Computer Fraud and Security
ISSN (online): 1873-7056

Automated Model Management for Microservices: A CI/CD Approach

Wangdong Wu[1,*], Mingming Zhang[1], Xiben Min[2], Xiaoliang Zhang[3], Ling Zhuang[1], Jinhui Li[1]

[1]State Grid Jiangsu Electric Power Co., Ltd. Information & Telecommunication Branch, Nanjing 210024, Jiangsu, China

[2]China Electric Power Research Institute, Beijing 102299, China

[3]State Grid Information & Telecommunication Branch, Beijing 100052, China

**Corresponding Author.*

Abstract:

The rapid advancement of microservices architecture in software and internet applications has introduced significant challenges in managing distributed systems, particularly in maintaining operational efficiency and real-time responsiveness. This paper presents an innovative automated encapsulation and deployment framework tailored to address the frequent updates required by data-driven models in dynamic microservices environments. By integrating Docker containerization with Continuous Integration/Continuous Deployment (CI/CD) practices, the framework streamlines model management processes, enhancing both adaptability and operational efficiency. Leveraging modern tools such as Kubernetes and GitLab, we developed and tested a prototype system that facilitates rapid updates and robust management of intelligent models, ensuring their responsiveness to evolving data patterns. Empirical evaluations demonstrated substantial reductions in model deployment times and improved responsiveness to business requirements, while also identifying critical bottlenecks in traditional packaging and deployment methods. Our findings underscore significant improvements in operational efficiency and reduced manual intervention for model updates, with further optimizations to minimize deployment times and resource consumption. This research contributes a scalable and efficient solution for managing the lifecycle of models in microservices environments, addressing key challenges such as service granularity, dependency management, and rapid deployment needs, thereby enhancing overall system performance and reliability.

Keywords: microservice, CI/CD, DevOps, MLOps

INTRODUCTION

With the rapid advancement of software and Internet applications based on microservices architecture, the complexity of system management has significantly increased [1,2]. This architecture enhances the granularity and dynamism of services, particularly evident in data-driven intelligent model services. In the microservices scenario, ensuring the compatibility and performance of services across different environments is crucial to maintain their functionality in diverse configurations and infrastructures. Operational management requires continuous monitoring of service health and performance metrics, and an ability to promptly respond to potential faults or performance degradation [3-5]. These demands necessitate automation and real-time response capabilities to minimize manual intervention and enhance system efficiency.

Furthermore, due to the constant evolution and variation of data, intelligent models need to be updated within specific time frames to adapt to the latest data patterns and maintain optimal performance. Existing model management methods face numerous challenges in handling frequent service updates and rollbacks, with common update mechanisms often too slow to meet the needs for real-time responsiveness [6,7].

In microservices systems, the automation of model updates and deployments is particularly critical. Traditional model deployment processes, including code packaging, dependency installation, and environment configuration, are prone to errors and highly inefficient. Containerization technologies, such as encapsulating model code into Docker images [8], provide an effective means to realize Models-as-a-Service (MaaS). To further enhance the efficiency of updates and deployments, it is essential to simplify and automate multiple related steps, including model training, validation, packaging, and deployment. At this juncture, Continuous Integration and Continuous Deployment (CI/CD) become crucial [9-11]. Although current CI/CD pipelines are predominantly designed to automate the integration of software code, there is a lack of effective tools and mechanisms for integrating machine learning models. To address this gap, it is necessary to develop custom model metadata descriptions and deployment scripts that support the packaging and deployment of models, which are key to enabling rapid updates of intelligent model services.

Existing tools such as Kubeflow [12] and MLflow [13], while widely used in the Machine learning operations (MLOps) field [14], have certain limitations when addressing frequent model updates and deployments in microservices architectures. Kubeflow, despite its high scalability and support for complex workflows, has a complex deployment process and lacks flexibility, especially in handling rapid iterations and updates, involving lengthy resource configuration and management steps. MLflow, while simplifying the development process of machine learning projects through experiment tracking and model version management, primarily caters to standalone project management and struggles to cope with high-frequency automated updates in microservices architectures, lacking fine-grained deployment control and model packaging optimization. Researchers like Satvik Garg et al. [15] have discussed the implementation of CI/CD pipelines in MLOps methods, while Mattia Antonini and others have concentrated on Tiny-MLOps [16], focusing on the orchestration of ML applications in IoT systems. Although these studies provide valuable insights, they primarily focus on theoretical analysis or applications within specific domains. In contrast, this paper is more concerned with the implementation of frameworks that are quick to deploy and broadly applicable.

This paper proposes a framework that addresses these limitations, optimizing the automated encapsulation and rapid deployment of intelligent models in microservices environments, significantly enhancing update efficiency and flexibility. By integrating support from Kubernetes and GitLab, this paper designs and implements an automated model management prototype system, simplifying the management and deployment processes of models under microservices architectures. Experimental validation shows that the system is not only highly practical but also identifies bottlenecks in packaging and deployment through detailed performance testing, providing directions for future optimization.

The primary contributions of this paper are described as follows:

- The proposal and implementation of an automated model management framework that overcomes the shortcomings in flexibility and rapid updating found in existing tools, particularly suited for high-frequency iteration scenarios in microservices architectures.
- The design and implementation of a prototype system that validates the practicality of the framework and is supported by experimental data for effective deployment in real-world settings.
- Detailed performance analysis that identifies bottlenecks in model packaging and deployment processes, offering optimization strategies for rapid model updates in microservices environments.

RELATED WORK

This section explores the current state of machine learning model management, with a particular focus on its practical applications, challenges, and trends across various domains.

Evolution from DevOps to MLOps

DevOps optimizes the entire software development lifecycle, including development, testing, deployment, updates, and operational maintenance. By implementing Continuous Integration and Continuous Deployment, DevOps significantly accelerates software delivery and enhances quality [17,18]. The machine learning field has adopted these DevOps principles, evolving into MLOps, which focuses on automating the development, deployment, and maintenance of machine learning models to meet the needs for continuous iteration and optimization, especially enabling rapid adaptation to data changes and business strategy adjustments [19,20].

Continuous software engineering has become an emerging practice, emphasizing rapid feedback and frequent deployments to maintain agile responses to changing business needs. This process is supported by continuous integration, where code is frequently integrated into shared repositories, and continuous delivery, which ensures that code is always in a production-ready state [21,22]. These practices, combined with the principles of DevOps, lay the foundation for MLOps, which extends these methods to the machine learning domain, enabling faster model updates and improvements [11,23,24].

Challenges in MLOps

Managing machine learning models involves complex challenges, particularly in the areas of data acquisition, model training, deployment, and the iterative nature of model improvements. Traditional AI development often encounters issues such as complex dependencies and coupling between models, which can hinder the efficient management and scalability of models [25]. MLOps addresses these issues by providing structured workflows that not only streamline the development processes but also facilitate effective lifecycle management of models. The machine learning lifecycle typically includes business objective definition, data collection, model training, and deployment, and MLOps helps automate these steps [26,27]. A key challenge in machine learning lifecycle management is its continuous nature—models must be iterated and updated frequently based on changing data patterns, which differs from traditional software systems where long-term stability after deployment is expected [28-30].

Advanced MLOps Tools and Platforms

To support the end-to-end lifecycle of machine learning models, advanced tools like Kubeflow [12] and MLflow [13] have been developed. Kubeflow, an open-source platform based on Kubernetes, supports various machine learning operations from data validation to model serving, enabling machine learning workflows to operate seamlessly on any microservices platform equipped with Kubeflow [12,31,32]. MLflow simplifies the development and management of machine learning projects by focusing on aspects such as experiment tracking and model version management, critical for maintaining the integrity and efficacy of machine learning models over time [13]. However, despite their capabilities, these platforms often exhibit limitations in flexibility, struggling to adapt workflows based on specific needs or integrate with new tools and practices, which can impede the ability to tailor processes to evolving project requirements [33].

AUTOMATED ENCAPSULATION AND DEPLOYMENT ARCHITECTURE

In this study, we developed an automated model management system designed to address the challenges of updating and managing models and services within a microservices architecture. This system is composed of two core components: model encapsulation and deployment, enabling developers to conveniently manage the entire lifecycle of models. The overall system architecture is illustrated in Figure 1, which provides a visual representation of the components and their interactions within the system. The system's design aims to enhance the level of automation in model deployment, reduce the input of human resources, and ensure the speed and quality of updates. This section highlights the core technologies and working principles of model encapsulation, which are crucial for achieving rapid deployment and efficient operation of models.

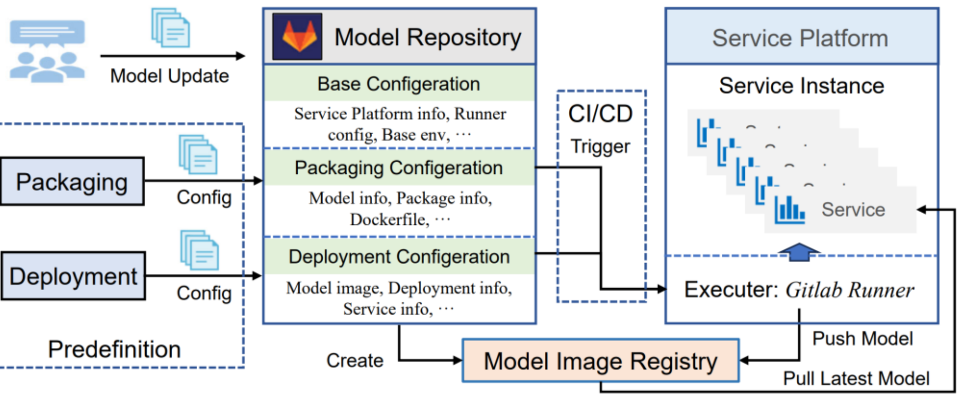


Figure 1. The architecture overview of automated encapsulation and deployment system

Model Encapsulation Component

The design of the model encapsulation component aims to create a unified and scalable method to support the consistent management and deployment of models with various architectures and functionalities within a microservices platform. The adopted technological strategy should ensure the efficiency of the encapsulation process and the convenience of subsequent model deployment. For this purpose, we have chosen Docker containerization technology, which not only provides an independent and secure operating environment for models but also significantly simplifies the complexity of deployment and maintenance.

During the encapsulation process, the first step is to decouple the model code from the business logic code to ensure the independence and flexibility of the model. This decoupling allows each model to be encapsulated into a separate Docker image and deployed on any Docker-supported microservices platform. Once deployed, the model interacts with external modules via REST APIs, enabling dynamic invocation of its functions and adaptive detection and maintenance of services.

To accommodate different types of service models, we employ a more general model encapsulation technique that involves the use of a specific Python Class for pre-packaging the models. This pre-packaging Python Class serves two main purposes:

Initialization and Parameter Loading: The Python Class aids in initializing the model and loading pre-trained parameters. This relies on the model's configuration files and pre-trained parameter files to ensure the reliability of the model. This step effectively prevents the redundant loading of training parameters, thereby enhancing the efficiency and speed of the model.

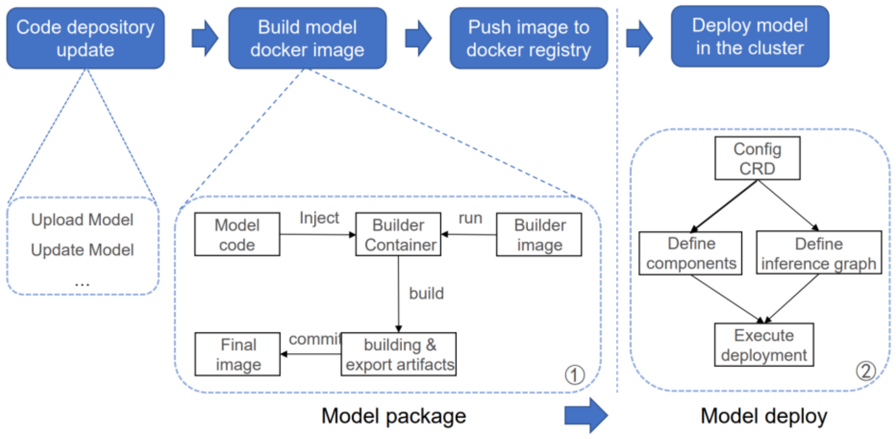


Figure 2. Process of model packaging and deploying component

Input Processing and Output Generation: Additionally, the Python Class can receive input and produce corresponding outputs. This capability allows us to encapsulate the model into a form that can be called by external services.

The working principle of the model encapsulation component is illustrated in Figure 2 part ①.

Technical Implementation Details

Decoupling and Service-Oriented Encapsulation: Initially, model code is decoupled from business logic code to ensure that the model can be updated and optimized independently of the existing services. Once decoupled, the model is encapsulated into a separate Docker image, allowing it to function as an independent microservice unit.

Unified Encapsulation Rules: We have developed a set of unified model encapsulation rules that define the entire process from model preparation to encapsulation completion. This includes pre-processing of the model, configuration of the dependency environment, and the final output format of the encapsulation. These encapsulation rules are primarily implemented through a GitLab Runner that runs on a microservice platform, with the core component being a base GitLab Runner image that supports Docker-in-Docker (DinD). This setup allows for the execution of encapsulation tasks within isolated Docker containers.

The process is controlled by the *Dockerfile* and executed via *.gitlab-ci.yml* in the GitLab repository. Upon submission, the GitLab Runner is triggered, following the defined steps in the *.gitlab-ci.yml* and *Dockerfile* to handle dependencies, package the model, and push the final image for deployment.

Our approach stands out by using Docker-in-Docker for task isolation and integrating fully with GitLab CI/CD for automated workflows. It's tailored for microservice architectures, offering modular, scalable management of models.

Application of Pre-encapsulation Technology: A specially developed Python class is used for model pre-encapsulation. This class is responsible for initializing the model, loading pre-trained parameters, and setting up the model's input and output interfaces. Pre-encapsulation not only optimizes the execution efficiency of the model within the container but also simplifies subsequent deployment and maintenance tasks.

Docker Encapsulation Process: After pre-encapsulation, the model is encapsulated through a specific Dockerfile. This Dockerfile configures all necessary environmental variables and includes the required libraries and runtime settings. During the encapsulation process, the Dockerfile also sets up network and storage interfaces to ensure that the model image can operate stably in different environments.

Management and Maintenance of Model Images: Once encapsulation is complete, the model images are uploaded to an image repository, which supports rapid iteration and deployment of model versions. The use of the image repository not only enhances the efficiency of model deployment but also facilitates model rollback and version control.

Based on the aforementioned technologies, the model encapsulation component not only improves the deployment speed and reliability of models but also ensures the scalability of the system and the convenience of maintenance. This efficient encapsulation technology provides robust technical support for the rapid iteration and stable operation of models.

Model Deployment Component

In this system, the model deployment component is primarily responsible for deploying machine learning models as production-level REST microservices using Docker images on a Kubernetes platform. This component not only manages the deployment process but also ensures the scalability of services and the continuous maintenance of API functionalities.

Technical Implementation and Principles: The model deployment component utilizes Kubernetes Custom Resource Definitions (CRD) to deploy model images. The deployment process is configured through parameters defined in a YAML configuration file, including the service name, Docker image, and the namespace in which it operates. The working principle of the model deployment component is illustrated in Figure 2 part ②. Below are the technical principles and key steps of the model deployment process:

Configuration Parsing and Submission: The YAML configuration file is submitted using the Kubernetes command-line tool, Kubectl. This file is initially sent to the Master node of the Kubernetes cluster.

Resource Scheduling: Upon receiving the YAML file, the Master node's Kube-Apiserver stores it in Etcd. Subsequently, the Kube-Scheduler is responsible for parsing the configuration file and deciding on the most suitable worker node for Pod deployment based on the cluster's resource utilization and the requirements specified in the file.

Pod Creation and Deployment: The Kube-Apiserver notifies the Kube-Controller-Manager of the scheduling decision, which then instructs the Kubelet on the worker node to create a Pod. The Kubelet invokes the node's Container Runtime to pull the specified Docker image and create the container, ultimately forming the Pod.

This deployment mechanism not only ensures the flexibility and scalability of model deployment but also optimizes the operational efficiency and stability of model services through automated container management. Once deployed, the model service is exposed on the port set during the encapsulation phase, allowing external services to interact with the model via REST APIs.

Continuous Integration and Continuous Deployment

The model deployment component also integrates capabilities for Continuous Integration (CI) and Continuous Deployment (CD), which are crucial for agile development. We utilize GitLab CI as the automation tool, managing the packaging and deployment of models through scripts defined in the *.gitlab-ci.yml* file. Whenever a new model file is committed to the GitLab repository, the GitLab CI Runner automatically triggers the defined workflow, enabling the automatic update and deployment of the model. This automated process significantly enhances the efficiency of model deployment, ensuring that model services can rapidly respond to changes in business requirements.

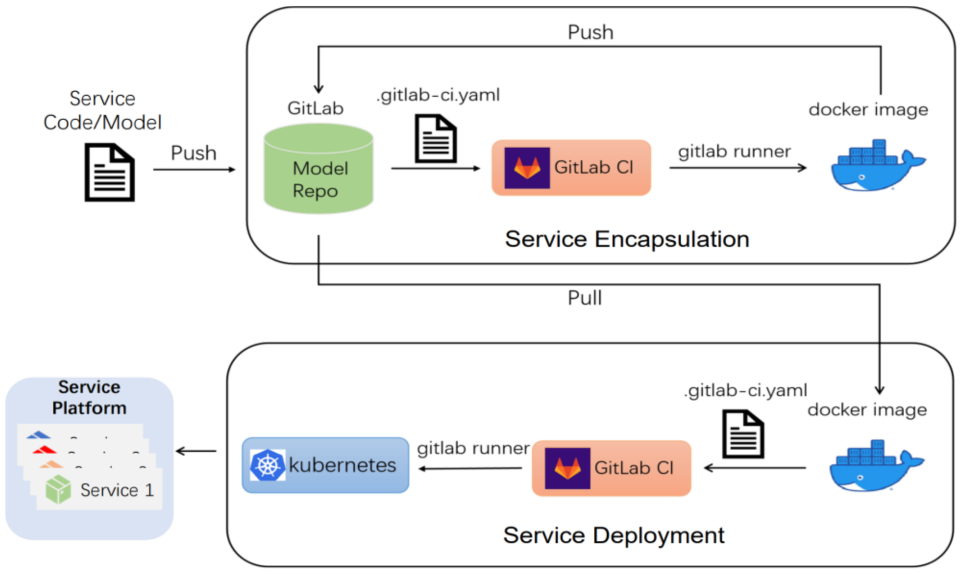


Figure 3. CI/CD-Based model packaging and deployment wrkflow diagram

Figure 3 illustrates the principle of the CI/CD-based model packaging and deployment process, providing a detailed visual representation of how components interact within the deployment pipeline. In the packaging process, the first step involves creating a model repository in GitLab, followed by configuring the GitLab Runner, which will handle the execution of CI/CD tasks. The packaging steps are defined in the *.gitlab-ci.yml* file within the repository. Once these configurations are in place, whenever a new model submission is detected, GitLab will automatically execute the packaging and image creation process as outlined in the YAML file. It will also push the built images to the appropriate registry and trigger the deployment process.

Similar to the packaging process, the deployment steps must also be defined in the same *.gitlab-ci.yml* file, specifying the deployment procedures and parameters. After the packaging process is completed and the deployment process is triggered, GitLab will initiate the configured runner to execute the defined deployment tasks, ensuring that the updated model is deployed successfully.

The use of GitLab CI/CD, combined with Docker-in-Docker (DinD), ensures an isolated and automated workflow for model packaging and deployment. This approach minimizes manual intervention and guarantees consistency in the execution environment. The system allows dynamic updates, automatically detecting new model submissions and triggering the complete CI/CD pipeline from packaging to deployment. This guarantees that the models are always up-to-date and that the integration with the microservices platform supports scalability and high availability.

Additionally, the deployment process is fully customizable, as it is defined within the *.gitlab-ci.yml* file. This enables teams to tailor the pipeline to specific project needs, allowing for flexibility in handling diverse environments, dependencies, or infrastructure changes. This adaptability and automation set our approach apart from traditional model deployment strategies, which often require more manual intervention and lack flexibility.

Through this highly automated deployment process, the system effectively manages and maintains models deployed on the microservices platform, while ensuring the high availability and scalability of model services.

EXPERIMENT

We conducted experiments on a microservices cluster built on a Kubernetes framework to validate our prototype system. Preliminary tests were carried out on the main functional components of the system.

Experimental Setup and Results

We deployed a Kubernetes cluster with ten nodes on local service machines; the cluster ran Kubernetes version v1.20.6. We managed the Kubernetes cluster using Rancher. In the cluster, we deployed Harbor to serve as the repository for models and model images. We developed GitLab as a repository for models and configuration files. Additionally, we deployed GitLab Runner within the cluster and configured it for packaging and deploying models.

In our study, we evaluated four distinct models tailored for different aspects of service operations within a microservices architecture. These included two models designed for multi-dimensional time series prediction: an LSTM model and a Transformer model. Additionally, a graph neural network (GNN) model was tested for its capability in classifying service nodes, addressing unique challenges in network management and service distribution. We also incorporated a Proximal Policy Optimization (PPO) model, which is used for service auto-scaling in microservices.

Table 1. Time consumption and percentage distribution of packaging and deployment processes

	Packaging Process		Deployment Process		Total
	Time (s)	Percentage (%)	Time (s)	Percentage (%)	Time (s)
LSTM	365	88.2	49	11.8	414
Transformer	491	85.4	84	14.6	575
GNN	305	81.1	71	18.9	376
PPO	336	85.5	57	14.5	393

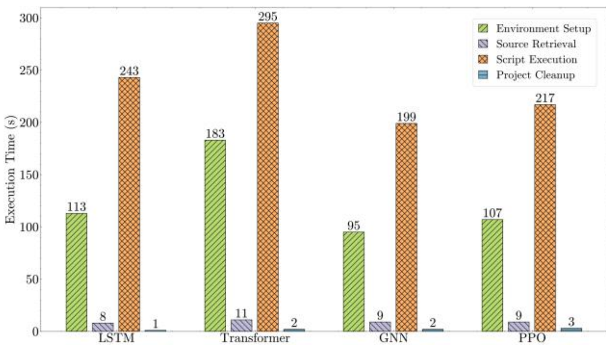
A model repository was prepared for these pre-trained LSTM, Transformer, GNN, and PPO models, with specific attention given to configuring packaging and deployment parameters to reflect their operational use cases. We meticulously documented the time expended on each process step, with detailed results presented in Table 1 to demonstrate the efficiency of each phase.

As depicted in Table 1, the packaging phase consumed the majority of the time, representing over 80% of the total time spent preparing the models for deployment. This phase was primarily extended due to the need for downloading extensive dependencies and constructing Docker images, which were subsequently pushed to the repository.

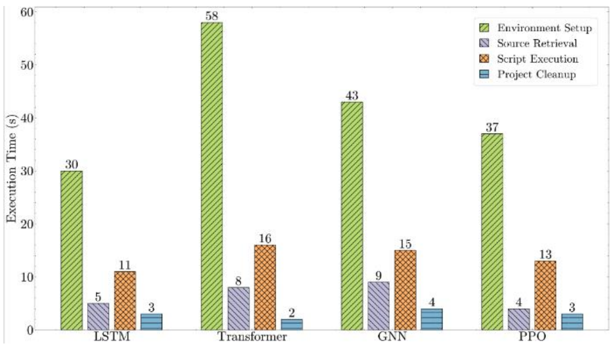
In contrast, the deployment phase proved much quicker, benefiting significantly from the local storage of model images which expedited the retrieval process. This highlights the advantages of local image storage in reducing deployment times.

The variability in time requirements across the models primarily stemmed from their dependency loads. Models with more complex dependencies required more setup time, and network speed variations further influenced these durations.

The exact execution times for each step of the packaging and deployment processes have been recorded, and the results are displayed in Figure 4. During the packaging phase, environment preparation and task script execution are the two most time-consuming steps. In contrast, during the deployment phase, most of the time is spent on environment preparation. We also calculated the percentage of time spent on each step, and the results are shown in Table 2.



(a) Packaging pipelines



(b) Deployment pipelines

Figure 4. Execution time for each step of packaging and deployment pipelines

The experimental data reveals that task script execution constitutes over 60% of the total time spent during the image packaging process. This phase primarily involves setting up the environment, installing dependencies, and constructing Docker images, all guided by configurations specified in the *.gitlab-ci.yml* file. This significant portion suggests that enhancing script efficiency could notably decrease overall packaging time.

Conversely, more than half of the deployment process time is dedicated to environment preparation, which includes creating and running runner Pods, pulling images, and starting up services. This phase's duration is largely dependent on the Kubernetes cluster's network conditions, highlighting a critical area for potential improvement.

To streamline these processes, we propose several optimization strategies:

Implementing a Shared Container Cache: By establishing a shared container cache within the Kubernetes cluster, we can effectively bypass the time-consuming image pulling steps, thereby accelerating the deployment process.

Pre-constructed Base Images: Developing base images pre-loaded with common dependencies and environment configurations can drastically reduce the setup time during the packaging phase. This approach minimizes the need for repeated installations and configurations across different deployment cycles, enhancing both efficiency and consistency.

Table 2. Main process time consumption and percentage in packaging and deployment phases

Model	Packaging Pipeline				Deployment Pipeline			
	LSTM	Transformer	GNN	PPO	LSTM	Transformer	GNN	PPO
Environment Preparation (s)	113	183	95	107	30	58	43	37
Percentage (%)	31.0%	37.3%	31.1%	31.8%	61.2%	69.0%	60.6%	64.9%
Source Code Retrieval (s)	8	11	9	9	5	8	9	4
Percentage (%)	2.2%	2.2%	3.0%	2.7%	10.2%	9.5%	12.7%	7.0%
Task Script Execution (s)	243	295	199	217	11	16	15	13
Percentage (%)	66.6%	60.1%	65.2%	64.6%	22.4%	19.0%	21.1%	22.8%
Project Cleanup (s)	1	2	2	3	3	2	4	3
Percentage (%)	0.3%	0.4%	0.7%	0.9%	6.1%	2.4%	5.6%	5.3%

To verify the effectiveness of this optimization, we conducted a simple test experiment using a pre-constructed base image based on the Transformer model. We built a base image that includes commonly used dependencies for such models, such as PyTorch, NumPy, and Scikit-learn. In the traditional method, dependencies and environment configurations had to be reinstalled for each packaging cycle, whereas the use of a pre-constructed image significantly reduced the packaging time. Experimental results showed that using the pre-constructed base image reduced the packaging time by over 40%, which not only improved overall efficiency but also eliminated potential inconsistencies caused by manual configurations.

These strategic improvements aim to not only speed up the deployment process but also ensure that the system, when handling complex tasks, can be updated and deployed with greater agility and fewer delays. This optimization is crucial for maintaining high performance in dynamic service environments, ensuring that the system remains robust and responsive to operational demands.

Based on the experimental results, the model management system implemented in this paper is capable of completing the automatic packaging and deployment of models in a relatively short period. Furthermore, the above process can be automatically initiated after initial configuration without manual intervention, greatly enhancing the convenience and efficiency of model updates and demonstrating the potential of the proposed model management framework.

CONCLUSION

The experimental results demonstrate that our automated model management system significantly optimizes the deployment and maintenance of machine learning models within a Kubernetes-based microservices framework. The packaging and deployment processes are notably accelerated, with packaging consuming the majority of the total time due to dependency management and image construction. Deployment times are minimized due to efficient local image retrieval strategies and the streamlined configuration of Kubernetes resources. By integrating tools like GitLab CI, the system supports agile development practices, offering rapid and reliable model updates. This study not only confirms the feasibility of the proposed model management framework but also highlights its potential to facilitate robust and scalable machine learning operations in cloud-native environments.

ACKNOWLEDGMENTS

This work was supported by the project "Research and Application of Cloud Infrastructure Adaptation Technology for XinChuang in State Grid," with project number 5700-202318307A-1-1-ZN. We are grateful for the resources and backing provided through this initiative, which significantly contributed to our research endeavors.

REFRENCES

[1] T. Cerny, M. J. Donahoo, and M. Trnka, Contextual understanding of microservice architecture: current and future directions, ACM SIGAPP Appl. Comput. Rev., 17(4) (2018), pp. 29–45.

[2] V. Singh and S. K. Peddoju, Container-based microservice architecture for cloud applications, in 2017 Intl. Conf. on Computing, Communication and Automation (ICCCA), IEEE, 2017, pp. 847–852.

[3] M. Jin, A. Lv, Y. Zhu, et al., An anomaly detection algorithm for microservice architecture based on robust principal component analysis, IEEE Access, 8 (2020), pp. 226397–226408.

[4] A. Hrusto, Towards Optimization of Anomaly Detection Using Autonomous Monitors in DevOps, (2022).

[5] L. Wang, S. Chen, and Q. He, Concept drift-based runtime reliability anomaly detection for edge services adaptation, IEEE Trans. Knowl. Data Eng., 35.12 (2021), pp. 12153–12166.

[6] Y. Dang, Q. Lin, and P. Huang, Aiops: Real-world challenges and research innovations, in 2019 IEEE/ACM 41st Intl. Conf. on Software Engineering: Companion Proc. (ICSE-Companion), IEEE, 2019, pp. 4–5.

[7] A. Masood and A. Hashmi, Aiops: Predictive analytics & machine learning in operations, in Cognitive Computing Recipes, Springer, 2019, pp. 359–382.

[8] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J., 2014(239) (2014), pp. 2.

[9] M. S. Arefeen and M. Schiller, Continuous Integration Using GitLab, Undergrad. Res. in Nat. and Clin. Sci. and Tech. J., 3 (2019), pp. 1–6.

[10] R. Schneider, Continuous integration: Improving software quality and reducing risk, Software Quality Professional, 10(4) (2008), pp. 51.

[11] J. Humble, Continuous delivery vs. continuous deployment, Available online: https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment (accessed on 15 June 2020).

[12] E. Bisong and E. Bisong, Kubeflow and Kubeflow Pipelines, Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners, 2019, pp. 671–685.

[13] M. Zaharia, A. Chen, A. Davidson, et al., Accelerating the machine learning lifecycle with MLflow, IEEE Data Eng. Bull., 41.4 (2018), pp. 39–45.

[14] Kreuzberger D, N. Kühl, and S. Hirschl, Machine learning operations (mlops): Overview, definition, and architecture, IEEE Access, 2023, 11: 31866-31879.

[15] S. Garg, P. Pundir, G. Rathee, et al., On continuous integration / continuous delivery for automated deployment of machine learning models using mlops, in 2021 IEEE Fourth Intl. Conf. on Artificial Intelligence and Knowledge Engineering (AIKE), IEEE, 2021, pp. 25–28.

[16] M. Antonini, M. Pincheira, M. Vecchio, et al., TinyMLOps: a framework for orchestrating ml applications at the far edge of iot systems, in 2022 IEEE Intl. Conf. on Evolving and Adaptive Intelligent Systems (EAIS), IEEE, 2022, pp. 1–8.

[17] L. Zhu, L. Bass, G. Champlin-Scharff, Devops and its practices, IEEE Software, 33(3) (2016), pp. 32–34.

[18] R. Jabbari, N. bin Ali, K. Petersen, et al., What is DevOps? A systematic mapping study on definitions and practices, Proc. of the scientific workshop proc. of XP2016, 2016.

[19] D. A. Tamburri, Sustainable mlops: Trends and challenges, in 2020 22nd Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, 2020, pp. 17–23.

[20] C. Ebert et al., Devops, IEEE Software, 33(3) (2016), pp. 94–100.

[21] B. Fitzgerald and K. J. Stol, Continuous Software Engineering: A Roadmap and Agenda, J. Syst. Softw., 2017, 123, pp. 176–189.

[22] J. Bosch, Continuous Software Engineering: An Introduction, in Continuous Software Engineering, Springer, Berlin/Heidelberg, Germany, 2014, pp. 3–13.

[23] I. Weber, S. Nepal, and L. Zhu, Developing Dependable and Secure Cloud Applications, IEEE Internet Comput., 2016, 20, pp. 74–79.

[24] 2015 State of DevOps Report, Available online: https://puppetlabs.com/2015-devops-report (accessed on 15 June 2020).

[25] G. Dong and H. Liu, Feature Engineering for Machine Learning and Data Analytics, CRC Press, 2018.

[26] M. Treveil et al., Introducing MLOps, O'Reilly Media, Inc., 2020.

[27] D. Sculley, G. Holt, D. Golovin, et al., Hidden technical debt in machine learning systems, Advances in neural information processing systems, 2015, 28, pp. 2503-2511.

[28] G. Fursin, H. Guillou, N. Essayan, CodeReef: An Open Platform for Portable MLOps, Reusable Automation Actions and Reproducible Benchmarking, arXiv preprint arXiv:200107935, 2020.

[29] M. Popp, Comprehensive Support of the Lifecycle of Machine Learning Models in Model Management Systems, Ph.D. dissertation, 2019.

[30] G. Fedoseev, A. Degtyarev, O. Iakushkina et al., A Continuous Integration System for MPD Root: Deployment and Setup in GitLab, Saint-Petersburg State University, 2016, pp. 525-529.

[31] Y. Zhou, Y. Yu, and B. Ding, Towards mlops: A case study of ml pipeline platform, in 2020 Intl. Conf. on Artificial Intelligence and Computer Engineering (ICAICE), IEEE, 2020, pp. 494–500.

[32] D. Baylor, E. Breck, H. T. Cheng, et al., TFX: A TensorFlow-based production-scale machine learning platform, in ACM SIGKDD Intl. Conf., 2017.

[33] C. Singh, N. S. Gaba, M. Kaur, et al., Comparison of Different CI/CD Tools Integrated with Cloud Platform, in 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence), IEEE, 2019.