Requirements and Reference Architecture for MLOps: Insights from Industry

Indika Kumara
JADS and Tilburg University, Netherlands

Rowan Arts
JADS and Tilburg University, Netherlands

Dario Di Nucci
University of Salerno, Italy

Rick Kazman
University of Hawaii, United States

Willem Jan Van Den Heuvel
JADS and Tilburg University, Netherlands

Damian Andrew Tamburri
JADS and Eindhoven University of Technology, Netherlands

***Abstract*—Machine Learning Operations (MLOps) streamline the lifecycle of machine-learning models in production. In recent years, the topic has picked the interest of practitioners, and consequently, a considerable number of tools and gray literature on architecting MLOps environments has emerged. However, this has created a new problem for organizations: selecting the most appropriate tools and design options for implementing their MLOps environments. To alleviate this problem, this paper proposes a reference architecture and requirements for MLOps by systematically reviewing 58 industrial gray literature articles. Such reference architecture drawn from the state of practice shall aid organizations in making better design and technology choices when embarking on their MLOps journey while providing a technology-independent baseline for further MLOps research.**

Introduction

Organizations are increasingly adopting artificial intelligence (AI) and machine-learning (ML) in their businesses to turn their raw business data into value [1]. However, achieving productive ML solutions is tough, mainly due to the challenges of maturing an ML model—developed by data scientists who are typically not skilled software engineers—into production and keeping it operating at scale [2]. For example, according to Algorithmia [3], only 22% of organizations that use ML have successfully deployed a model into production.

ML operations (MLOps) has emerged as a discipline to help bridge the gap between the development of ML models and their (continuous, resilient, etc.) operations [4], [5]. Like DevOps, MLOps aims to offer a set of practices and tools to automate and combine the processes between model development and model operations, accel-

erating the delivery of models.

MLOps has gained the attention of technology providers, and the MLOps landscape is exploding with tools and platforms [5], [6]. With many diverse technology options available, a problem for organizations is to select specific options and build or assemble an MLOps environment that can best serve their needs. To help address this problem, we synthesized a reference architecture for an MLOps environment and distilled the typical minimum requirements for each component in the architecture. An organization can use this reference architecture as a template for creating their MLOps system designs. MLOps should be agnostic from language, tool/platform, and infrastructure [5]; a reference architecture and a common set of requirements could facilitate this objective.

Despite the popularity of MLOps, little academic literature exists on the topic, particularly on designing MLOps solutions [6], [7]. However, we have witnessed a significant amount of gray literature on the topic because practitioners (*e.g.*, tool developers, data scientists, and consultants) are increasingly publishing their advice and experiences. Hence, we saw an opportunity to conduct a systematic review of the gray literature on MLOps to identify the requirements for an MLOps environment and derive a reference architecture that addresses those requirements.

Methodology

We adopted the standard guidelines used by the research community for systematically reviewing white, gray, and multi-vocal literature [8], [9].

We first defined three research questions to obtain the data necessary for creating a reference architecture for an MLOps environment.

- **RQ1: What requirements should an MLOps environment meet?**
- **RQ2: What components should an MLOps environment include?**
- **RQ3: What architectures have practitioners proposed for MLOps environments?**

Guided by the above research questions, we next formulated the query: *mlops AND architecture(s) OR requirement(s) OR feature(s) OR component(s)*. We ran the query on the Google search engine, scanning each resulting page until saturation. We only considered textual sources such as articles, blogs, white papers, and slides. We identified 257 sources and applied inclusion/exclusion criteria and quality assessment criteria to create a final list of 58 literature sources. For the eligibility criteria, we used the standard measures (commonly used by the gray literature studies [9]) such as the focus of the study (as defined by the research questions), written in English, publisher's reputation, and author's expertise. The first two authors of the paper independently performed this study selection process. We measured Cohen's kappa coefficient to assess the inter-rater reliability, which was 0.74, indicating substantial agreement.

To extract the data from the selected sources, we applied the qualitative data generation techniques from the grounded theory approach [10]. In particular, we used structural and descriptive coding to establish codes, groups, and categories. The first author of this paper coded all sources, and the second and third authors reviewed the generated codes. We resolved all the discrepancies via discussions. In addition to the text, we analyzed the architectural designs in the sources.

The replication package for our study is available online [1]. It contains the flow chart of the review methodology, the complete list of sources, the qualitative analysis performed with the *Atlas.ti* tool, and the extracts from the sources.

Reference Architecture and Requirements

MLOps processes take place on a compute stack with the necessary development and operations capabilities. We formulated the requirements (see Table 1) and a reference architecture (see Figure 1) for such a stack from the results of our gray literature analysis. The reference architecture consists of three horizontal layers and two vertical cross-cutting modules. The horizontal layers present the key capabilities and responsibilities of infrastructures, platforms, and applications (*i.e.*, MLOps pipelines). The vertical modules, *i.e.*, automation and governance, capture the key functionalities relevant to each horizontal layer. This section discusses each layer and

[1] https://github.com/IndikaKuma/MLOpsGLR.git

requirement in detail.

Infrastructure Layer

This layer provides computing resources to host and execute platform services, pipelines, governance applications, and CI/CD automation services. The infrastructure needs to be flexible and portable to prevent vendor lock-in and enable the rapid (re)deployment of pipelines (**R1**). A reproducible and versionable infrastructure supports auditing and debugging infrastructure changes and allows switching between different versions of platform services, pipelines, and ML models. The other desired features of the infrastructure layer include multi-cloud support, auto-scalability, and hardware accelerators (**R2**). The multi-cloud support allows different teams in an organization to use the best possible cloud and tools for building and deploying their models. Auto-scaling enables scaling up and down training pipelines and models (serving components) automatically to cope with fluctuating data and serving requests. Hardware accelerators may be necessary to speed up pipeline execution during experimentation and (re)training and to improve real-time serving latency.

An MLOps environment can include separate development, staging, and production environments (**R3**). These environments' configurations (*e.g.*, tools and hardware resources) may exhibit variations. Infrastructure as code (IaC) can be used to automate the provisioning and management of such environments while preserving their reproducibility and auditability (**R4**).

Platform Layer

This layer facilitates applying platform thinking [11] to build a *self-serve* MLOps platform to empower different actors involved in creating, deploying, and maintaining ML models. It supports managing the entire lifecycle of models and related artifacts such as data and ML code. In this section, we present the platform capabilities that we synthesized from the gray literature.

Pipeline Development and Execution The pipelines implement ML processes such as data pre-processing, feature engineering, (re)training, and prediction. A pipeline consists of steps that must be executed in a specific order. The platform should empower developers to publish, share and reuse pipelines to enable fast development and automated (re)deployment of pipelines (**R5**). Moreover, the pipeline steps need to be implemented as modular containerized components that can be easily reused and composed (**R6**). The pipeline execution can adopt a choreography model or an orchestration model (**R7**). The former needs an event bus to facilitate the event/message-driven coordination of pipeline steps. The latter defines the pipeline as a workflow model that a workflow engine can execute centrally.

Experimentation, Training, and Testing The experimentation of a pipeline requires its deployment, execution, and debugging, followed by the analysis and interpretation of the produced artifacts (*e.g.*, models and features). Notebooks are commonly used for experiments. However, they are not recommended in production due to the difficulties of versioning, instrumentation, and automated execution. Hence, the platform should provide services to export notebooks into deployable pipelines (**R8**). Moreover, the platform needs to offer a service to record and query the metadata of each experiment to enable reproducing and troubleshooting experiments (**R9**). The typical metadata includes, but is not limited to, code versions, data versions, configuration files, output artifacts, and performance metrics.

Platform services for training may improve the performance and reliability of training jobs through strategies such as model check-pointing, distributed model training, exploitation of specific heterogeneous hardware (*e.g.*, GPU and TPU accelerators), prioritizing training activities, training on a slice of the data set, and AutoML (Automated Machine Learning) (**R10 and R11**). Check-pointing enables incrementally training a model using more iterations and recovering from failures during the training. The distributed training needs specific middleware capable of running a training job elastically over multiple compute nodes. A scheduler service can queue and prioritize the training jobs, enabling policing training activities, for example, capping the amount of data used by training or preventing long-running jobs from blocking the deployment of critical tasks such as security or bug fixes. Finally, AutoML can simplify and accelerate building ML

Table 1: Requirements for an MLOps Environment

Category	Requirements		Sources
Infrastructure	R1	Portability, reproducibility, and versionability	S2, S4, S12, S15, S18, S35
	R2	Auto-scaling and use of GPU and hardware accelerators	S3, S5, S19, S28, S35, S37, S38, S44, S52, S53
	R3	Cater for different environments (e.g., test and production)	S3, S2, S6, S17, S27
	R4	Manage the infrastructure using IaC (Infrastructure as Code)	S7, S12, S22, S54
Pipeline Development, and Execution	R5	Create, publish, discover, use, and customize pipelines	S3, S5, S17, S26, S29, S28, S32, S33, S34, S35, S42, S44, S47, S49, S53, S56, S58
	R6	Modular and reusable pipelines and components	S3, S14, S17, S19, S32, S41, S49, S58
	R7	Execution of pipelines via orchestration or choreography	S2, S7, S10, S26, S52, S53, S58
Experimentation, Training, and Testing	R8	Export experimental code from notebooks into pipelines	S4, S17, S22, S49
	R9	Record and query experiments and training runs	S7, S9, S16, S22, S23, S25, S30, S33, S38, S53, S58
	R10	Apply training scaling strategies such as model check-pointing, distributed training, use of hardware accelerators, and training with data set slices	S2, S3, S4, S8, S9, S10, S23, S24, S36, S38, S53
	R11	Use automated machine learning tools	S7, S10, S11, S16, S29, S58
	R12	Validation tests for all ML artifacts (e.g., model, code, and data)	S3, S4, S5, S9, S24, S25, S33, S39, S45, S47, S58
	R13	Prioritization and scheduling of tests and training jobs	S3, S4, S33
Model Deployment and Serving	R14	Ensure the compatibility of the model with the target infrastructure	S17, S22, S44, S58
	R15	Use open model formats for portable and flexible deployments	S7, S9, S10, S16, S24, S35, S38, S40, S57
	R16	Package models for ease of deployment, integration, and testing	S3, S17, S28, S48, S49
	R17	Support different patterns of deploying, releasing, and serving models	S1, S2, S7, S9, S17, S23, S24, S26, S28, S31, S33, S34, S35, S51, S58
Monitoring and Feedback Loops	R18	Realtime monitoring of and alerting for various issues in models, data, pipelines, and infrastructure	S1, S2, S3, S4, S7, S10, S11, S13, S15, S16, S17, S19, S20, S23, S25, S26, S28, S29, S31, S32, S33, S34, S35, S43, S44, S48, S52, S54, S57, S58
	R19	Automated triggering of corrective actions for alerts	S1, S2, S16, S17, S25, S35
	R20	Generation of custom actionable dashboards for models	S2, S3, S10, S34, S40, S54
	R21	Support different pipeline triggering models	S2, S3, S4, S5, S7, S9, S17, S19, S20
Model Life Cycle Management and Governance	R22	ML asset storage and marketplace	S1, S2, S3, S9, S16, S17, S18, S19, S21, S23, S26, S35, S50, S51, S53, S56, S58
	R23	Version control and lineage tracking for ML artifacts	S2, S4, S5, S7, S9, S20, S22, S23, S26, S25, S31, S33, S34, S35, S41, S45, S57, S58
	R24	ML asset metadata management	S9, S14, S16, S17, S24, S35, S49, S58
	R25	Access control and privacy compliance for ML artifacts	S2, S5, S7, S10, S11, S12, S24, S28, S54, S56, S58
	R26	Ensure adversarial robustness and interpretability of models	S2, S7, S10, S19, S22, S24, S34, S41, S43, S54, S58
General Platform Services	R27	Support for general data platform services such as data catalog, data storage, data discovery, data exploration, data augmentation/labeling, and data fusion	S7, S10, S11, S19, S23, S24, S31, S33, S36, S38, S50, S53, S58
	R28	Support for general ML platform services such as feature engineering, model exploration, model selection, hyper-parameter tuning, and model validation	S2, S4, S6, S9, S26, S33, S58, etc. (most articles)
CI/CD Automation	R29	Treat ML artifacts as first-class citizens in DevOps processes	S1, S2, S3, S4, S5, S7, S8, S9, S22, S25, S27, S29, S31, S32, S35, S45, S55, S57
	R30	Integrate unit, integration, and smoke tests of ML artifacts to CI/CD pipelines	S8, S9, S24, S25, S26, S33, S53, S58

applications and lower domain experts' barriers to developing their ML models.

All ML artifacts must be tested appropriately; the key testable artifacts are data, code, and model (**R12**). Data tests can verify that input data and features do not exhibit data quality issues such as malformed data, anomalies, and mismatches with the expected schemas and distributions. Such tests can prevent training-serving skew, namely differences in model performance during training and serving, which can be caused by differences between training data and serving data. Similar to testing of conventional software applications, source code tests can assess the quality of the ML code, *e.g.*, violation of best practices, existence of known defects, and resource efficiency of training tasks. Finally, the model validation tests can verify model fairness and consistency.

As model training and data processing can be expensive and time-consuming, the execution of tests needs to be prioritized and scheduled appropriately, for example, running a subset of tests or long-running testing during off-hours and training on small datasets (**R13**).

Deployment and Serving When deploying a model to the target serving environment, it is necessary to ensure its compatibility with the infrastructure regarding compute resources, software dependencies, and model formats (**R14**). As ML libraries may use specific formats for their models, a model translation service may be necessary to convert models into an open model format for portable and flexible deployments (**R15**). A platform can also offer an image-building service to package models, scoring code, and dependencies into container images (**R16**). In this way, the operational team can quickly deploy a model into staging and production environments, test it, and integrate it with the applications that need to consume the predictions from the model.

The platform must support different strategies for deploying/releasing and serving models (**R17**). Standard model deployment approaches include shadow, canary, and blue/green deployment. A shadow deployment does not immediately release the new model to users but instead uses the production traffic to test the model. A canary deployment releases the new model

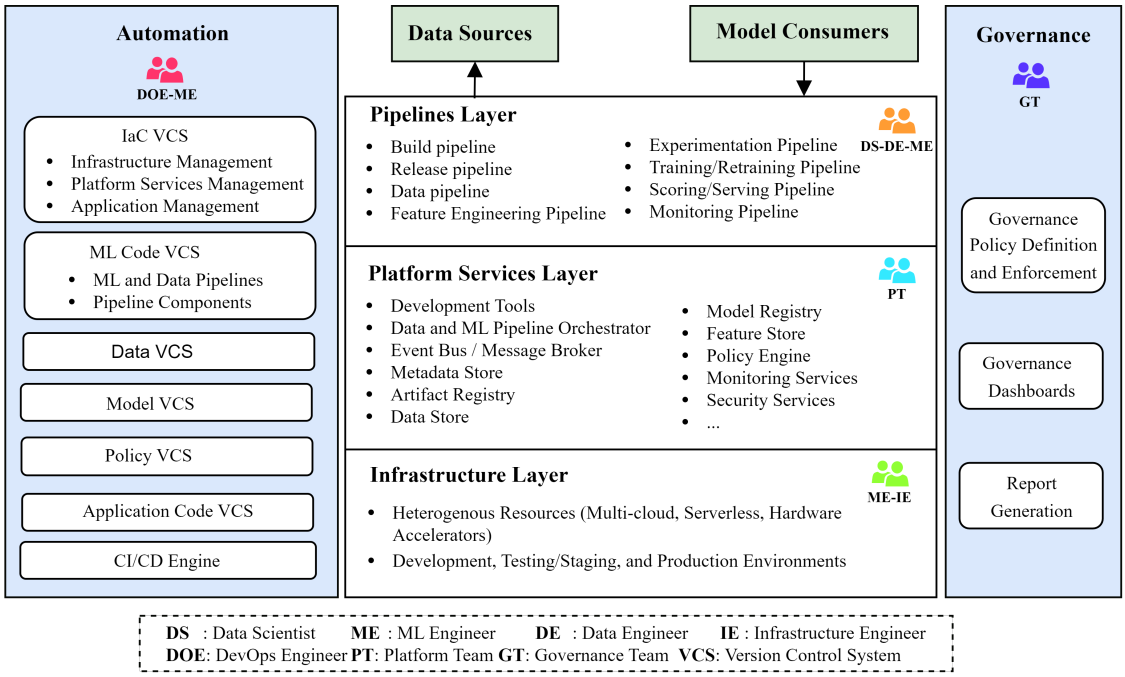

Figure 1: Layered Reference Architecture for an MLOps Environment

to users incrementally. In contrast, a blue/green deployment immediately deploys and releases the new model to users.

Common patterns from serving predictions from a model include *model-as-service*, *precompute*, and *model-as-dependency*. Model-as-service exposes the model as a web service or a messaging endpoint. Precompute compute predictions for a batch of input data and store them to serve the clients later. In the model-as-dependency pattern, the application embeds the model as a binary and loads it at runtime to make predictions. The application can also download the model at runtime from a model registry. Serving methods can also be classified into offline and online serving. Each method may need specific platform services, for example, a middleware to run batch and streaming data processing pipelines.

Monitoring and Feedback Loops The platform layer should continuously monitor various quality issues in models and data at runtime, generating alerts and triggering corrective actions (**R18**). Two typical quality issues are data drift and concept drift, which refer to the changes to the statistical properties of the model input and the target variable, respectively. Both can contribute to the degradation of the model performance over time, *i.e.*, *model drift*. The metrics related to the stability of the model need to be monitored continuously to detect drifts. The model needs to be retrained regularly in response to drift alerts to address the model drift. Pipelines also should be monitored continuously as their failures can prevent updating models. The logs of a pipeline can be collected and analyzed to gauge its health. Common performance metrics such as resource usage, execution time, and throughput should also be collected for pipelines and inference services. Such metrics can be used to trigger auto-scaling services to scale pipelines and models up and down.

Platform services can automate monitoring, altering, and enacting corrective actions (**R19**). For example, the developer should be able to develop and run a monitoring pipeline that can process logs and metrics and generate alerts, and define the rules to select and trigger actions on alerts. In addition, a dashboard service can support creating interactive dashboards that allow monitoring models, pipelines, and infrastructure and troubleshooting suspicious or poorly performing models and failed pipelines (**R20**).

Platform also needs to support common

approaches for triggering continuous training pipelines (**R21**): *metrics-driven*, *schedule-driven*, *event-driven*, and *ad-hoc manual*. In the metrics-driven approach, data and model performance metrics are measured and used to determine pipeline (re)execution. The schedule-driven strategy triggers the pipelines at a specified time or regularity. In the even-drive model, events, such as changes to the model's source code and the availability of a new training data set, trigger pipelines. Finally, a human operator can manually execute a pipeline.

Lifecycle Management and Governance Services ML artifacts and metadata must be versioned, stored, and managed to support their reproduction, discovery, auditing, and reuse (**R22-24**). Different artifacts may need specific storage components, *e.g.*, a model registry for models, a feature store for features, a pipeline store for data/ML pipelines, and a source code repository for ML and IaC scripts. The artifact metadata (*e.g.*, schemas, hyper-parameters and model metrics) can also be placed in the data store or a separate metadata store.

The platform should provide the services for enforcing policies governing ML artifacts' life cycle, for instance, identity and access management services for implementing access control policies, and privacy-preserving mechanisms (*e.g.*, data anonymization and federated learning) for enforcing data privacy compliance. Moreover, the practitioners may need tools for authoring, testing, and observing policies.

The models should be resilient to model attacks such as membership inference attacks, adversarial attacks, and model inversion attacks. Moreover, model decisions should be explainable and interpretable (**R26**). Hence, the practitioners need platform services to test models for their vulnerability to attacks and to generate and visualize explanations for model behavior.

Data Platform Services Developers use data pipelines to turn raw historical and online data from multiple sources into features for their ML models. A data pipeline typically considers tasks such as data ingestion, storage, discovery, standardization, labeling, cleaning, transformation, and feature extraction (**R27**). The platform should offer services to simplify the implementation of these tasks, *e.g.*, a messaging service and a data catalog service. Moreover, the platform also should provide tools for building, testing, orchestrating, monitoring, and managing data pipelines.

Pipelines Layer

This layer consists of the pipelines that can be built, tested, and executed using platform services and CI/CD automation services. From the gray literature, we identified eight major pipeline types in an MLOps environment: build, release (and deployment) , data , feature engineering, experimentation, training, scoring/serving, and monitoring. Build and release pipelines are CI/CD pipelines. The first pipeline builds the code, executes tests to verify code quality, and publishes the artifacts produced. The second pipeline operationalizes and promotes the artifacts across different environments (*e.g.*, staging and production) to enable their consumption and testing. When building ML and data pipelines, the source codes are pipeline models (*e.g.*, workflow configurations) and pipeline components (*e.g.*, Python programs). The pipeline models are generally published on the platform services of pipeline coordinators/engines as endpoints to enable their execution via API calls. In addition, the pipeline components are containerized and stored in an image registry. The same processes apply to platform services and scoring/prediction services.

Figure 2 shows the pipelines and (a subset of) their interconnections. Developers create and test pipelines and platform services, potentially reusing the relevant existing implementations. The deployment of pipelines and platform services may need provisioning and configuring compute stacks, which can be automated via IaC. All source codes should be version-controlled. Changes to the code can trigger build pipelines, which can result in publishing ML/data pipelines and container images. Let us consider the first-time execution of the training process. A data pipeline extracts the raw data sets from multiple sources, cleans, standardizes, and stores them in a data store. A feature engineering pipeline creates, selects, and stores features in a feature store. Finally, the training pipeline builds and tunes a model and publishes it to the model registry, which triggers the model release pipeline, which

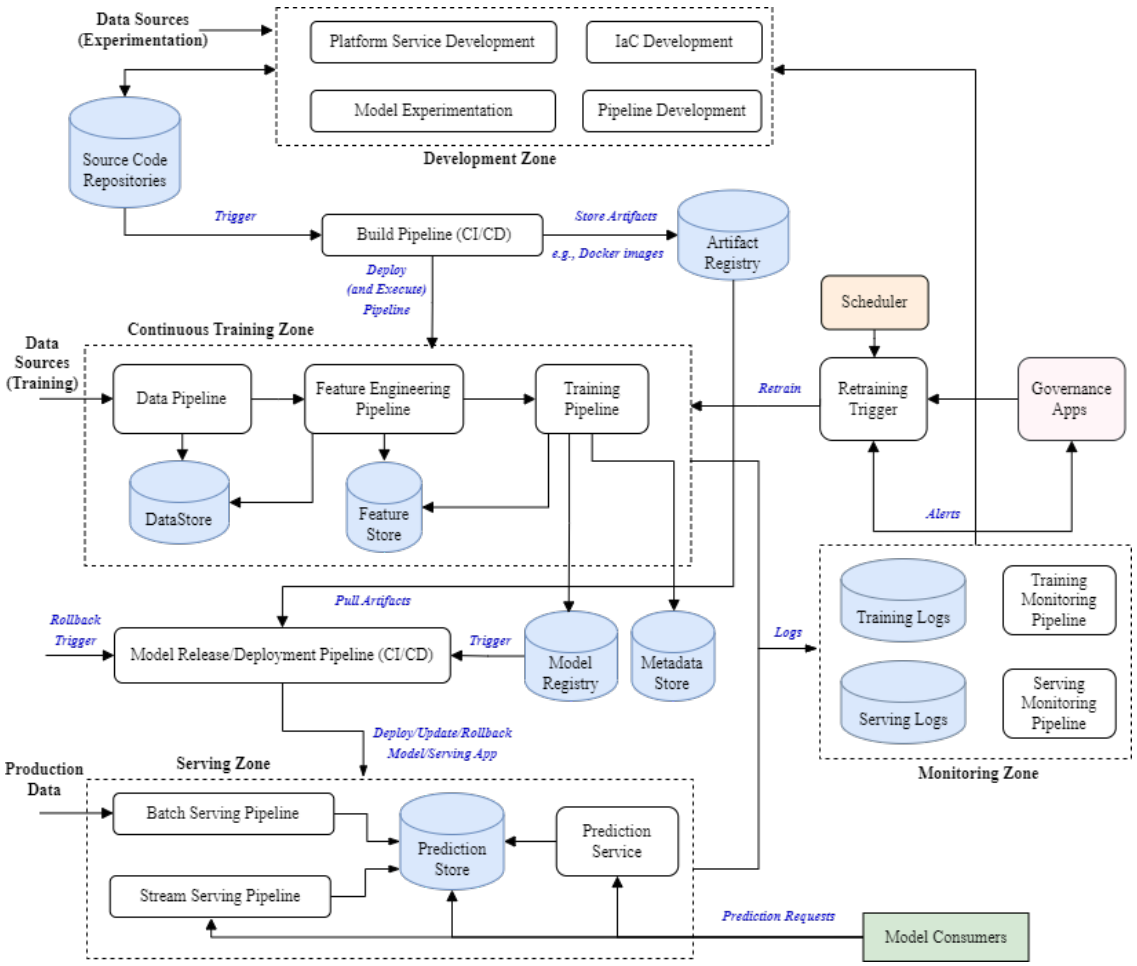

Figure 2: MLOps Pipelines and their Relationships

builds and deploys serving pipelines/services. A deployed model may serve client requests using batch, event-driven, or real-time serving methods. The logs from the execution of each step of the training and serving processes are continuously collected. The monitoring pipelines can analyze such logs and generate alerts indicating quality issues in models, data, pipelines, and infrastructure. Alerts can be used to trigger the re-execution of the training process and, consequently, the deployment of a new model.

Governance and Automation Cross-Cutting Modules

Governance needs the ability to consistently specify, configure, observe, and enforce policies for all ML artifacts and MLOps processes (*e.g.*, model approval and data collection/sharing). Policies can consider various metrics, for example, model accuracy, model fairness, and data privacy. Platform services can empower pipeline developers to use the *policy-as-code* approach for automating policy enforcement. For example, access control policies for controlling data and model access can be embedded into the stages of data and ML pipelines using a policy authoring service. Those policies can then be tested when developing pipelines using a policy testing service, and observed and enforced during their execution using a policy engine service and model/data monitoring services.

Automation applies CI/CD techniques and processes to automate the provisioning and configuration of infrastructure resources in target environments and the building, testing, deploying, and configuring of components such as platform services, pipelines, serving apps, and governance apps (**R29-30**). The relevant artifacts are stored

in version-control repositories, and their CI-CD Pipelines are triggered in various ways, *e.g.*, manual, event-driven, or scheduled.

Roles and Responsibilities

From the gray literature, we identified three key roles for MLOps team members: *data engineer*, *data scientist*, and *ML engineer*. Data engineers build, test, deploy, execute and manage data pipelines that pull raw data from various sources, validate, clean and standardize the raw data, and make the curated data accessible to the data scientists in a secure and timely manner. Data scientists analyze a business problem that needs a data science solution, and then build ML models that address the business problem. They typically work in an experimentation environment. ML engineers are responsible for operationalizing models developed by data scientists.

An MLOs team must interact with other individuals and groups, such as business analysts, DevOps engineers, application developers, and infrastructure engineers. Our reference architecture introduces two new teams: platform and governance. The former builds (or sourcing), tests, deploys and manages platform services. The latter defines and enforces governance policies using the services offered by the platform.

Related Work

Kolltveit and Li [4] reviewed 25 academic articles on MLOps, focusing on tooling and infrastructure aspects. John et al. [12] synthesized a maturity model for MLOps adoption, using the findings from the gray and academic literature. They also proposed an MLOps framework consisting of three main pipelines (*i.e.*, data, modeling, and release) and a governance layer. Warnett and Zdun [7] systematically reviewed 35 gray literature articles to identify design decisions for model deployment, where MLOps is a specific design option. Symeonidis et al. [13] surveyed the tools that support various tasks in MLOps, such as model deployment, experiment tracking, and feature engineering. They also identified several MLOps challenges, including pipeline development, retraining, and monitoring. Idowu et al. [6] compared 17 tools for managing ML assets such as data, models, pipelines, and experiments. Philipp et al. [14] introduced 22 requirements for MLOps and mapped 26 existing tools to them. They did not conduct a systematic literature review, and the requirements mainly consider general platform features such as data ingestion, versioning, model selection, model registry, and model performance monitoring.

Different from the work above, we aimed to determine the requirements for, and components of, a reference architecture for a complete MLOps stack (from infrastructure to applications) by systematically reviewing the relevant gray literature.

Conclusions

Organizations are adopting MLOps to accelerate the deployment and delivery of high-quality ML models and pipelines into production. Not surprisingly, there has been a rapid proliferation of gray literature on MLOps. This paper investigated the requirements and architectures for MLOps proposed by the gray literature. By systematically analyzing 58 sources, we distilled a catalog of 30 requirements and a reference architecture for MLOps environments. The requirements and reference architecture can provide a research framework for MLOps and guide identifying key research challenges in bringing ML models into widespread use. They can also help practitioners build or assemble an MLOps environment, select and adapt an existing MLOps environment, and select or develop tools for supporting various tasks in an MLOps environment.

REFERENCES

1. A. Cam, M. Chui, and B. Hall, “Global ai survey: Ai proves its worth, but few scale impact,” 2019.
2. A. Paleyes, R.-G. Urma, and N. D. Lawrence, “Challenges in deploying machine learning: A survey of case studies,” ACM Comput. Surv., apr 2022.
3. Algorithmia, “2020 state of enterprise machine learning,” Algorithmia, Tech. Rep., 2020.
4. A. B. Kolltveit and J. Li, “Operationalizing machine learning models - a systematic literature review,” in 2022 IEEE/ACM 1st International Workshop on Software Engineering for Responsible Artificial Intelligence (SE4RAI), 2022, pp. 1–8.
5. Thoughtworks, “Guide to evaluating mlops platforms,” Tech. Rep., 11 2021.
6. S. Idowu, D. Strüber, and T. Berger, “Asset management in machine learning: A survey,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering:

Software Engineering in Practice (ICSE-SEIP), 2021, pp. 51–60.
7. S. J. Warnett and U. Zdun, “Architectural design decisions for machine learning deployment,” in 2022 IEEE 19th International Conference on Software Architecture (ICSA), 2022, pp. 90–100.
8. S. Keele et al., “Guidelines for performing systematic literature reviews in software engineering,” Technical report, Ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.
9. V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” Information and Software Technology, vol. 106, pp. 101–121, 2019.
10. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering. Springer Science & Business Media, 2012.
11. Z. Dehghani, “Data mesh: Delivering data-driven value at scale,” 2022.
12. M. M. John, H. H. Olsson, and J. Bosch, “Towards mlops: A framework and maturity model,” in 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2021, pp. 1–8.
13. G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, “Mlops - definitions, tools and challenges,” in 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), 2022, pp. 0453–0460.
14. P. Ruf, M. Madan, C. Reich, and D. Ould-Abdeslam, “Demystifying mlops and presenting a recipe for the selection of open-source tools,” Applied Sciences, vol. 11, no. 19, 2021.

Indika Kumara is an Assistant Professor at the Jheronimus Academy of Data Science (JADS) and Tilburg University, the Netherlands.

Rowan Arts was a master student at JADS and Tilburg University, the Netherlands.

Dario Di Nucci is an Assistant Professor at University of Salerno, Italy.

Rick Kazman is a Professor at University of Hawaii and a Visiting Researcher at Software Engineering Institute of Carnegie Mellon University.

Willem-Jan Van Den Heuvel is a Full Professor at JADS and Tilburg University, the Netherlands.
Damian Andrew Tamburri is an Associate Professor at JADS and Eindhoven University of Technology, the Netherlands.