In pursuit of reproducible release engineering pipelines for extensible machine learning based systems

Konrad Ponichtera, Krzysztof Baran, Călin Georgescu
Dan Plămădeală, Casper Henkes

ABSTRACT

Machine learning based projects become more common as time passes by. However, they also pose unique challenges stemming from their experimental nature, which are not as prominent in the traditional expert systems that can be verified for correctness in a deterministic way. In this paper, we present the machine learning based system, developed with modern release engineering practices in mind. We also extended it by implementing continuous learning and accommodating new data from its end users to help make the model perform better. We emphasize the long-term maintainability of the project by providing observability features that allow for monitoring the model's performance. Last but not least, we used containerization to tackle the environment reproducibility challenges posed by the technological stack of Python, which is the most commonly used language in the machine learning industry.

1 INTRODUCTION

With the growing performance of modern computers in the last two decades, the mathematical principles behind machine learning could be finally put into practice. However, the machine learning systems' experimental and often "black box" nature creates challenges that are difficult to predict in advance, like the accuracy of the returned results or the data drift. Moreover, the technological stack of the data science utilities is often curated towards experimentation and data exploration rather than the production use in the final product. This means that many implementations of the machine learning pipelines, for instance, in the Jupyter notebook [2], cannot be used directly in practice. For this, it is necessary to design a system from scratch and carefully cherry-pick the solutions from the notebook. Since the vast majority of machine learning projects and experiments are done in Python, creating a system in it brings additional challenges, stemming from the fact that its ecosystem was not created with the self-contained executables in mind. Unlike Java with its JAR archives or Go and Rust with statically-compiled native binaries, Python programs often rely on the dependencies and libraries being installed as part of the system they are running in.

Fortunately, many of the modern software engineering practices can be applied to machine learning projects. This includes automation of the different parts of the release pipeline through continuous integration, delivery, and deployment. CI/CD can validate the changes, drifts and inconsistencies in the learning procedures, as well as lint the code to ensure good coding practices from the machine learning perspective. Thanks to the rise of Docker [11] and rapid adoption of containerization, it became possible to provide lightweight, deterministic environments for Python applications, each with its interpreter and dependencies. Since the cloud infrastructure providers expose APIs, which allow their clients to create the resources on their premises programmatically, it opens new possibilities for declaring the infrastructure itself as a code. This makes it possible to utilize good coding practices and use CI/CD not only to develop and deploy the application but also to provision the environments.

One of the crucial elements of modern software engineering is providing the deployed system with various observability capabilities, like logging, metrics or tracing. The runtime metrics become particularly useful in the case of systems equipped with machine learning capabilities, allowing data scientists and engineers to assess the performance of the models used in production.

In this paper, we will describe the system we created from the proof of concept machine learning pipeline, delivered as a Jupyter notebook. Its purpose is to tag the titles of StackOverflow issues with related technologies. The developed system embeds this pipeline into a microservice-based solution, which can be freely scaled up and down as the amount of users changes. Moreover, we have extended the pipeline with continuous learning capability, allowing the users to provide feedback on the tags predicted by the system and use this feedback to facilitate model learning further. Thanks to the implemented observability measures like the metrics, it is possible to monitor the effectiveness of continuous learning and detect anomalies like the data drifts. What's more, we explored the ways of using Docker containers to benefit from their determinism not only in the deployment stage but also during development and running CI/CD jobs.

First, we discuss the solution's architecture and how the modern release engineering practices have been implemented there. Then we focus on the implications of the decisions that we made during development and if they managed to bring value to the maintainability of the machine learning part of the system. Lastly, we discuss the trade-offs made and how the system could be extended in the future - either to tackle these trade-offs or to make better use of the design decisions that were implemented.

2 RELATED WORKS

The twelve-factor app paradigm proposed in [25] is a methodology developed for the modern *software-as-a-service* paradigm that seeks to improve the quality of applications facing contemporary challenges. To do this, best practices are recommended in twelve key areas of the development process, ranging from codebase version control to logging and concurrency management. One of the factors mentioned is the environment parity, which aims to increase development agility by making the development environment as similar to the deployed one as possible.

Another active area of research is the verification of machine learning applications. The authors of [23] propose a generalizable and adaptable way of assessing the condition of the quality assurance process that a machine learning system is subjected to. They recommend a point-based scoring system that developers

can use as a guideline for determining the quality of their testing practices. This score nudges developers to take machine learning testing seriously and explore different methods to improve testing.

Furthermore, much research time has been dedicated to improving the reproducibility of machine learning applications. The authors of [22] identify ten common shortcomings of machine learning experiment reproducibility and propose a set of 8 questions that should be answered in publications to remedy these issues. Additionally, Pineau et al. [19] created a checklist that should be looked into before submitting machine learning papers to ensure the results are reproducible.

Also, in the academic fields of machine learning and deep learning, it is widespread for the algorithms described in the research papers not to provide sufficient information about the setup, configuration, dependency versioning, and so on. This leads to non-determinism, as described in [18], which can be particularly damaging to real-world applications because it is not feasible for end-users like engineers and scientists to fine-tune the large number of parameters present in the models they use.

3 SYSTEM DESIGN

The application's initial state was a proof of concept, implemented as a Jupyter Notebook. The notebook contains logic, which describes the whole machine learning pipeline of the solution: preprocessing the data, fitting the logistic regression classifier, and testing its performance. However, it was by no means a solution that could be directly used in the production system. In order to be used like that, it had to be rewritten into a standalone, fully-fledged system that can be deployed to a server machine.

3.1 Implementation

The Jupyter notebook has been repurposed into a micro-service system consisting of two applications: the learning and inference service. The learning service is responsible for training the model, while the inference service uses it to perform predictions. Additionally, a small web application integrates with the inference service's API and allows users to query the service for predictions and provide feedback on the predictions.

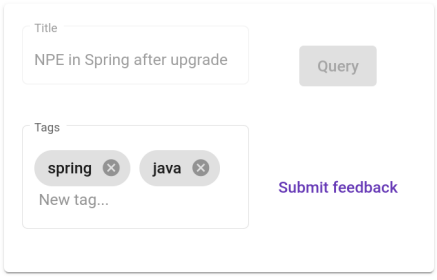

Figure 1: Web interface, allowing users to provide the StackOverflow title, query the inference service for the labels, and provide feedback on the returned labels.

3.2 Deployment

All the applications have been packaged, along with their dependencies, into Docker container images. The images contain the minimum environment required by each of the services to run. Not only does it allow to run each service independently from the programs and libraries on the host operating system, but it also allows making use of the container orchestration software. We chose Kubernetes [6] since it is the most prominent and widely-used orchestrator in the market. Moreover, many public cloud service providers have Kubernetes distributions available in their offer, removing the necessity to administer the cluster on-premises and allowing for integration with other cloud resources. For this reason, we decided to deploy the system to Google Cloud [12], utilizing its Google Kubernetes Engine (GKE) [16].

Managing multiple Kubernetes resources is a potentially error-prone procedure, which becomes even more problematic when multiple environments are meant to be deployed. For this reason, we decided to use Helm [5], which bundles Kubernetes resources in archives called Charts and uses the Go templating language to fill in the values that change between different deployments. Helm is often called a "package manager for Kubernetes" due to its workflow, which resembles how Linux package managers work, allowing for deploying the whole system with one command and optionally providing values specific to the environment.

Cloud infrastructure providers often expose an API, which allows their clients to create the infrastructure in an automated way. Choosing Google Cloud as a provider allowed us to utilize the concept of "infrastructure as code" and design the system's environment with Terraform [10]. It defines infrastructure components as "resources" in the HashiCorp Configuration Language (HCL) [9], where each resource accepts input values and produces outputs upon creation. This allows representing dependencies between different components in a natural way, where outputs of one resource can be used as inputs of another one, resulting in a deterministic deployment behavior. In addition to creating the resources, Terraform performs reconciliation with the actual state of the infrastructure, which allows it to be aware of all the changes that could have been made outside of the HCL code. We used Terraform to provision the infrastructure in Google Cloud and to manage the resources in the Kubernetes cluster created as a part of the infrastructure. This includes cluster components like our system's Helm Chart and the Prometheus [7] instance, which is used to collect the application runtime metrics and send them to the Google Cloud's operation suite [15].

3.3 Pipeline extension

In addition to implementing a solution from the Jupyter notebook as a production-ready system, we extended it with continuous learning functionality. Instead of relying on the model, trained locally by the developer or in the CI/CD pipeline, the users can send feedback on the tags predicted for them by the system, for example, by changing, adding, or removing them. The feedback is then put in the Pub/Sub [13] message queue and collected by the learning service, which caches it locally into the training CSV file. That file is saved in the Kubernetes' persistent volume, which

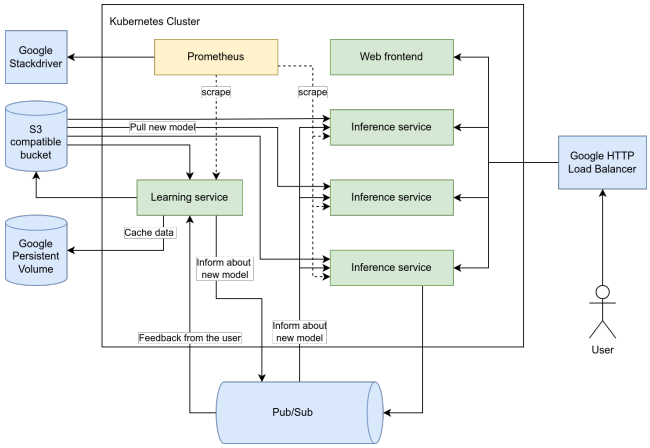

Figure 2: Infrastructure diagram of the system, deployed on Google Cloud.

allows it to survive container recreation during system updates or redeployment.

After a certain amount of feedback training samples is collected, the learning service fits the model on the new training data. The newly trained model is then sent to Google Cloud Storage (GCS) [14] bucket, an S3-compatible object storage service. The models in the bucket are versioned, so it is always possible to restore the previous one. Moreover, the learning service sends a message to the Pub/Sub queue topic, to which the inference services are subscribed, and informs them about the new model's availability. Upon receiving that message, each instance of the inference service downloads the new model from the GCS bucket and uses it for future predictions.

3.4 Metrics

The learning and inference services expose metrics in the Prometheus format. In addition to standard metrics containing information about Python's runtime, custom ones about the trained model's performance were added. This includes accuracy, F1 score, average precision, area under the receiver operating characteristic curve (ROC), and timestamp of the latest model update. The Prometheus instance, deployed to the Kubernetes cluster by Terraform, scrapes all the service instances periodically and forwards collected metrics to the Google Cloud's operation suite. They can then be queried from the Monitoring panel in the Google Cloud Console and used to create dashboards.

3.5 Continuous integration, delivery and deployment

Since the system's source code is hosted on GitHub, the natural choice for the continuous integration and continuous delivery (CI/CD) pipeline was GitHub Actions. All the steps, from validating the code to deploying it to the infrastructure, are executed in the workflows of the GitHub Actions.

Continuous integration is realized by executing workflows that validate the quality of the code. This includes checking the code formatting, executing Python's linting utilities like pylint [8] and mllint [24], as well as executing tests. In particular, learning service tests are responsible for ensuring that the preprocessing pipeline functions as expected, and so does the learning pipeline. The tests are also used to ensure no anomalies in the testing data set, which is stored in the application resources and used to assess the model performance after each training batch. The Helm Chart is also linted, as well as the Terraform project. Testing, linting, and formatting validation are executed for every commit in the repository to provide developers with early feedback about their changes. Moreover, the Docker images of all the system components are built for every pull request to ensure no changes like incompatible Python dependencies, which would block the pipeline from proceeding to the next steps.

Continuous delivery is triggered by creating a semver-formatted tag [3] in the repository's main branch. The workflow builds the Docker images of the services and pushes them to the GitHub Container Registry so that they can be pulled later by the Kubernetes cluster. Each image is tagged in accordance with the Git tag that triggered the build. After the images are built and pushed, the Helm Chart is packaged and uploaded as a release in the repository by the chart releaser [1] utility. It also uses GitHub Pages to generate and publish a Helm repository based on the created releases. The repository can be used by the administrators and Terraform to deploy the system.

Continuous deployment is done by using GitHub Actions to apply Terraform execution plans. Terraform is responsible for provisioning the infrastructure in Google Cloud and creating all the required resources in the Kubernetes cluster. The initial provisioning of the Google Cloud project has to be done manually. It includes executing scripts to create the project, Terraform service account, GCS bucket to store Terraform's state file, and generating service account key for GitHub Actions. However, once that is done, the pipeline will automatically apply all the subsequent changes to the infrastructure. This includes managing the system's Helm Chart release - even if there were no changes to the Google Cloud resources like the cluster or networking, the Terraform plan will still run and upgrade the Helm Chart release in the cluster.

Configuration of the unique environments is stored in the repository as Terraform variable files for each environment. These variables do not include sensitive information like passwords or private keys; instead, these are generated by Terraform and passed to the Helm during the Chart release upgrade. There are two environments - test and production. The first one is deployed automatically whenever a new release is created. The production deployment is suspended until the GitHub repository administrator approves or rejects it manually.

3.6 The concept of builder image

Although containerization provides a large degree of reproducibility when used for deploying applications, the necessity to rebuild the images every time the code change is made would make the local development cumbersome. This is particularly crippling where interpreted languages like Python are used, where a considerable benefit is the ability to change a file and simply restart the developed process quickly. However, working on the Python-based system directly on the local machine can lead to a series of compatibility issues, like the Python interpreter versions mismatch or

having to manage the virtual environments with dependencies of the individual applications.

In order to simulate that incompatibility, the learning service uses Python 3.8, while the inference service requires Python 3.10. This discrepancy automatically implies that the versions of the dependencies are different as well since not all Python 3.10 libraries are compatible with Python 3.8 and vice versa.

Our solution to that problem is a Docker builder image. It is created from the first stage in the multi-stage build of the application's production container image. The builder stage contains all the utilities and libraries necessary to build the application. After the build is finished, the final artifact can be copied to the lightweight, final stage, which results in the final image. Such an approach is commonly used when containerizing the software.

The uniqueness of our approach comes from the fact that during development, only the builder stage is used to create the image. This image can then be used to create a container with an environment explicitly curated for developing a particular application. The project's directory from the host machine is mounted directly into the container, allowing the containerized interpreter to directly execute the source code, modified by the developer in their IDE. By making the containerized process use host's networking namespace, the ports exposed by the application can be accessed directly, without Docker's default network isolation. Although, by default, all the processes in the container are executed as root, the process inside the builder container is run with the same user and group identifiers as the host machine user. This prevents permission problems on UNIX systems from occurring when the containerized process creates a root-owned file in the directory, mounted from the host machine. This makes them cumbersome to delete from the host and can break the version control systems like Git, which are not able to modify these files.

Python applications do not produce executables in the same sense as other technologies like C/C++, Go, or Java. Instead, their builder image is often used as a base for the final image. However, splitting the stages and having access to the builder image still brings certain benefits to the development process. Not only does it allow the use of different Python versions for each application, but it is also not necessary to create a virtual environment directory on the developer's local machine either - both the interpreter and the libraries are available from within the builder container.

This approach is also used for the frontend web app, where its builder image bundles pre-defined versions of Node.js and Angular CLI.

3.7 Build task management

Although the GNU Make [20] is often used in Python projects as a tool for grouping all the development operations, we chose Task [4] instead. To define the tasks, it uses YAML-formatted files called Taskfiles, which conceptually build on the idea of Makefile. However, the workflow is not opinionated by the C/C++ build stack and is much easier to use with other technologies. In particular, Taskfiles are much more flexible in terms of specifying the conditions on when the particular tasks are up to date, which has proven to be particularly useful when automating Terraform's initial project provisioning - for example, to skip new Google Cloud's project creation if it already exists, which requires invoking Google Cloud SDK command. Task's ability to create a hierarchy of Taskfiles allows for splitting the tasks per module, making them much easier to call and maintain than one large Makefile.

Most importantly, Task can be invoked on the host machine to create the builder container with all the necessary parameters, like host networking, specific user and group identifiers, and volume mounts. Since all the builder images have Task installed, it is possible to chain Task executions so that operations like starting the developed application or running the linter are always executed in the same environment.

4 IMPLICATIONS

4.1 Production-ready machine learning system

The changes to the initial Jupyter notebook transformed the proof of concept into a production-ready machine learning system, which uses modern software engineering practices. This includes automated validation, build, and deployment pipelines, all the way through providing observability features like a centralized log and metrics browsing.

The containerization of the solution brought multiple benefits as well. It allowed for creating a scalable and reproducible environment on the Kubernetes cluster, which uses the orchestrator's capabilities of monitoring the application uptime and readiness, as well as leverages its internal load balancing. As a result, the system became much more resilient to failures like the application or machine crashes since the failed applications can be easily rescheduled on the working machines.

Moreover, extending the learning pipeline with continuous learning capability allows the system to potentially accommodate the new data and use it to improve the model. Thanks to the observability features like the logs, it is possible to look into the ongoing learning process and compare it to how the previous learning worked. More importantly, exposing the metrics by the components of the system allows monitoring the learning results like accuracy and other statistics, giving a glimpse of how the currently used model behaves. This is important for the production-grade system, where business decisions are often made based on the insights collected from the working system.

4.2 Environment replicability

One of the rules of the twelve-factor app paradigm [25] is the environment parity, which means keeping the environments as similar as possible. Thanks to the reproducible nature of the Docker containers, ensuring the same application's behavior during the testing and production stages is simple and boils down to using the same Docker image. Moreover, the containers can be executed not only on the fully-fledged Kubernetes cluster but also locally on the developer's machine using Docker Compose. This allows to quickly validate the behavior of the production build of the system locally, without the necessity of configuring the local cluster and its service exposal, which would be necessary for the Kubernetes.

However, this assumes that these environments use the same infrastructure which Google Cloud provides in our case. Developers cannot access Google Cloud resources like Pub/Sub message queue or GCS buckets unless they provision them manually or with

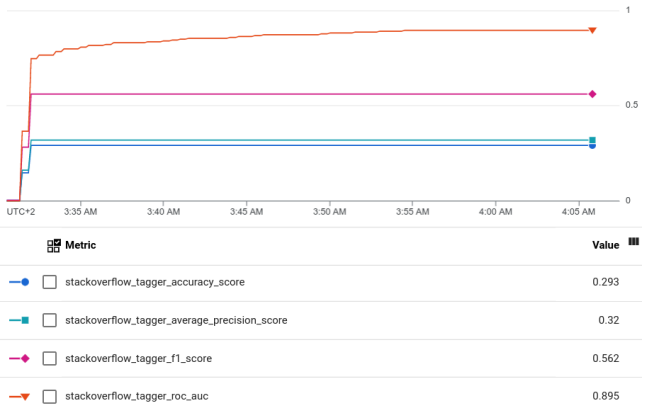

Figure 3: Model performance metrics in the Google Cloud Monitoring dashboard after feeding the system with 30000 new feedback entries.

Terraform. To solve that problem, we used containerized services, which run on the developer's machines and use the same protocols as their cloud counterparts. As a local replacement for GCS, we chose MinIO [17], an open-source implementation of S3 that can be easily run as a container. Google provides the containerized emulator of the Pub/Sub message queue, a non-scalable, feature-restricted version of the cloud Pub/Sub. It is compatible with the cloud version and allows to test the core functionality of the message queue, save for things like IAM authentication, for which the full Google Cloud access is necessary.

Using Docker containers to satisfy developed applications' requirements for external services simplifies the development process since it does not require every developer to provision them on the infrastructure provider's side. This also means that there are no additional costs involved during development.

4.3 Reproducible CI/CD operations

GitHub Actions, used in our project as a CI/CD pipeline, execute their workflows in the scope of ephemeral virtual machine workers spawned for each job. The Linux workers are based on Ubuntu LTS versions and have a plethora of preinstalled tools like Git or Docker. Moreover, there exist multiple actions which prepare the worker for running tasks like building, linting or testing by installing additional utilities and toolchains like Java and Python. However, this requires managing dependency versioning in multiple places and ensuring that the developers are using the same version of the utilities as it will be used when building the application. Moreover, suppose there are tools which do not have a dedicated GitHub action that installs them in the worker. In this case, it is necessary to write a custom one, which creates additional elements to maintain.

Using the builder container in the CI/CD pipeline can solve these problems by providing exactly the same execution environment for the local development and pipeline jobs. Not only does it solve the problem of managing the versions in multiple places, but it also allows for all tools to behave the same and avoid situations like the linter, reporting different places in the code on the developer's machine than it does in the pull request's pipeline. Since the workers

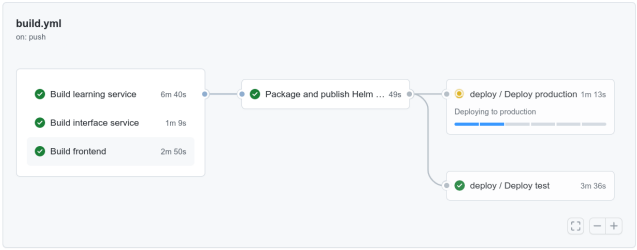

Figure 4: GitHub Actions build workflow. Each depicted job is executed in the builder container, created from the builder image.

already have Docker daemon installed, the only thing that has to be done is to build the builder image and then run the operations like linting or testing inside the spawned container. Moreover, it is possible to store the layers of the builder image in the GitHub Actions cache by utilizing Docker BuildKit's [21] caching, which speeds up the subsequent workflow executions. This has the potential of being even faster than using pre-made actions for installing utilities on the runners, as they don't have to be downloaded from the package repositories on every job run and will merely get fetched from the GitHub cache, alongside the rest of the builder image.

5 DISCUSSION

5.1 Trade-offs

A significant trade-off we made was choosing to use only services that run on the cloud and a local machine or have good emulators for running locally. This choice was made to remove the problem of "this runs on my machine". Everything is containerized and runs the same way: locally, in the CI/CD pipeline, and on the cloud. Also, launching the whole system and its dependencies locally creates an excellent opportunity for reproducible integration and end-to-end tests. Since all the services are in the Docker containers, their ephemeral nature ensures deterministic results of each test suite execution - both on the developer's machine and the CI/CD pipeline. The trade-off is that we can thus also only use tools that support this approach, which cannot always be done with cloud-native proprietary solutions. However, we think that for most use cases, the benefit of everything being consistent across every platform far outweighs the drawbacks of a more limited choice of tooling and a more complex initial setup.

One of the most important things in creating machine learning applications is ensuring that the model created matches the real world. We tried to achieve this by allowing the model to learn from the users, giving it more data to learn from and, thus, higher theoretical accuracy and relevancy in an ideal world. Our world is usually not ideal, and adequately dealing with new data is difficult. For example, a user might submit a tag that did not exist in the system before. In this case, you have an interesting problem: do you add the new tag or not? The system should not blindly accept all the new tags a user submits since the user might, for example, make an error while typing. There can also be tags that might be worded differently but have the same meaning, with capital or non-capital letters being an example. As seen from these small examples, a lot

of care must be taken when automatically creating new training data for the service. Another problem with continuous learning is that the original test data might eventually not be representative anymore of the problem the application wants to solve. This makes it more challenging to decide if a new model should be released or not, as it is hard to have a good indication of how well the model is performing.

5.2 Next steps

For future work, we would like to extend our design with new features that would further help assess model performance: shadow deployment and continuous improvement of test data. Shadow deployments would allow us to provide the newer model to only a fraction of the users and see how it performs in the real world before it is fully released. It could be achieved by developing a model version manager. The manager would be responsible for determining which version of the model from the bucket is to be used by the particular inference service. The monitoring would also need to be slightly altered to include a dashboard for comparing the performance of multiple models. Another idea that fits well is the continuous improvement of test data. This is an extension of the continuous learning idea, allowing to collect not only new training data, but also the test one. However, it might not be a good idea to have the test data updated fully automatically, as the reason behind test data being treated as a trusted knowledge base is often the fact that it was checked by a human expert. For this reason, it could be beneficial to utilize distributed crowdsourcing solutions like Amazon Mechanical Turk, to have a human validate the newly collected test data before using it to determine learning performance.

Another significant extension would be the inclusion of metric alerts. Having metrics that monitor the model's performance is crucial when working with machine learning based systems. However, it might come to the point where the amount of metrics is too large to be efficiently displayed on the dashboards. Configuring alerts to send a notification whenever there are anomalies in the system's normal behavior might significantly simplify the job of site reliability engineers and reduce their response time.

In addition to having Terraform provision the infrastructure, it is possible to use it to remove certain cloud resources that it created. Although it is not something that should be done with production environments, it starts to make sense when used with test and staging ones. In the cloud environment, the machines equipped with the tensor processing units (TPUs) and other AI accelerators tend to be much more expensive than the normal ones. For this reason, it might be a good solution to utilize Terraform's targeted destroy functionality to remove specific resources that generate huge costs - for example, Kubernetes cluster's node pools - without affecting the stateful elements like the buckets, volumes, Pub/Sub queues or Kubernetes cluster control plane. Recreating the node pool later will allow the control plane to reschedule all the applications as they worked before. This process can be automated by using GitHub Actions scheduled workflows - for example, by removing the expensive resources before the weekends or outside of working hours and recreating them later.

The inference services can be easily scaled horizontally to accommodate an arbitrarily large number of users; the learning service's scalability in the current system design is a potential bottleneck. It is a singleton and the only consumer of the user feedback suggestions. Although the asynchronous nature of the message queue means that the user experience is not affected by this bottleneck (e.g., the API response times are the same), it also means that for the system under constant load, the number of new feedbacks in the queue would grow faster than the learning service would be able to consume. The only way to increase its throughput is by assigning more resources to the machine on which the service is deployed (vertical scaling). However, this cannot be done indefinitely, and there are limits to how powerful the single machine can be, whether the system is deployed on-premises or in the cloud. For this reason, it would be beneficial to research the available federated learning options to coordinate the learning between multiple machines.

6 SUMMARY

In the span of a month, we developed a fully functional machine learning based system with production-grade continuous integration, delivery, and deployment pipelines. Not only does it implement the proof of concept machine learning pipeline, delivered as a Jupyter notebook, but it also allows to collect feedback from the users about the delivered predictions. This feedback is then used for continuous learning to train the model further and potentially increase its performance, which can be monitored thanks to exposed metrics. The project also serves as a testing ground for the concept of the builder image. It aims to further streamline the developer experience by providing a replicable runtime environment on all the steps that precede the deployment, including but not limited to linting, testing, and running the developed application. Thanks to Terraform and containerization, an arbitrary amount of environments can be deployed and maintained in a reproducible fashion.

Considering that the initial plan was to build an extensible machine learning system with a strong emphasis on environment replicability and ease of development, it is safe to say that the goal was accomplished. The created project scaffolding serves as a solid fundamental for developing production systems in Python with the widely available machine learning frameworks and helps to avoid problems that plague the ecosystem, like the interpreter version discrepancies.

An important take-home message from the project is that environment observability plays a significant role in modern software engineering, whether applied to a simple web application or a complicated machine learning system. There is always information that the working application can expose as metrics, which might bring valuable insight into the system's behavior and help in making decisions that guide the project's evolution. Moreover, creating a project scaffolding with emphasis on environment replicability is crucial from the perspective of its long-term maintainability - even if it might require extra effort in the initial stages of the development.

REFERENCES

[1] [n.d.]. Chart Releaser. https://github.com/helm/chart-releaser.
[2] [n.d.]. Project Jupyter. https://jupyter.org/.
[3] [n.d.]. Semantic Versioning 2.0.0. https://semver.org/.
[4] [n.d.]. Task. https://taskfile.dev/.
[5] Cloud Native Computing Foundation. [n.d.]. Helm. https://helm.sh/.
[6] Cloud Native Computing Foundation. [n.d.]. Kubernetes. https://kubernetes.io/.
[7] Cloud Native Computing Foundation. [n.d.]. Prometheus. https://prometheus.io/.
[8] Cloud Native Computing Foundation. [n.d.]. Pylint static code analyzer. https://pypi.org/project/pylint/.
[9] HashiCorp. [n.d.]. HashiCorp Configuration Language. https://github.com/hashicorp/hcl.
[10] HashiCorp. [n.d.]. Terraform. https://www.terraform.io/.
[11] Docker Inc. [n.d.]. Docker. https://docker.com/.
[12] Google Inc. [n.d.]. Google Cloud. https://cloud.google.com/.
[13] Google Inc. [n.d.]. Google Cloud Pub/Sub. https://cloud.google.com/pubsub.
[14] Google Inc. [n.d.]. Google Cloud Storage. https://cloud.google.com/storage.
[15] Google Inc. [n.d.]. Google Cloud's operations suite. https://cloud.google.com/products/operations.
[16] Google Inc. [n.d.]. Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine.
[17] MinIO Inc. [n.d.]. MinIO. https://min.io/.
[18] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2018. The impact of nondeterminism on reproducibility in deep reinforcement learning. (2018).
[19] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving reproducibility in machine learning research: a report from the NeurIPS 2019 reproducibility program. *Journal of Machine Learning Research* 22 (2021).
[20] GNU Project. [n.d.]. Make. https://www.gnu.org/software/make/.
[21] The Moby Project. [n.d.]. BuildKit. https://github.com/moby/buildkit.
[22] Sheeba Samuel, Frank Löffler, and Birgitta König-Ries. 2021. Machine Learning Pipelines: Provenance, Reproducibility and FAIR Data Principles. In *Provenance and Annotation of Data and Processes*, Boris Glavic, Vanessa Braganholo, and David Koop (Eds.). Springer International Publishing, Cham, 226–230.
[23] Eric Breck Shanqing Cai Eric Nielsen Michael Salib D. Sculley. 2016. What's your ML Test Score? A rubric for ML production systems. *30th Conference on Neural Information Processing Systems* (2016).
[24] Bart van Oort. [n.d.]. mllint. https://github.com/helm/chart-releaser.
[25] Adam Wiggins. 2017. The Twelve-factor App. https://12factor.net/.