AI Marketplace: Serving Environment for AI Solutions Using Kubernetes

Marc A. Riedlinger[1], Ruslan Bernijazov[2] and Fabian Hanke[2]
[1]*Fraunhofer IOSB-INA, Fraunhofer Institute of Optronics, System Technologies and Image Exploitation, Lemgo, Germany*
[2]*Fraunhofer IEM, Fraunhofer Institute for Mechatronic Systems Design, Paderborn, Germany*

Keywords: Kubernetes, Service Mesh, Artificial Intelligence, Machine Learning, Cloud-Native.

Abstract: Recent advances in the field of artificial intelligence (AI) provide numerous potentials for industrial companies. However, the adoption of AI in practice is still left behind. One of the main reasons is a lack of knowledge about possible AI application areas by industry experts. The AI Marketplace addresses this problem by providing a platform for the cooperation between industry experts and AI developers. An essential function of this platform is a serving environment that allows AI developers to present their solution to industry experts. The solutions are packaged in a uniform way and made accessible to all platform members via the serving environment. In this paper, we present the conceptual design of this environment, its implementation using Amazon Web Services, and illustrate its application on two exemplary use cases.

1 INTRODUCTION

Recent developments in the field of Machine Learning (ML) and related areas accelerate the adoption of ML in business as well as consumer-facing solutions. Established software companies start integrating ML into existing solutions and new start-ups that focus on specific artificial intelligence (AI) application areas are being formed. Especially in knowledge-intensive application areas, like product creation, utilization of ML promises great potentials in terms of efficiency gains and quality improvements (Dumitrescu et al., 2021), (Schräder et al., 2022). The term product creation refers to the development process of technical systems and consists of the four main cycles, namely, strategic product planning, product development, service development, and production system development. Figure 1 visualizes the main steps of this process based on (Gausemeier et al., 2019).

As of yet, the practical application of AI in product creation is still a major hurdle for the industry (Bernijazov et al., 2021). A key challenge for the adoption of ML solutions by industry companies is to build a common understanding of the capabilities of ML-based software solutions. Industry practitioners often lack the required ML expertise to assess how their processes can be supported by ML-based systems, whereas ML experts lack the required domain knowledge to identify suitable use cases for ML in product creation.

The project AI Marketplace[1] (AIM) addresses these challenges by building a platform that facilitates the collaboration between industrial companies with AI experts and AI solution providers. One of the envisioned functionalities of the platform is a serving environment that allows AI providers to demonstrate their solutions to possible industrial customers by uploading a demo version to the platform. That way, industrial customers can experience existing AI solutions via a web-based user interface (UI) and get in touch with the solution provider, if their business might benefit from a similar solution.

The development of this serving environment requires a standard format for the provisioning of AI solutions together with an infrastructure for the deployment and operation of the solutions. The standard must assure that all uploaded demonstrators can be handled in a uniform way by the platform but also be applicable to the majority of existing AI solutions to minimize the additional development overhead for solution providers. The infrastructure must be able to operate the uploaded demonstrators, support the addition and removal of them, and ensure that they do not interfere with each other. Moreover, the amount of available demo apps is not defined a priori and is expected to grow as AI is being adopted more and more

[1]ki-marktplatz.com

Riedlinger, M., Bernijazov, R. and Hanke, F.
AI Marketplace: Serving Environment for AI Solutions Using Kubernetes.
DOI: 10.5220/0011980800003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 269-276
ISBN: 978-989-758-650-7; ISSN: 2184-5042

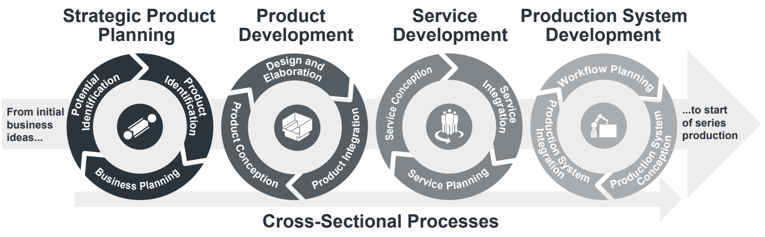


Figure 1: Illustration of the 4-cycle model of product creation. Adapted from (Gausemeier et al., 2019).

by industry. Therefore, the platform must be able to scale and handle increasing workloads in the future.

The present work explores the question of how such a platform could look like from a technological point of view. As a result, this paper contributes a production-ready serving environment for web applications based on Kubernetes (Google, 2014) along with a standardized packaging approach to facilitate the deployment of apps onto the provided platform.

The remainder of the paper is organized as follows. First, the related work is analyzed in Section 2. Next, the requirements towards the platform and the hosted AI apps are defined in Section 3. Thereafter, an implementation of the serving environment is outlined using a managed Kubernetes instance of Amazon Web Services (AWS) and validated using two different use cases of varying complexity regarding hosting in Section 4. Finally, the topic is concluded with an outlook about future work in Section 5.

2 RELATED WORK

In recent years, the adoption of AI solutions in small and medium-sized enterprises (SMEs) has been obstructed by the lack of expertise in developing and deploying ML models (Schauf and Neuburger, 2021), (Salum and Abd Rozan, 2016). As companies increasingly turn towards AI applications to support their business processes, the pressure to keep up with competition and meet customer demand has led to an increase in complexity and shorter production times (Kreuzberger et al., 2022), (Schauf and Neuburger, 2021), (Salum and Abd Rozan, 2016).

This section explores the current state of ML in SMEs and the challenges they face, as well as the potential solutions offered by machine learning operations (MLOps) to overcome these challenges (Kreuzberger et al., 2022). In contrast to usual software development processes, the development of ML solutions entails additional challenges. As highlighted by SCULLEY ET AL., there are specific ML-specific risk factors that must be taken into account in system design, such as boundary erosion, entanglement, hidden feedback loops, undeclared consumers, and data dependencies. Additionally, the software engineering framework of technical debt applies to ML systems as well (Sculley et al., 2015). ZHOU ET AL. also noted that the deployment of ML solutions can be challenging, especially when it comes to keeping the development phase short and offering the product to customers quickly (Zhou et al., 2020).

DevOps has seen increasing attention in recent years as organizations aim to bridge the gap between software development (Dev) and operations (Ops). MLOps is the consistent further development of DevOps to address the ML-specific challenges that DevOps does not meet. MLOps provides best practices, concepts, and the development culture needed to effectively manage the end-to-end life cycle of ML applications (Symeonidis et al., 2022). The emergence of MLOps from the DevOps approach has highlighted the unique challenges posed by ML, such as concerns around data complexity and model development and training. As MLOps offers a solution to these challenges, this led to the integration of MLOps processes into development (Mäkinen et al., 2021). ZHOU ET AL. present a functional ML platform built with DevOps capabilities using existing continuous integration (CI) and continuous delivery (CD) tools together with Kubeflow (Zhou et al., 2020).

In recent years, there has been a shift in the approach to deploying applications, with the current state of the art being the use of Docker (Merkel, 2014) containers along with a microservice architecture (Dragoni et al., 2016). On the one hand, Docker provides a Platform-as-a-Service (PaaS) solution via containers at the operating system level. On the other hand, the microservice architecture aims to combine small, self-contained units of system elements. When both concepts are combined, microservices are deployed as individual containers in a network, increasing the modularity of the system but also the complexity of the deployment process (Richardson, 2018). KARABEY AKSAKALLI ET AL. identified several challenges related to the deployment of microservices, such as architectural complexity and fault tolerance. Besides, further challenges were identified related to the communication concerns of a microservice platform, e.g. service discovery, and load balancing (Karabey Aksakalli et al., 2021).

Moreover, companies expressing growing interest in cloud computing have identified its potential benefits, as well as the challenges in deploying software on Infrastructure-as-a-Service (IaaS) platforms (Karabey Aksakalli et al., 2021). Cloud computing offers many benefits, such as on-demand use and the ability to scale infrastructure according to needs. However, deploying software on IaaS platforms can be challenging and requires expertise in operations. SMEs often lack the personnel to put these services into operation (Schauf and Neuburger, 2021), (Salum and Abd Rozan, 2016).

In addition to traditional DevOps and MLOps tools, several specialized tools have been developed specifically for managing ML models. Container orchestration tools are also playing a key role in deploying scalable solutions. These technologies, in combination with cloud computing environments, enable the full potential of microservices to be leveraged. Hereby, Kubernetes is one of the most widely used open-source container orchestration platforms. Developed by Google in 2014 and now operated by the Cloud Native Computing Foundation (CNCF), Kubernetes is designed to automate the deployment, scaling, and management of containerized applications (Google, 2014). It is often used as the foundation for building MLOps platforms and workflows. In the following, multiple of these frameworks for serving AI solutions are presented.

KServe originated from Kubeflow and is now a stand-alone Kubernetes-native model serving platform. It allows for the deployment and management of ML models and is designed to work seamlessly with Kubernetes and other cloud-based technologies to provide a complete MLOps solution (Kubeflow Serving Working Group, 2021).

Seldon Core is another popular platform for deploying and managing ML models on Kubernetes. It provides a set of tools and frameworks for building, deploying, and monitoring ML models in production (Seldon Technologies Ltd., 2018).

BentoML is an open-source platform for deploying ML models to various serving platforms, including Kubernetes. It provides a consistent interface for packaging and deploying models, as well as tools for monitoring and managing deployed models. BentoML also facilitates the integration with other MLOps tools and workflows (BentoML, 2019).

Overall, these tools provide a variety of options for deploying and managing ML models on Kubernetes and other cloud-based platforms. They help to automate many of the complex tasks associated with MLOps and provide a more efficient and effective way to manage ML models in production.

However, in contrast to the described frameworks for providing AI solutions, the solution presented in this paper aims at providing a generalized approach. The presented frameworks, such as KServe, focus on the deployment of ML models, but, generally-speaking, an AI solution does not necessarily consist of an ML model alone. With the generic approach presented in this work, it is possible to deploy various software components necessary for web-based AI solutions. For instance, this could be a web-based front end that is linked to an AI algorithm.

3 CONCEPT

The motivation for the present work is to design a platform for the AIM that enables the hosting of an undefined number of AI solutions. Hereby, these solutions are created by AI authors and serve as demonstrators that are publicly hosted on the platform.

On a conceptual level, the requirements towards such a serving environment are addressed in the next sub-section which is followed by the packaging approach that unifies hosting of AI applications.

3.1 Requirements and Implications

An AI solution can be everything from a mere ML model with a web interface to a full-fledged web application. Moreover, the serving environment shall host an indefinite number of these solutions. Therefore, the environment has to be hosted using a cloud-provider offering this degree of elasticity. This way, computing resources can automatically be added on-demand which prevents manual scaling of the cluster.

As a result of being a cloud-deployment, the serving environment is designed in accordance with the cloud-native principles as postulated by the CNCF (CNCF, 2018b), such as observability, security and loose coupling. Furthermore, to design a cloud-platform with inherent interoperability, only open-source components of the CNCF cloud-native landscape (CNCF, 2018a) are chosen. Additionally, a service mesh using the open-source container orchestration system Kubernetes is considered to achieve an observable and secure platform. Hereby, AI apps are deployed as isolated microservices which results in a decoupled architecture.

Besides, by choosing Kubernetes as orchestration system, a certain level of independence from cloud-providers is baked into the platform as additional benefit. Moreover, the whole infrastructure of the serving environment is laid out in code which facilitates its automatic deployment.

Furthermore, the traffic between the user of an AI service and the service itself has to be encrypted using transport layer security (TLS) to avoid potentially sensitive data from being transmitted in plain text. In addition to that, the communication between microservices inside the Kubernetes cluster shall also be encrypted to maximize security.

3.2 Standardized Solution Packaging

The AI solutions must follow a standardized packaging in order to be deployed on the serving environment. Hereby, the requirements demanded by the packaging process are designed to be only as restrictive as necessary to provide maximum freedom to the AI developer. On the other hand, certain restrictions have to be employed to achieve a unified deployment.

Since the AI applications are hosted on the cloud, resources are expensive. Therefore, it is recommended that apps should be as conservative as possible in terms of resource consumption. Furthermore, efficient apps have a minimal latency on startup and teardown which improves the scaling process and, in the end, benefits the user experience.

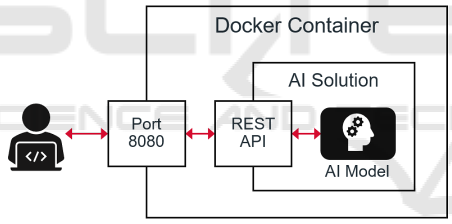


Figure 2: Solution Packaging.

Figure 2 illustrates the standard format of how an AI app has to be packaged so that it can be deployed on the serving environment. First, the application and its logic or AI model can be created with any language and AI framework as long as a representational state transfer (REST) application programming interface (API) can be integrated. The REST API is a widely used and platform-agnostic standard in web development and therefore well suited as interface implementation. The API denotes the interface to talk to the AI model beneath and provides all relevant functionalities to work with it. Second, the AI application along with the API are wrapped by a Docker image. This image is responsible for storing all necessary dependencies to run the solution once a container is instantiated. Thereby, the Docker container must expose port 8080 that is used to consume external traffic which is then routed to the application.

Another requirement to the AI application is that it has to host a web page under its root path. This way, a user accessing the app via the browser sees a visual representation of what's included in the solution. For example, a web page displaying the OpenAPI specification (OpenAPI Initiative, 2011) of the REST API along with the functionality to test each given endpoint is well-suited under the root path (see Fig. 6).

Lastly, the AI solution has to be stateless. Therefore, no data should be aggregated within the application. The rationale behind this restriction is two-fold. On the one side, storage, i.e. a database, does not scale similarly to other services since only one single source of truth is allowed. Different concepts are needed to accomplish scaling of storage in contrast to scaling stateless apps. On the other side, the Docker container could grow indefinitely which complicates the scheduling on resource-bounded cloud resources. This issue could be solved by providing a separate storage service which manages the state of stateful AI apps and scales on its own. However, in this paper, the focus is on stateless applications and incorporating stateful apps is left for future work.

4 IMPLEMENTATION

The following section demonstrates the implementation of the serving environment using a managed Kubernetes instance of AWS. First, the architectural design of the environment is provided. Next, the deployment of two different use cases are discussed in the follow-up sub-section.

4.1 Serving Environment Architecture

The serving environment is designed as microservice architecture that is deployed onto a Kubernetes cluster. Consequently, each AI application is deployed as separate service which allows automated scaling of these services only where it is needed. Hereby, Kubernetes provides the heavy lifting of managing the scaling of the individual services and the cluster as a whole when further compute resources are needed.

A high-level overview of the architecture of the serving environment including its main components is given in Fig. 3. The Kubernetes cluster is divided into two different node groups *General* and *Solutions*. Thereby, each node group is associated to one instance type that defines the compute and storage capabilities, processor type and operating system of the underlying virtual machine (VM). Within a node group, only nodes of the associated instance type can be created. Kubernetes automates the scaling of node groups whenever compute resources have to be added

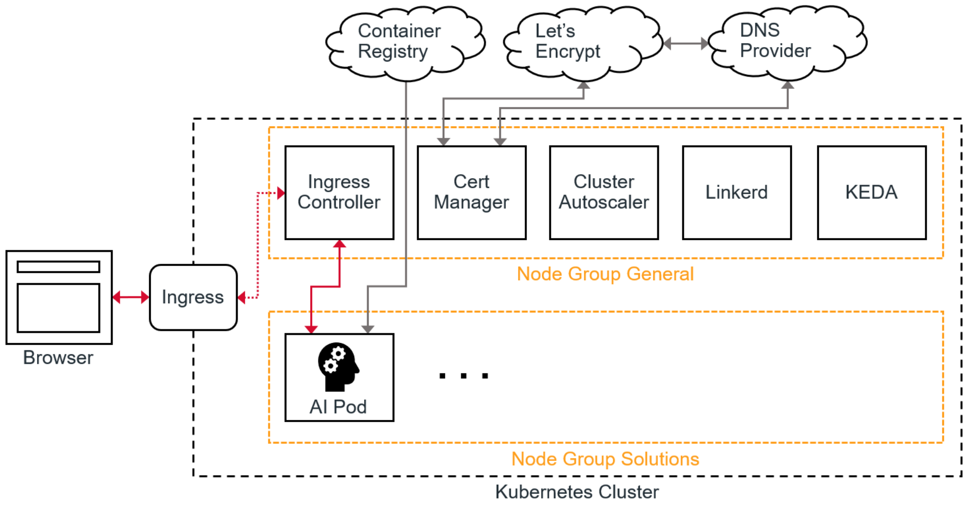


Figure 3: Serving Environment Architecture.

or removed. The clear distinction between the two node groups is achieved with Kubernetes taints, tolerations and affinities. These settings can be attached to node groups and pods to avoid that pods are scheduled in the wrong node group.

Node group *General* hosts general Kubernetes pods (visualized as rectangles) that provide core functionality, such as traffic routing, or extend the features of the cluster, such as TLS, auto-scaling, and service mesh capabilities. On the other hand, node group *Solutions* only hosts AI pods and therefore needs a more powerful VM type than *General*. If further node groups are required to match different resource requirements, they can be added to the cluster alongside *Solutions*. By providing different node groups, AI authors can select the group that best suits the needs of their AI solutions. Additionally, only related AI pods are hosted per node to avoid having underutilized nodes after AI solutions are shut down.

The entry point of external traffic, e.g. a browser or a script, into the cluster is denoted by the *Ingress* resource sitting at the edge of the Kubernetes cluster and defining the mapping between routes and corresponding services. Each ingress resource is associated with an *Ingress Controller* that actually does the traffic routing defined in the ingress resource. Hereby, certain limiters such as rate limits or request size limits can be employed in the controller to protect the internal AI services from becoming exhausted.

The *Cert Manager* (Jetstack, 2017) pod is responsible for generating a free certificate issued by *Let's Encrypt* and automatically keeping the certificate up-to-date. For the serving environment, a wildcard certificate is utilized which allows TLS communication not only for the main domain associated to the environment, but also for sub-domains which are created for each AI solution. As a result, the communication between the calling party and the *Ingress Controller* is encrypted and secure. Due to the usage of a wildcard certificate, *Cert Manager* needs access to the domain name system (DNS) provider where the domain is registered so the DNS challenge can be conducted. This is needed by *Let's Encrypt* to verify that the domain belongs to the party requesting the certificate.

The *Cluster Autoscaler* enables automatic scaling of the node groups when there are not enough compute resources for incoming workloads such as freshly scheduled pods. Additionally, the *Cluster Autoscaler* also terminates underutilized pods to improve costs.

Linkerd (Buoyant, 2016) is a light-weight and fast service mesh implementation. It improves the observability of a cluster by making the traffic between pods visible. This is achieved by injecting sidecar proxies into the respective pods. This meshing of pods is exemplarily visualized for the ingress pod and an AI pod in Fig. 4. Hereby, both containers do not directly communicate with each other anymore, but via proxy containers that gather traffic metrics alongside. By using *Linkerd*, these performance metrics are stored to the Prometheus database (SoundCloud, 2012) and can also be visualized in the *Linkerd* dashboard.

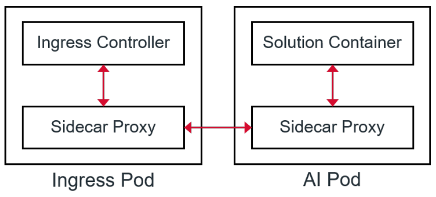


Figure 4: Meshing of Pods according to Linkerd.

Furthermore, since TLS is terminated at the

Ingress Controller, the internal communication inside the cluster takes place unsecured via the hypertext transfer protocol. To remedy this security concern, *Linkerd* provides mutual TLS for cluster-internal communication. Thereby, the communication between two proxies (see Fig. 4) is bi-directionally encrypted adding an extra layer of security to the cluster in case a node becomes compromised.

Kubernetes Event-driven Autoscaling (*KEDA*) (Microsoft and Red Hat, 2019) extends the conditions upon which auto-scaling of pods is triggered. By default, the Kubernetes Horizontal Pod Autoscaler (HPA) can only scale pods based on their memory or CPU consumption. As a result, traffic-based scaling is not possible without an extension like *KEDA*. Furthermore, HPA cannot scale down to zero pods. Therefore, at least one replica of a pod must always remain active even though it might not be needed. *KEDA* provides scaling down to zero and also scaling up from zero maximizing cost efficiency.

Traffic-based scaling is realized by having the *Ingress Controller* store request-related events to Prometheus. Subsequently, each request to the subdomains, where the AI solutions are hosted, are stored along with a timestamp. From there, *KEDA* can query those metrics and decide about scaling. For example, each AI pod that has not received any requests in the past 30 minutes is removed. Since only related AI pods are hosted per node, all pods are terminated and consequently, the node is removed by the *Cluster Autoscaler* as well afterwards which optimizes costs. On the other hand, *KEDA* detects requests regarding terminated pods and re-schedules them accordingly.

The AI applications are hosted in the node group *Solutions*. However, before a new AI app can be hosted, it needs to be packaged according to the standardized solution packaging presented in Section 3.2. After packaging, the AI app has to be uploaded to a *Container Registry* the Kubernetes cluster has access to. Next, the new AI service has to be added to the cluster via a Kubernetes manifest. Finally, the *Ingress* resource needs to be extended and re-applied to the cluster to include the routing to the new AI service.

4.2 Use Cases

This section demonstrates two exemplary engineering apps of varying complexity that are hosted on the serving environment presented in the last section.

On the one hand, this demonstrates the capabilities of the serving environment but also serves as guidance about how an actual AI application has to be designed to be hosted on the platform.

4.2.1 Natural Language Processing

The first use case addresses the hosting of a natural language processing (NLP) model. Such models are, for example, often employed to analyze requirements in the scope of strategic product planning and product development. Possible applications are the classification of existing requirements or the search for similar requirements from prior projects. In our use case, the Universal Sentence Encoder (USE) (Cer et al., 2018) is utilized to compute embeddings of text. These embeddings can then, in a follow-up step, be used to compute the similarity between two pieces of text.

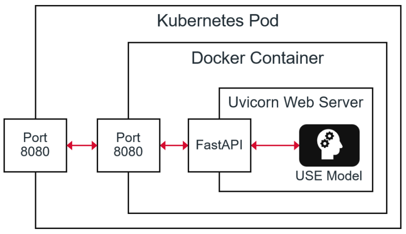
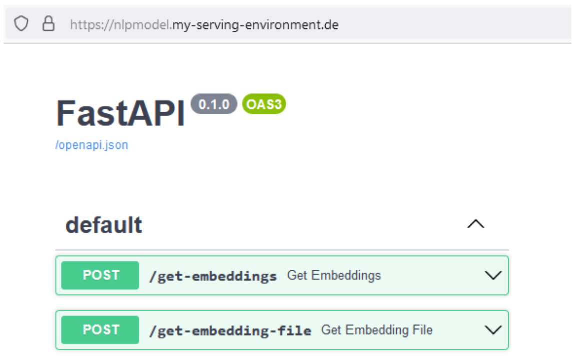


Figure 5: USE Deployment.

Figure 5 illustrates how the USE web application is designed in order to be hosted on the serving environment. For that, the TensorFlow (Abadi et al., 2015) model of the USE is wrapped in a Python-based REST API using the FastAPI library (Ramírez et al., 2018). This API exposes two different endpoints, one that expects a mere string input and another for taking care of text files. In summary, these endpoints take the input from the request, compute the embeddings and return them to the caller.

Following the standardized packaging established in Section 3.2, the whole application is packaged as Docker image along with all necessary dependencies for execution. Afterwards, the resulting image is pushed to an AWS container registry accessible by the serving environment. Once the image is downloaded to the environment and executed, the REST API is made available on a Uvicorn web server


https://nlpmodel.my-serving-environment.de

FastAPI 0.1.0 OAS3

/openapi.json

default

POST /get-embeddings Get Embeddings

POST /get-embedding-file Get Embedding File

Figure 6: USE Swagger UI.

(Christie et al., 2017) in the container that is hosted within the scheduled Kubernetes pod.

FastAPI automatically generates a Swagger UI from the specified REST API. This UI of the USE web application is depicted in Fig. 6. Moreover, the given endpoints can also directly be tested in the browser using that web page facilitating quick testing.

Users can integrate this service in their apps by calling the API functions and using the embeddings e.g. for a similarity analysis. Also, this hides the complexity of creating such a service from the user.

4.2.2 Intelligent Part Comparison

The intelligent part comparer is a tool with which CAD models can be evaluated with respect to their similarity. The evaluation is mainly based on the geometry of the models. This AI-based application intends to relieve the challenges coming with the management of identical parts which is especially a concern in companies that work in product development.

The web application consists of three components, a JavaScript web UI the user can interact with (see Fig. 8), a Python-based Flask server (Grinberg, 2018) that only exposes endpoints of the part comparer meant to be publicly available, and an ASP.NET server (Microsoft, 2002) which provides the functionality to conduct the comparison of CAD models. Besides, the ASP.NET server holds a static collection of CAD models used to perform the comparison against.

Figure 7 illustrates how the components of the part comparer are mapped to Kubernetes in order to be hosted on the serving environment. Thereby, all services are deployed on individual Kubernetes pods to allow them to be scaled separately. Furthermore, each service follows the standardized packaging presented in Section 3.2 to facilitate the deployment and resides as Docker image in an AWS container registry accessible by the serving environment.

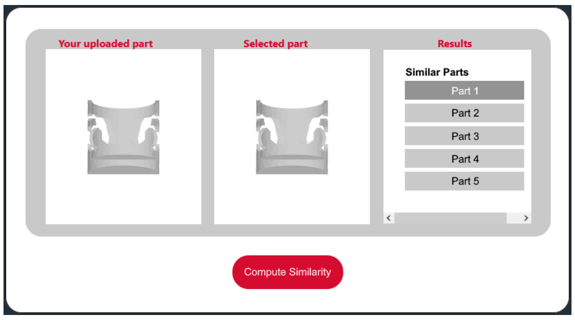


Figure 8: Intelligent Part Comparer UI.

Once a user opens the web link of the part comparer in the browser, it will be redirected to the web UI service as depicted in Fig. 8. Then, the web page is downloaded to the browser of the user and rendered afterwards. Thereafter, the user can interact with the web page by uploading a CAD file and pressing the *Compute Similarity* button to receive a list of similar CAD models. In the background, the web UI calls functions exposed by the Flask server's public REST API. Finally, the Flask server calls the private comparison function of the cluster-internal ASP.NET server and returns the results back to the user's browser.

5 CONCLUSION

Bringing AI applications into production is a challenging and costly task. In this paper, a serving environment is outlined that can be used to host different AI applications in a uniform way. It allows AI developers delivering their AI apps in a cost-effective way and thereby increases the overall accessibility of AI. The serving environment is implemented by utilizing a managed Kubernetes instance of AWS and tested with multiple AI applications. In future work, further experiments are planned to gain a better understand-

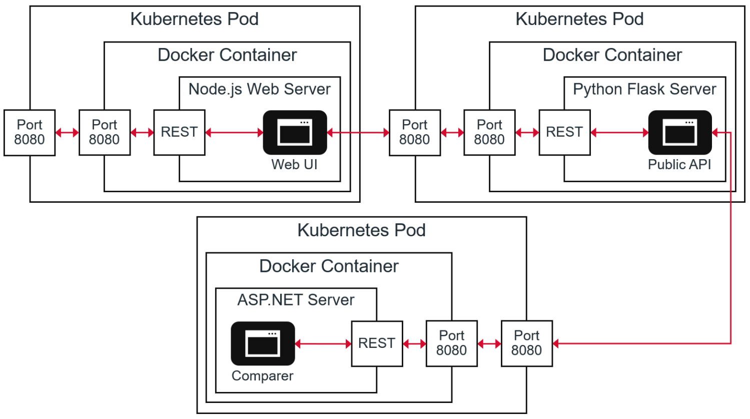


Figure 7: Intelligent Part Comparer Deployment.

ing of the limitations of the approach (e.g. with respect to scalability). Moreover, various functional improvements are to be done, like the integration of Kubernetes *StatefulSets* to provide storage services and the integration of role-based access control (RBAC) for the applications. RBAC can be realized, for example, using the identity and access management solution Keycloak (Red Hat, 2014).

ACKNOWLEDGMENT

This work was supported by the research and development project "AI Marketplace" which is funded by the Federal Ministry for Economic Affairs and Climate Action in Germany.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.

BentoML (2019). BentoML. https://www.bentoml.com (accessed 9 December 2022).

Bernijazov, R., Dicks, A., Dumitrescu, R., Foullois, M., Hanselle, J. M., Hüllermeier, E., Karakaya, G., Ködding, P., Lohweg, V., Malatyali, M., and et al. (2021). A meta-review on artificial intelligence in product creation. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*.

Buoyant (2016). Linkerd. https://linkerd.io (accessed 28 November 2022).

Cer, D., Yang, Y., Kong, S.-y., Hua, N., et al. (2018). Universal sentence encoder.

Christie, T. et al. (2017). Uvicorn. https://www.uvicorn.org (accessed 28 November 2022).

CNCF (2018a). Cloud Native Landscape. https://landscape.cncf.io (accessed 28 November 2022).

CNCF (2018b). Trail Map. https://github.com/cncf/trail map (accessed 28 November 2022).

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: yesterday, today, and tomorrow.

Dumitrescu, R., Riedel, O., Gausemeier, J., Albers, A., and (Eds.), R. (2021). Engineering in germany – status quo in business and science.

Gausemeier, J., Dumitrescu, R., Echterfeld, J., Pfänder, T., Steffen, D., and Thielemann, F. (2019). *Innovationen für die Märkte von morgen: Strategische Planung von Produkten, Dienstleistungen und Geschäftsmodellen*. Hanser, München.

Google (2014). Kubernetes. https://kubernetes.io (accessed 28 November 2022).

Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc.

Jetstack (2017). cert-manager. https://cert-manager.io (accessed 28 November 2022).

Karabey Aksakalli, I., Çelik, T., Can, A., and Tekinerdogan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180.

Kreuzberger, D., Kühl, N., and Hirschl, S. (2022). Machine learning operations (mlops): Overview, definition, and architecture.

Kubeflow Serving Working Group (2021). KServe. https://kserve.github.io/website (accessed 9 December 2022).

Mäkinen, S., Skogström, H., Laaksonen, E., and Mikkonen, T. (2021). Who needs mlops: What data scientists seek to accomplish and how can mlops help?

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*.

Microsoft (2002). ASP.NET. https://dotnet.microsoft.com/en-us/apps/aspnet (accessed 9 January 2023).

Microsoft and Red Hat (2019). KEDA. https://keda.sh (accessed 28 November 2022).

OpenAPI Initiative (2011). OpenAPI. https://www.open apis.org (accessed 28 November 2022).

Ramírez, S. et al. (2018). FastAPI. https://fastapi.tiangolo.com (accessed 28 November 2022).

Red Hat (2014). Keycloak. https://www.keycloak.org (accessed 3 March 2023).

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning.

Salum, K. and Abd Rozan, M. Z. (2016). Exploring the challenge impacted smes to adopt cloud erp. *Indian Journal of Science and Technology*, 9.

Schauf, T. and Neuburger, R. (2021). *Supplementarische Informationen zum DiDaT Weißbuch*, chapter 3.4 Cloudabhängigkeit von KMU. Nomos.

Schräder, E., Bernijazov, R., Foullois, M., Hillebrand, M., Kaiser, L., and Dumitrescu, R. (2022). Examples of ai-based assistance systems in context of model-based systems engineering. In *2022 IEEE International Symposium on Systems Engineering (ISSE)*.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., and Dennison, D. (2015). Hidden technical debt in machine learning systems. *NIPS*.

Seldon Technologies Ltd. (2018). Seldon Core. https://www.seldon.io/solutions/open-source-projects/core (accessed 9 December 2022).

SoundCloud (2012). Prometheus. https://prometheus.io (accessed 28 November 2022).

Symeonidis, G., Nerantzis, E., Kazakis, A., and Papakostas, G. A. (2022). Mlops - definitions, tools and challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*.

Zhou, Y., Yu, Y., and Ding, B. (2020). Towards mlops: A case study of ml pipeline platform. In *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*. IEEE.