 5s | 29s | 2s | 56min 30s | 1s |
| #7 Dec 17 15:36 17 commits | 321ms | 11s | 15s | 3min 15s | 7min 25s | 2s | 29s | 1s | 55min 44s | 1s |
| #6 Dec 17 12:54 9 commits | 333ms | 8s | 41s | 3min 16s | 30s failed | | | | | |

**Fig. 1.** Jenkins build pipeline.

and -stores; the integration tests simulate small-scale deployment.
Pipeline steps: "Aux: Preparing tests", "Aux: Running unit tests" and "Aux: Running integration tests".

- Build – `cli build` – and test – `cli up` followed by `cli test` – of the ML model components. Besides unit and integration tests, all available functional tests are executed to assert the interfacing with the rest of the system, including complete-solution simulation. For functional testing, we treat the containers as black boxes. At the very least, every developed ML model has functional tests to check the train and predict functionality.
Relevant pipeline steps are: "Model: Preparing", "Model: Installing Prerequisites", "Model: Building Docker images" and "Model: Running tests".
- Aggregation of test reports and cleanup, as part of the "Clean-up" step.

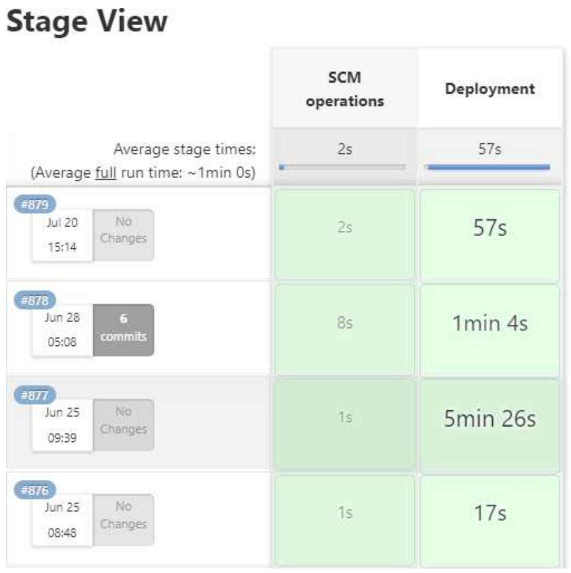
The build pipeline is configured to fail fast. If any of the stages reports errors, the build is interrupted. Figure 1 presents two builds, #6 which failed the integration tests for the auxiliary services and #7 which successfully completed all stages after further 17 commits. The average build time for the whole pipeline is roughly 76 minutes, with the ML model tests amounting to 73% of the total.

For deployment of any service, a second Jenkins pipeline is employed. Depicted in Figure 2, the deploy pipeline consists of two stages:

- Source code checkout – "SCM operations".
- The actual deployment – "Deployment".

The deploy pipeline leverages the CLI's `push` functionality to persist all services and their artefacts in the versioning system. This not only allows us to trace any prediction to the exact version of the model that generated it and its source code, but also to identify all supporting services of that particular setup, including their versions.

This Jenkins pipeline takes 3 parameters, as illustrated in Figure 3. The first two, `SCM_REF` and `SERVICE_NAME` are relevant for deployment of new builds and specify the source



Stage View

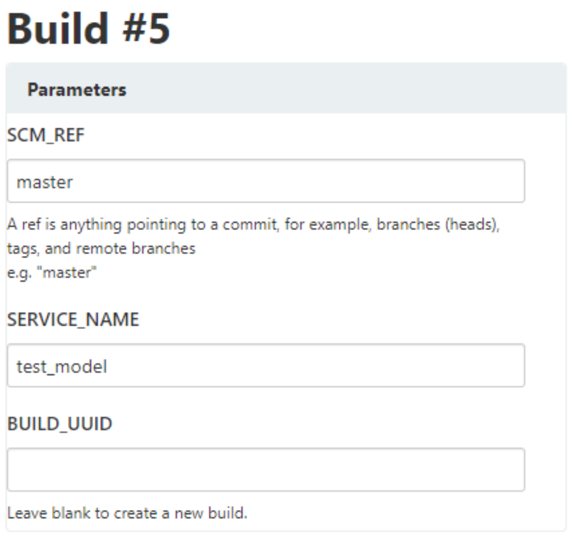
| | SCM operations | Deployment |
|---|---|---|
| Average stage times: (Average full run time: ~1min 0s) | 2s | 57s |
| #879 Jul 20 15:14 No Changes | 2s | 57s |
| #878 Jun 28 05:08 6 commits | 8s | 1min 4s |
| #877 Jun 25 09:39 No Changes | 1s | 5min 26s |
| #876 Jun 25 08:48 No Changes | 1s | 17s |

**Fig. 2.** Jenkins deploy pipeline.

code revision to use and the service to be deployed.

The deploy pipeline can also be used to roll services and models back to previous versions. To roll items back, the third parameter of the deploy pipeline, `BUILD_UUID`, must be configured. Rollback operations do not build any new images or services since these are readily available in the versioning system.

Although it can be manually invoked from the Jenkins UI, the deploy pipeline is meant to run as part of the automated workflow of the CLI's `deploy` command. When triggered through the CLI hook, the pipeline is augmented with pre- and post-deploy checks, and the



Build #5

Parameters

SCM_REF
master
A ref is anything pointing to a commit, for example, branches (heads), tags, and remote branches
e.g. "master"

SERVICE_NAME
test_model

BUILD_UUID

Leave blank to create a new build.

**Fig. 3.** Jenkins deploy pipeline parameters.

`SERVICE_NAME` is implied from the context the CLI was run in.

Figure 2 shows four successful deploy jobs, with execution times ranging from under 20 seconds to several minutes. The time required for the "Deployment" step is proportional to the complexity of the service being rolled out.

Our automated build and deploy workflows abstract away DevOps-specifics, allowing software developers, data scientists and ML experts to focus their efforts where they matter most, thus increasing efficiency. Furthermore, the consistent execution of all involved steps minimizes the risk of human error and improves reproducibility. Being configured to fail fast ensures relevant personnel is immediately notified on build or deploy errors, reducing the risk of production outage.

Our solution improves on the recommendations of [10] in several ways. To name a few:

- Our bespoke CLI abstracts common functionality for maximum reuse in local development, as well as in the CI/CD context.
- Our CI/CD approach entails building and testing of code directly in Docker images, ensuring compatibility with the target runtime.
- The proposed CI/CD flow includes staging environment simulation as part of the functional tests to ensure system-wide cohesion without the need of a separate environment.
- The CLI and CI/CD automations produce and store audit artefacts.

For Process & Pipeline Services' use-case, the CLI's model `build` automation allowed training iterations to run unattended for several weeks, enabling the project team to focus on different tasks. With validation metrics and training artefacts captured at the end of each cycle, audit logs were readily available, and performance walks across model iterations and production rollout were greatly simplified.

The CLI and CI/CD solutions were employed from prototyping to production rollout, as depicted in Figures 1 and 2. The resulting ML system was production-deployed on a cluster of AWS g3.4xlarge EC2s, serving batch predictions on several million data points per dataset, for datasets exceeding several TB in uncompressed storage, utilizing up to 64 Data Access connectors in parallel, without stability or memory exhaustion issues.

## 4. CONCLUSIONS AND FUTURE WORK

ML model development and deployment remains time-consuming and prone to errors if orchestration aspects are neglected. Poorly developed glue code, manual executed workflows, and the lack of testing and integration can lead to system fragility and instability, consistency and reproducibility issues, as well as undefined inference results.

Striving to help close the gap in ML workflow automation literature, we have not just discussed theoretical considerations, but also presented a practical implementation of our novel approach consisting of bespoke CLI commands and CI/CD pipelines, as applied to Process & Pipeline Services' use-case of detecting mechanically induced stress cracking in pipelines.

The presented approach has the potential to revolutionize the way organizations develop, train, and deploy machine learning models. The automation of workflows can greatly improve the efficiency and reliability, CLI standardization can increase overall code quality, improve code reuse and system accessibility for non-expert personnel, automated integration and testing efforts can mitigate code validation risks as well as ensure system cohesion in production settings.

Whilst the integration of CLI and CI/CD has been shown to be a successful approach in streamlining ML workflows, there is still room for improvement. One avenue for further exploration is to examine the integration of these tools with other DevOps technologies, such as cloud deployment.

## 5. REFERENCES

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, D. Dennison, *Hidden Technical Debt*

*in Machine Learning Systems*, 2015, NIPS. 2494-2502.

[2] A. -I. Argesanu, G. -D. Andreescu, *A Platform to Manage the End-to-End Lifecycle of Batch-Prediction Machine Learning Models*, 2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics (SACI), 2021, pp. 329-33.

[3] S. Amershi et al., *Software Engineering for Machine Learning: A Case Study*, 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291-300, doi: 10.1109/ICSE-SEIP.2019.00042.

[4] D. Xin, E.Y. Wu, D.J. Lee, N. Salehi, A. Parameswaran, *Whither automl? understanding the role of automation in machine learning workflows*, InProceedings of the 2021 CHI Conference on Human Factors in Computing Systems 2021 May 6 (pp. 1-16).

[5] I. Karamitsos, S. Albarhami, C. Apostolopoulos, *Applying DevOps Practices of Continuous Automation for Machine Learning*, Information 2020, 11, 363, https://doi.org/10.3390/info11070363

[6] S. Garg, P. Pundir, G. Rathee, P.K. Gupta, S. Garg, S. Ahlawat, *On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps*, 2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), 25-28, 2021.

[7] L.E. Lwakatare, A. Raj, I. Crnkovic, J. Bosch, H.H. & Olsson, *Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions*, Inf. Softw. Technol., 127, 106368, 2020.

[8] S. Amershi, A. Begel, C. Bird, R. DeLine, H.C. Gall, E. Kamar, N. Nagappan, B. Nushi, T. Zimmermann, *Software Engineering for Machine Learning: A Case Study*, 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 291-300, 2019.

[9] E.D. Nascimento, I., Ahmed, E. Oliveira, M.P. Palheta, I. Steinmacher, T.U. Conte, *Understanding Development Process of Machine Learning Systems: Challenges and Solutions*, 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1-6, 2019.

[10] I. Karamitsos, S. Albarhami, C. Apostolopoulos, *Applying DevOps Practices of Continuous Automation for Machine Learning*, Inf., 11, 363, 2020.

## AUTOMATIZAREA WORKFLOW-URILOR ML IN APLICATII INDUSTRIALE PRIN CLI SI CI/CD

**Rezumat:** Integrarea învățării automate (ML) în diverse organizații a devenit un aspect esențial cu o gamă largă de aplicații. Cu toate acestea, dezvoltarea și implementarea modelelor ML poate necesita o perioade de timp îndelungata, și poate fi predispusă la erori, datorită naturii iterative a procesului. Pe măsură ce ML devine din ce în ce mai integrat cu sistemele industriale, cererea de procese controlate, reproductibile și repetabile crește. Această lucrare propune o abordare nouă pentru automatizarea diferitelor workflow-uri ale ciclului de viață al modelelor ML prin intermediul interfețelor de linie de comandă (CLI) personalizate și pipeline-urile de Continuous Integration/Continuous Deployment (CI/CD). Dezbatem provocările și riscurile workflow-urilor ML neautomatizate, precum și beneficiile utilizării soluțiilor propuse, explorând use-case-ul al Process & Pipeline Services de detectare a fenomenelor de fisurare al pipeline-urilor datorate tensiunilor mecanice acumulate. In lucrare propunem elemente avansate de stricta noutate nemaiîntâlnite in literatura de specialitate.

**Adrian-Ioan ARGESANU,** PhD. Stud. Eng., Politehnica University Timisoara, Faculty of Automation and Computers, adrian.argesanu@student.upt.ro, Bulevardul Vasile Pârvan 2, 300223 Timișoara, Romania, Baker Hughes, Process & Pipeline Services, adrian.argesanu@bakerhughes.com, +4972447320, Lorenzstraße 10, 76297 Stutensee, Germany.

**Gheorghe-Daniel ANDREESCU,** PhD. Eng. Prof., Politehnica University Timisoara, Faculty of Automation and Computers, daniel.andreescu@upt.ro, +40256403507, Bulevardul Vasile Pârvan 2, 300223 Timișoara, Romania.