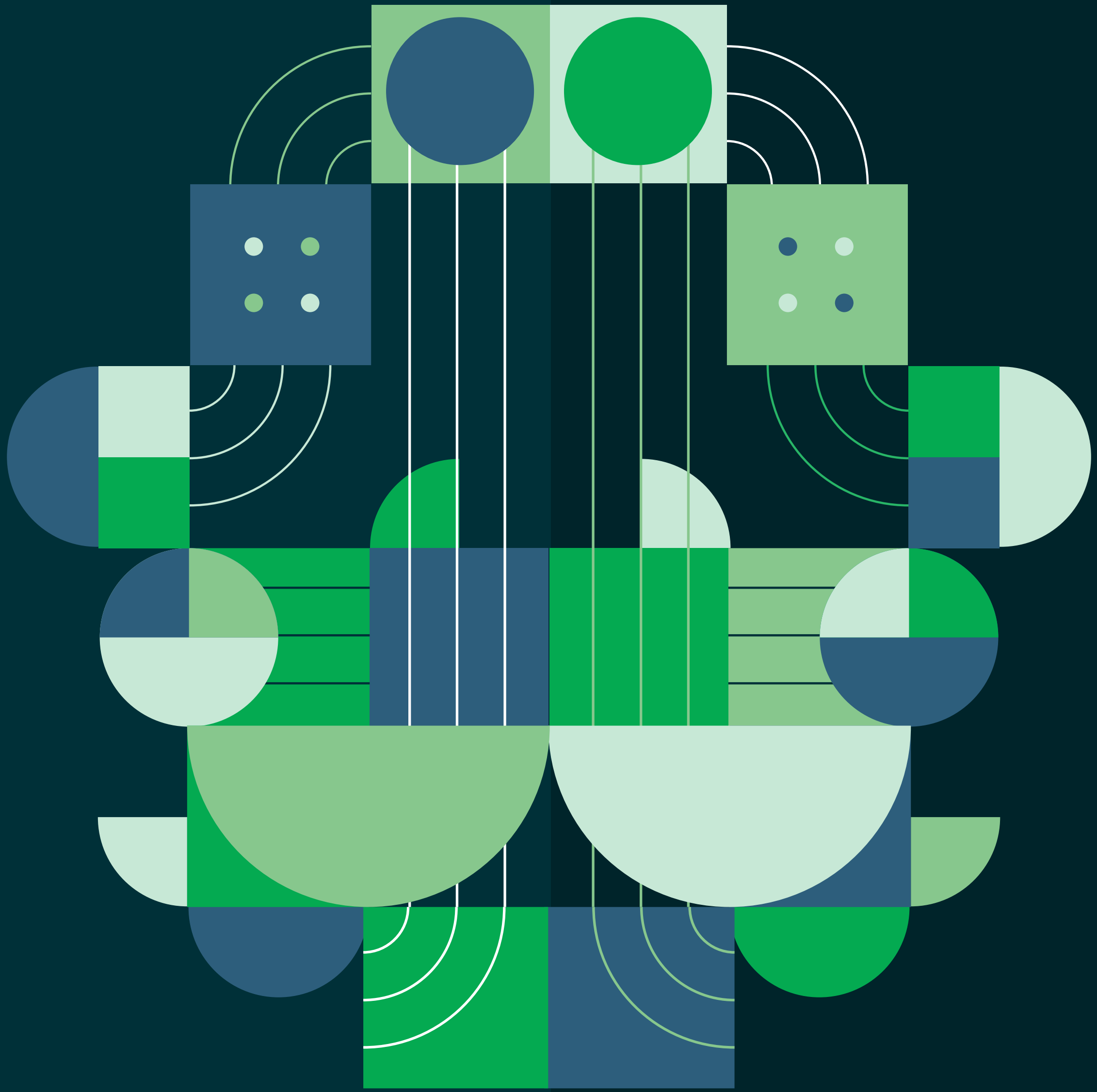
circleci

# Automating and Scaling Machine Learning Workflows with CI/CD



Machine learning (ML) is becoming an essential technology for many organizations, driving innovation and improving operational efficiencies. However, bringing ML models into production environments poses a range of challenges that require careful consideration and strategic problem-solving.

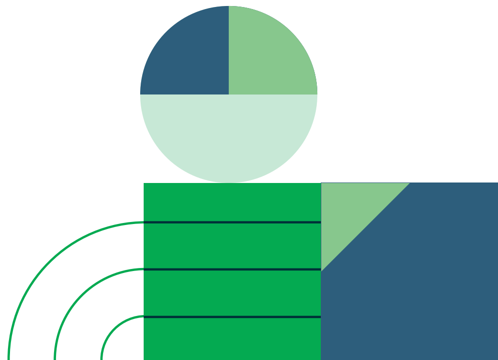
To streamline the process, continuous integration and continuous delivery (CI/CD) offers ML teams the ability to automate various stages of ML project development and deployment. In this guide, we'll explore the benefits of using CI/CD for ML projects and how it can help your organization overcome common ML challenges. Then, we'll walk through the process of setting up your own CI/CD pipeline to build, test, train, deploy, and retrain your models for faster, more efficient ML workflows.

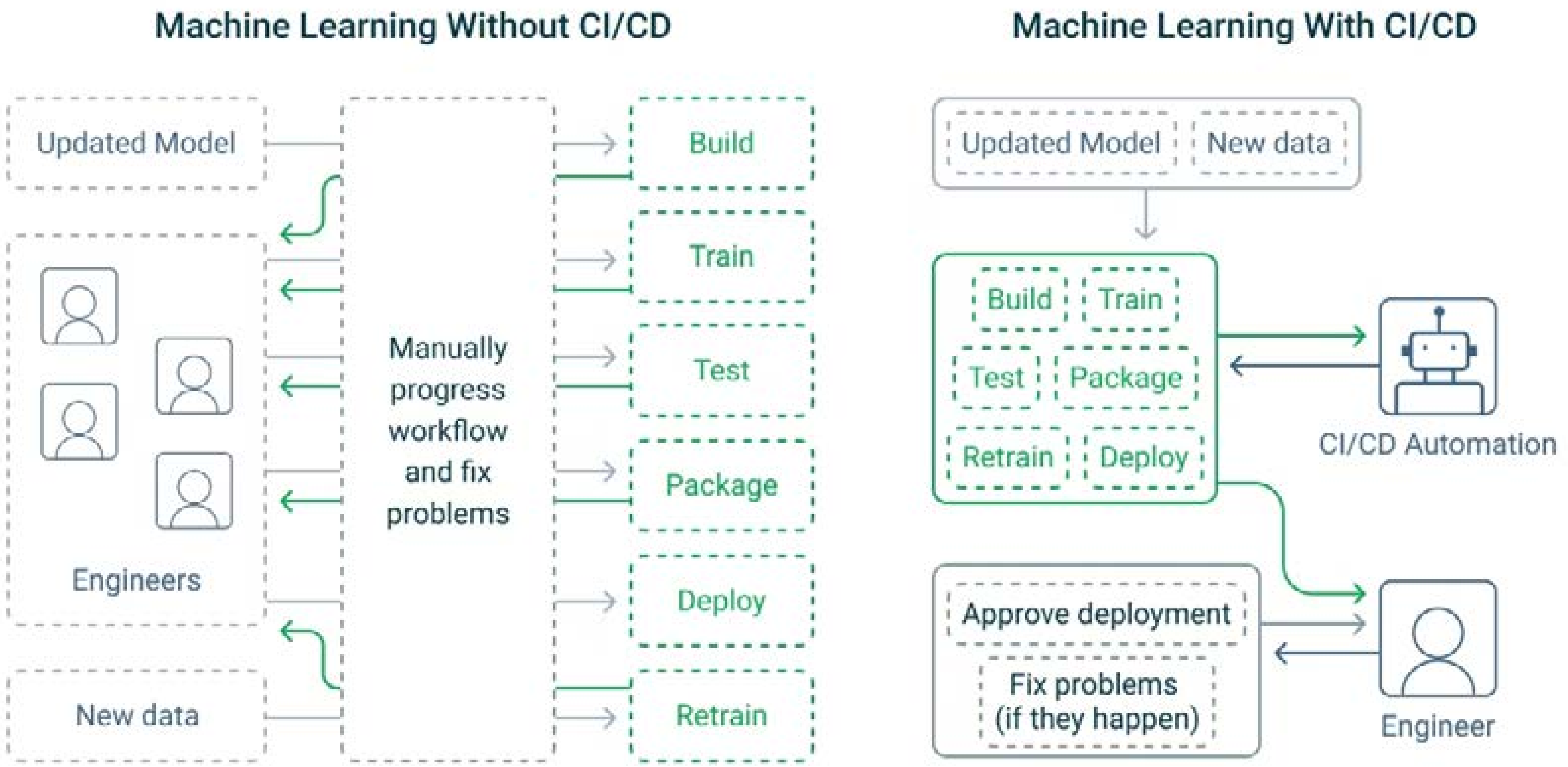
## What is CI/CD, and what can it do for your ML models and workflows?

**CI/CD pipelines** are an essential tool for any software development team, but their importance and utility become even more pronounced in the context of machine learning (ML) projects. A CI/CD pipeline is a framework that enables developers to integrate their code changes more frequently and reliably into the codebase. It automatically executes a series of steps every time a code change occurs, beginning with the integration of the new code, followed by the building and testing of the whole system to ensure there are no breaking changes.

For ML development teams, the use of CI/CD pipelines can significantly streamline their workflows. By automating the testing and validation of ML models, teams can catch and address issues quickly, speeding up the development process and reducing the risk of errors in the final product. Furthermore, CI/CD facilitates consistent and repeatable processes, making it easier to scale ML operations. With each change to the model or the data, the pipeline can retrain the model, validate its performance, and if the performance metrics meet the acceptance criteria, deploy the model to the production environment.

Effectively implementing a CI/CD pipeline paves the way for MLOps (machine learning operations) – a practice for collaboration and communication between data scientists and operations professionals to help manage production ML lifecycle. MLOps seeks to increase automation and improve the quality of production ML while also focusing on business and regulatory requirements. With CI/CD, teams can manage frequent, incremental updates to models in a manageable and predictable way, ultimately leading to MLOps that are quick, efficient, and robust.





Without automation, each step of the ML process must be managed manually - possibly requiring a team to turn around updates quickly. With automation using CI/CD tools and practices, the entire process can be run and monitored, often by just a single expert.

## Solving the top challenges of ML Model Development with CI/CD

ML model development is a complex process, and organizations often face a number of challenges that can affect the success of their projects. CI/CD's ability to automate the different stages of ML model development helps address some of these challenges. Let's take a look at some of the most common ones:

### Scalability and compute resource management

One of the main challenges that ML developers face is the intensive compute requirements for building and training large-scale ML models. Indeed, training large language models (LLMs) like ChatGPT typically consumes billions of input words and **costs millions of dollars in computational resources**.

Because of the scale needed to train and develop these models, analysts have proposed cloud computing to meet the computational demand. However, using GPU or CPU resources from popular cloud services — such as Amazon Web Services (AWS) and Google Cloud Platform (GCP) — for extended training tasks is costly. Moreover, cloud providers' "unlimited" scaling offerings can lead to runaway resource usage and associated costs.

CircleCI's features enable scaling while controlling and monitoring costs. For training models in the cloud, CircleCI offers **several tiers** of GPU resource classes with transparent pricing models. Alternatively, you can use **self-hosted runners** to run automated workflows on your own infrastructure for more flexibility. With this extra versatility, you can configure self-hosted runners to scale automatically or execute jobs concurrently.

## Reproducibility and environment consistency

Another critical aspect of managing ML model deployment is maintaining consistency and reproducibility in the build environment. These properties prevent unexpected errors when restarting CI/CD jobs or migrating from one build platform to another. Consequently, you can avoid costly build errors in ML model development, which often features long-running jobs that are difficult to interrupt.

Fortunately, you can use containerization to isolate deployment jobs from the surrounding environment to ensure consistency. Meanwhile, deployment using infrastructure as code (IaC) helps improve the build system's reproducibility by explicitly defining the environment details and resources required to execute a task. As a result, the build is less dependent on platform-specific settings — you can reproduce and audit it easily.

For these reasons, CircleCI provides tools like **Docker executor** and **container runner** for containerized CI/CD environments, offering a platform that supports YAML-based IaC configuration.

## Testing and validation

Testing is crucial in developing any software project and especially for ML-powered programs. By nature of their complexity and training, ML models tend to feature implementation that is opaque to the user, making it near-impossible to determine a model's correctness by inspection. Therefore, comprehensive testing is essential for proper software functionality.

CircleCI excels at integrating testing into the development process. Support for automated testing makes it easy to ensure code performs as expected before it goes to production. You can customize tests on the CircleCI platform using one of many third-party integrations called **orbs**. You can then monitor them via **SSH debugging** or the **Insights dashboard**.

## Security and compliance

Development teams must **ensure that software is secure and compliant** with consumer protection laws. This is particularly relevant for ML development, which often involves processing large amounts of user data during training. A vulnerability in the data pipeline or failure to sanitize the data could allow attackers to access sensitive user information. Therefore, security is a principal consideration at each stage of ML model development and deployment.

CircleCI provides several CI/CD features to improve the security and compliance of your application. You can control access to the pipeline using a **role-based credential system** with OpenID Connect (**OIDC**) authentication tokens, enabling fine-grained management of user access to each step within the pipeline. Additionally, CircleCI logs important security events and stores them in **audit logs**, which you can review later to understand the system's security better.

## Deployment automation

New versions of ML models are often developed rapidly, especially during periods of heightened interest in AI. This makes it challenging to manage frequent updates to ML systems with several versions in development or production. To ensure a consistent user experience, you need an easy way to push new updates to production and determine which versions are currently in use.

Fortunately, you can deploy code to AWS, GCP, or any other targeted platform **continuously and automatically** via CircleCI orbs. Moreover, these deployments are configurable through IaC to ensure process clarity and reproducibility. Users can add a **manual approval gate** at any point in the deployment pipeline to check that it proceeds successfully.

## Monitoring and performance analysis

After deploying an ML model, you must set up production monitoring and performance analysis software. Due to the size and complexity of modern ML models such as LLMs, even a comprehensive test suite may fail to ensure their validity. The only way to determine that a model is performing as expected is to observe its real-world performance by collecting and aggregating metrics from the production environment.

With CircleCI, it is easy to integrate monitoring into the post-deployment process. **CircleCI orbs offer** options to incorporate monitoring and data analysis tools like Datadog, New Relic, and Splunk into the CI/CD pipeline. You can configure these integrations to capture and analyze metrics on the performance and behavior of production-phase ML models.

## Continuous training

During intense AI investment and expansion periods, new research, datasets, and improved models emerge daily. Therefore, production ML models must adapt to incorporating new features and learning from new data.

CircleCI's support for third-party CI/CD observability platforms means you can add and monitor new features within CircleCI. But as new training data generates continuously, you can periodically feed it to the model using **scheduled pipelines**. This feature enables you to schedule events that trigger further training and deployment pipelines — allowing the production ML model to grow and update continuously.

# Automating your ML workflow with CI/CD

Now that you know how CI/CD can streamline and enhance your ML workflows, you're ready to implement a basic CI/CD pipeline to build, test, train, deploy, monitor, and retrain your ML models in production. You can find all of the code used in this section in our **sample repository**.

Since we're focused on building the CI/CD pipelines and not the ML model itself, we'll keep it simple and use the **example ML workflow provided by TensorFlow**. This code builds a model using **Keras** and **example data from MNIST** that identifies images, tests the model, packages it, and deploys it to **TensorFlow Serving**. Once the model is in production, we'll test that it is functioning properly and set up a scheduled pipeline to automatically retrain the model based on fresh production data.

## Prerequisites and installation

To build the automated ML pipeline for this guide, you will need the following:

### A CIRCLECI ACCOUNT AND PROJECT

- See the CircleCI **quickstart guide** to learn how to get up and running with both.
- You can **fork the example repository** for this tutorial from your own GitHub account and use it as the basis for your CircleCI project.

### A CIRCLECI SELF-HOSTED RUNNER

- This can be a local machine or set up as part of an **auto-scaling deployment** for larger workloads.
- The code in the example repository for this tutorial has been tested on Ubuntu 22.04.
- You can also use CircleCI's managed cloud compute resources (including **GPUs**).

### A SERVER WITH SSH ACCESS FOR STORING YOUR TRAINED AND PACKAGED MODELS

- Your models will be uploaded here for storage and, later, publishing to TensorFlow Serving.
- Your runner should be able to reach this machine on the network.

The ML scripts in this example are all written in Python.

On your runner, you will also need Python, pip, and Python venv. On Ubuntu, run the following terminal command to install these:

```
sudo apt install python3 python3-pip python3-venv
```

## Breaking down the ML workflow

The key to automating your ML workflow is to break it down into steps. This allows you to call each step in sequence, or run them in parallel if they do not rely on each other, and monitor the results. If something fails, you can automatically roll back or retrain and notify the responsible team member so that they can respond.

The TensorFlow example that this tutorial is based on was originally written as a single Python script. It has been broken down into the separate scripts below for automation.

### STEP 0: PREPARE YOUR DATA

This is a vital step that should be included in every ML workflow: you must understand your data before you work on it and make sure that it is clear of anomalies before it is used.

An ML model is not effective unless you can verify that it is actually working. If you do not have some cleaned data that you already understand to test it against, you have no way of knowing how accurate it is. Additionally, make sure that you have a source of reliable data to use when training or retraining your models.

### STEP 1: BUILD

Building an ML model is a multi-step process that involves collecting, validating, and understanding your data and then building a program that can analyze and create insights from it.

In our example, the build phase imports and prepares some demo data, ready to train an existing **Keras sequential model** in the next step. In a real-world scenario, you'd supply your own data.

The Python code for this step is in **ml/1_build.py**.

### STEP 2: TRAIN

In this step, carefully prepared, highly accurate data with known outcomes is fed into the model so that it can start learning. This uses the training data from the build phase.

It is best practice to be as verbose as possible in your scripts. Print out as much useful information as you can, as anything that is output to the console will be visible in the logs in CircleCI's web interface, allowing for easy monitoring and debugging.

The Python code for this step is in **ml/2_train.py**.

### STEP 3: TEST

As the training data used in the example ML workflow is already well understood, we can tell if the trained model is accurate by comparing its output with the already known outcomes.

In our scripts, we do this by comparing the testing data created in 1_build.py. If the accuracy is inadequate, an exception is thrown that halts the CI/CD pipeline and alerts the owner.

The Python code for this step is in **ml/3_test.py**.

### STEP 4: PACKAGE

The packaging step prepares the trained model for use in a separate environment — exporting it in a standard format and making it portable so that it can be deployed for use elsewhere. It then uploads it to a package staging location for future use.

This example uploads the files to a remote server using SSH. If you are running your ML pipelines entirely in the cloud and do not want to grant them access to your internal network assets, you can use the AWS S3 orb for this purpose to store your ML artifacts where both CircleCI's cloud compute resources and your own infrastructure can access them.

The Python code for this step is in **ml/4_package.py**.

### STEP 5: DEPLOY

Once a deployment environment has been set up, packaged models can be deployed and used. This stage involves deploying your trained and packaged model to your production ML environment.

In the example repository, the packaged model is uploaded to the directory TensorFlow Serving loads its models from.

The Python code for this step is in **ml/5_deploy.py**.

## STEP 6: TEST DEPLOYED MODEL

Ensuring a successful deployment is important to prevent downtime, so this example makes a quick HTTP POST request to TensorFlow Serving to ensure that it receives a response.

If the request is unsuccessful, the resulting error will be thrown by the Python script.

The Python code for this step is in **ml/7_test_deployed_model.py**.

## STEP 7: RETRAIN

ML is not a "one and done" task. Whether you are analyzing customer data or user behavior or modeling for scientific purposes, when new, high-quality data arrives, you will want to retrain your existing models so that you are not limited to just your initial data set.

Once your new data has been ingested and **validated**, you need to retest retrained models using known data to ensure they remain accurate.

While our example won't be able to load any fresh data, we can still provide a file and pipeline for retraining and retesting.

The Python code for retraining and testing the retrained data is located in the file **6_retrain.py**.

Note that in this script, the testing step is designed to fail! This is so that you can see what a failed job looks like when this script is added to a job in CircleCI.

## Implementing your build, test, train pipeline

Once you've broken down your ML process and created the scripts to run your workflow (or copied the ones in this example), you can publish it to **your preferred version control system** and then **import it as a CircleCI project**.

### The CircleCI configuration file

The CircleCI configuration file is located at the path **.circleci/config.yml**. It defines the **commands**, **jobs**, **steps**, and **workflows** that make up the automated ML system. Below is an explanation of the features demonstrated in the example CircleCI configuration, which contains a full working automated ML system that you can test and build on.

In the configuration file, the `check-python` command demonstrates how to run a terminal command in the CircleCI execution environment. If this command fails, the job and workflow that call it will also fail, sending the error to CircleCI and notifying the owner:

```
commands:
    check-python:
        steps:
        - run:
            command: python3 --version
            name: Check Python version
```

Commands can be reused multiple times within jobs and can also call their own one-off steps. Below, the `install-build` job prepares the environment for running the other ML scripts. It checks out the project code, checks that Python is installed by using the previously defined check-python command, and then runs the `create-env` command to create the required configuration files. From there, it runs its own set of commands to install Python dependencies and then runs the first build step in the ML pipeline.

```
jobs:
install-build:
    # For running on CircleCI’s self-hosted runners - details taken from environment variables
    machine: true
    resource_class: RUNNER_NAMESPACE/RUNNER_RESOURCE_CLASS # Update this to reflect your self-hosted runner resource class details
    steps:
    - checkout # Check out the code in the project directory
    - check-python # Invoke command “check-python”
    - create-env
    - run:
        command: sh ./tools/install.sh
        name: Run script to install dependencies
    - run:
        command: python3 ./ml/1_build.py
        name: Build the model
    - persist_to_workspace:
        # Workspaces let you persist data between jobs - saving time on re-downloading or recreating assets https://circleci.com/
docs/workspaces/
        # Must be an absolute path or relative path from working_directory. This is a directory on the container that is
        # taken to be the root directory of the workspace.
        root: .
        # Must be relative path from root
        paths:
            - venv
            - ml
            - .env
            - tools
```

Workflows are made up of jobs that can run sequentially or concurrently. The `build-deploy` workflow runs the `install-build`, `train`, `test`, and `package` jobs and demonstrates how to use a **branch filter** to run the workflow only when commits are made to the main branch. It also shows how the `requires` option can be used to ensure that jobs execute in order and how multiple job names can be required by a subsequent job, allowing for them to be **executed concurrently**.

You can either create the file yourself at the path `.circleci/config.yml` or **create one in the web console** when you import your project into CircleCI (with the added advantages of linting and schema validation). If you are editing your CircleCI configuration locally, it's advised to use the **CircleCI command line tools** to **validate your configuration** before you commit your changes. VS Code users can also validate their config directly in their IDE using the **CircleCI VS Code extension**.

Take a look at the full working example CircleCI configuration, including all of the required commands, jobs, and workflows.

```
workflows:
    # This workflow does a full build from scratch and deploys the model
    build-deploy:
        jobs:
        - install-build:
            filters:
                branches:
                only:
                    - main # Only run the job when the main branch is updated
        - train:
            requires: # Only run the job when the preceding step in the ML process has been
completed so that they are run sequentially
                - install-build
        # To demonstrate how to run two tests concurrently, we'll run the same test twice under
different names - if either required test fails, the next job that requires them (in this case,
package) will not run - https://circleci.com/docs/workflows/#concurrent-job-execution
        - test:
            name: test-1
            requires:
                - train
        - test:
            name: test-2
            requires:
                - train
        - package:
            requires:
                - test-1
                - test-2
```

## Creating a CircleCI self-hosted runner

The CircleCI pipeline example provided with this tutorial executes everything on a **CircleCI self-hosted runner**. This is often preferred in production, as it means that your privileged data stays on your network. You will need to **configure your self-hosted runner** and provide the details in the `.circleci/config.yaml` configuration file.

After you've configured your self-hosted runner, you must set the correct `RUNNER_NAMESPACE` and `RUNNER_RESOURCE_CLASS` in all locations in the `.circleci/config.yml` file.

```
machine: true
resource_class: RUNNER_NAMESPACE/RUNNER_RESOURCE_CLASS # Update this to reflect your self-hosted runner resource class details
```

You can also define different execution environments for different jobs, which is especially useful when using CircleCI's cloud GPUs for compute-heavy jobs. In this tutorial, we use a single environment to keep things simple.

## Configuring environment variables in CircleCI

You'll notice that there are variables used in the configuration file (prefixed with a $ symbol). You will need to set the following **environment variables** in CircleCI, which will be used in these locations to generate the **.env** file Python uses to obtain your secrets on the runner when the pipeline is executed:

```
DEPLOY_SERVER_HOSTNAME
DEPLOY_SERVER_USERNAME
DEPLOY_SERVER_PASSWORD
DEPLOY_SERVER_PATH
```

Secrets like credentials and API keys should **never** be committed to source control. Environment variables are injected when a CircleCI pipeline is run so that you can create configuration files on the fly and avoid committing secrets.

Note that to keep things simple for this example, we're using SSH password authentication. In production, you should use certificate authentication and restrict users so that they can only access the resources they require. For even better security, consider storing your secrets in a **centralized vault** and retrieving them when they are required.

In the example configuration, the `.env` file is created using the following CircleCI command:

```
commands:
  create-env:
      steps:
      - run:
          # Environment variables must be configured in a
CircleCI project or context
          command: |
              cat \<<- EOF > .env
              DEPLOY_SERVER_HOSTNAME=$DEPLOY_SERVER_HOSTNAME
              DEPLOY_SERVER_USERNAME=$DEPLOY_SERVER_USERNAME
              DEPLOY_SERVER_PASSWORD=$DEPLOY_SERVER_PASSWORD
              DEPLOY_SERVER_PATH=$DEPLOY_SERVER_PATH
              EOF
          name: Create .env file containing secrets
```

See **this example** of what your generated `.env` file should look like.

## Persisting data between jobs

Jobs can be executed in different environments. As such, data does not persist between them by default. CircleCI supports **data persistence between jobs** using **workspaces**. In the jobs defined in the example configuration, data at the paths `venv`, `ml`, `.env`, and `tools` is persisted to a workspace when it is modified:

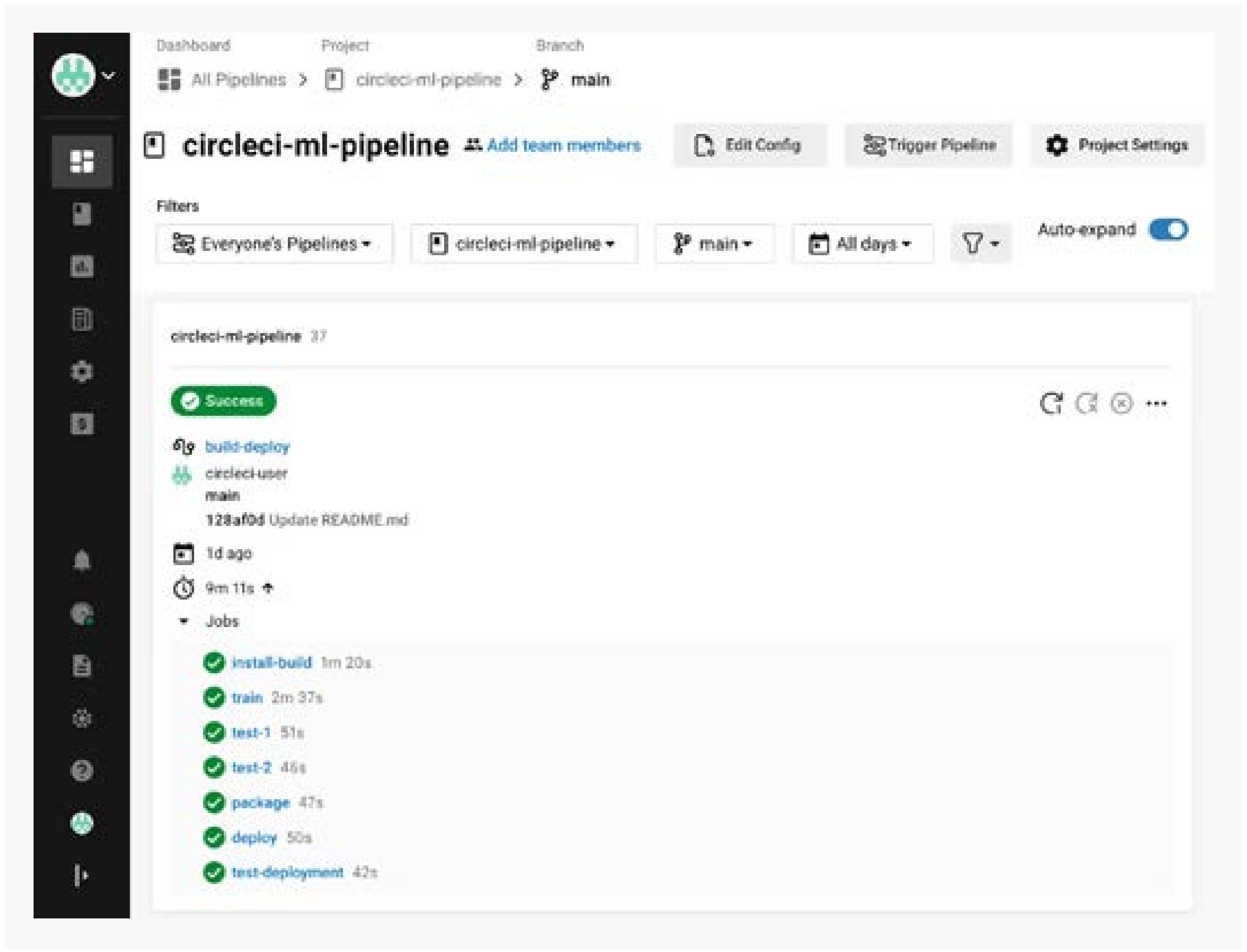
```
  - persist_to_workspace:
    # Workspaces let you persist data between jobs - saving
time on re-downloading or recreating assets https://
circleci.com/docs/workspaces/
    # Must be an absolute path or relative path from
working_directory. This is a directory on the container that
is
    # taken to be the root directory of the workspace.
    root: .
    # Must be relative path from root
    paths:
        - venv
        - ml
        - .env
        - tools
```

That data is then reloaded when needed by attaching the job to the existing workspace:

```
- attach_workspace:
    # Must be absolute path or relative path from working_
directory
    at: .
```

## Confirming your CircleCI workflow has run successfully

When code is committed to a Git branch or a scheduled pipeline is triggered, CircleCI reads the configuration file and determines whether any workflows should run.



circleci-ml-pipeline 37

Success

build-deploy
circleci-user
main
128af0d Update README.md
1d ago
9m 11s
Jobs

- install-build 1m 20s
- train 2m 37s
- test-1 51s
- test-2 46s
- package 47s
- deploy 50s
- test-deployment 42s

As a CircleCI workflow runs, all console output is shown in the CircleCI web console. Each step, job, and workflow will report its status and notify the owner on failure. If an approval is required, the job will pause until approval is given, and jobs can be rerun from the point of failure once a problem is corrected. We'll cover more on monitoring your CI/CD pipeline later in this guide.

# Adding deployment and retraining to your ML workflow

With the above steps, you have a pipeline that will build, test, and train your machine learning models with any change to the underlying code or on any schedule you specify. Next, you will add deployment steps to this workflow and create a separate retraining workflow using the example ML workflow in the **example repository**.

In the example repository, the **4_package.py** script uploads the trained and packaged model to a server via SSH. Below, we will deploy the packaged model to a **TensorFlow Serving** server. To keep things simple, we'll assume that this server is **running in a Docker container** on the same host that we uploaded the packaged models to.

## Setting up Tensorflow Serving

A **Bash script** is supplied for spinning up a Docker container running TensorFlow Serving for testing:

```
bash ./tools/install_server.sh
```

Note that you will first need to install Docker according to its **installation instructions** for your platform.

The ML deployment and retraining Python scripts will use the same SSH credentials that were used to upload the packaged models. These credentials are stored as CircleCI **environment variables** and written to the .env configuration. They will be used to interact with Docker on the deployment server.

## Adding deployment and retraining jobs to the CircleCI configuration

Below, deploy and test-deployment steps are added to the **existing build-deploy workflow** to be run after the package step:

```
# Do not deploy without manual approval - you can inspect
the console output from training and make sure you are happy
to deploy
- deploy:
    requires:
        - package
- test-deployment:
    requires:
        - deploy
```

A retrain-deploy workflow has also been defined to include the new scripts. In this example, it is triggered according to a **schedule** defined using **cron** syntax. To see this scheduled workflow in action, you will need to create a branch in your Git repository named `retrain`.

```
retrain-deploy:
    # Trigger on a schedule or when retrain branch is
updated
    triggers:
        - schedule:
            cron: “0 0 * * *” # Daily
            filters:
            branches:
                only:
                - retrain
    jobs:
        - install-build
        - retrain:
            requires:
            - install-build
        # Do not redeploy without manual approval - you can
inspect the console output from training and make sure you
are happy to deploy the retrained model
        - hold: # A job that will require manual approval in
the CircleCI web application.
```

```
            requires:
            - retrain
            type: approval # This key-value pair will set
your workflow to a status of “On Hold”
        - package:
            requires:
            - hold
        - deploy:
            requires:
            - package
        - test-deployment:
            requires:
            - deploy
```

In the `retrain-deploy` pipeline, a `hold` step has been added between the `retrain` and `package` steps. Pipeline execution will pause here until approval to proceed is given in the CircleCI web console, which is highly useful in ML pipelines where the accuracy of a retrained model needs to be verified before it is used.

The `on_fail` **condition** is demonstrated within the `retrain` job that is called in this workflow. This allows you to take specific actions when a job fails:

```
   - run:
          # You could trigger custom notifications here so
that the person responsible for a particular job is notified
via email, Slack, etc.
          name: Run on fail status
          command: |
              echo “I am the result of the above failed job”
          when: on_fail
```

By ensuring that your ML scripts are verbose, you can make sure that the user has the information they require to confirm that a model is ready for use. By throwing exceptions when retraining conditions are not met, pipelines can be halted entirely so that problems can be rectified before they reach production.

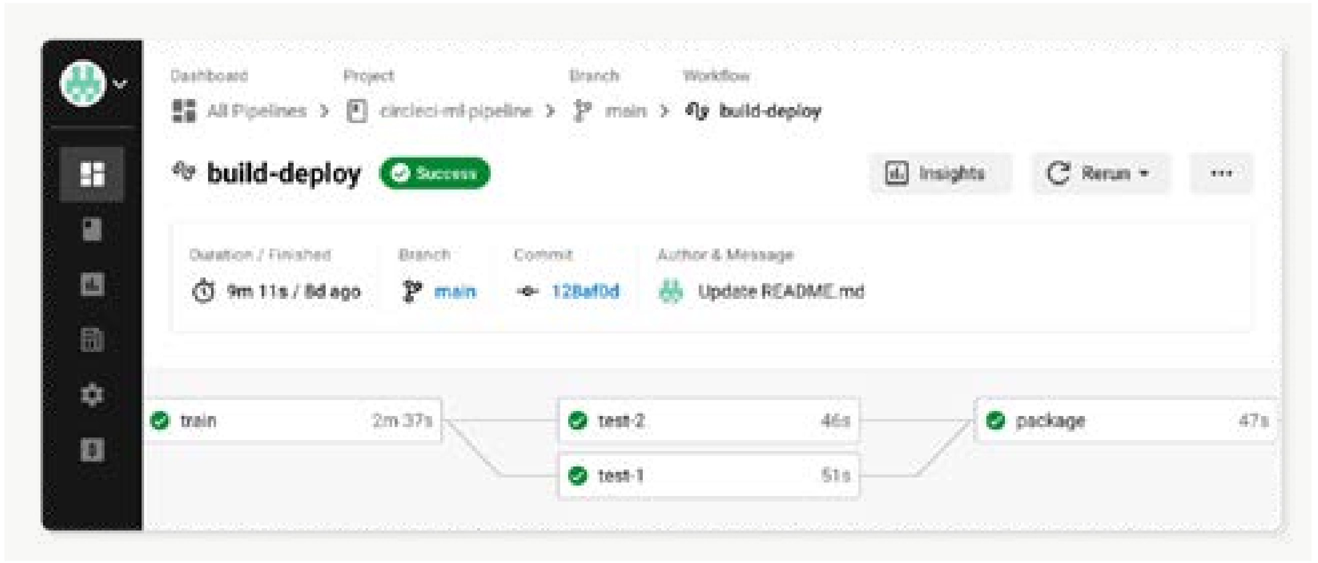
## Scheduling, branches, and manual pipeline execution

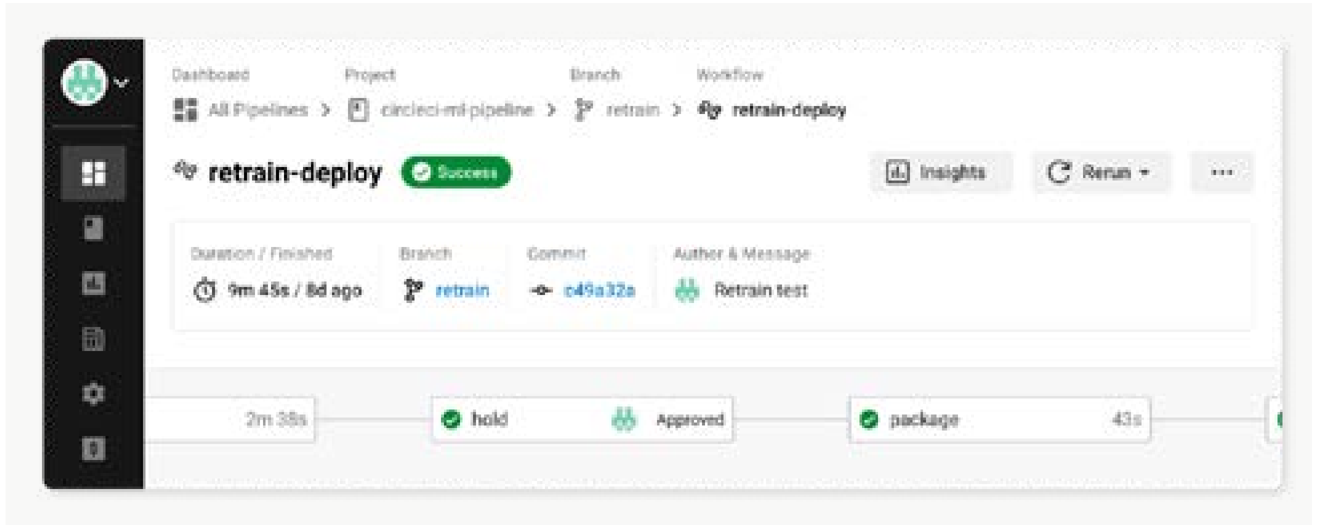
As shown in the code above, the `retrain-deploy` pipeline is run according to a user-defined schedule. This differs from the `build-deploy` pipeline, which only runs when a specified branch is updated.

You can also manually trigger a pipeline at any time from the CircleCI web console, rerun failed jobs, or trigger a pipeline to run using the **CircleCI API**. Using the CircleCI API, you can set up your data ingestion tools to **trigger a CircleCI pipeline externally** when new data has arrived.

## Using CircleCI to monitor your ML CI/CD pipelines

Once your CircleCI configuration has been committed to your Git repository, CircleCI will execute the workflows defined in it based on the defined filters and schedules. You will be able to see the output of tasks undertaken by CircleCI in the web console:





Jobs can be held for approval, and if a job fails, you can rapidly respond and confirm the issue in the CircleCI UI by **rerunning only the failed parts of your workflow**.

As your ML requirements and workflows expand, you can offload the increasing number of ML management and monitoring tasks to scripts triggered by CircleCI pipelines. This way, your workload will be significantly reduced, automated tasks will run and complete as data arrives or on a schedule, and you'll only have to take action when there's a problem.

## CUSTOMIZING NOTIFICATIONS

By default, you will receive notifications on job failures and required approvals to your default CircleCI email address. Additionally, you can **configure other notifications behavior**, including adding other team members to receive notifications on your pipelines, setting up web notifications, and connecting your CircleCI pipeline to Slack or IRC.

By customizing your notifications, you can make sure that the right person is notified to fix a failed job and that your ML system stays accurate and available.

## RESPONDING TO PROBLEMS IN PRODUCTION

Once your model is deployed, monitoring and logging will be handled by your ML platform. You can see how this is configured in TensorFlow in **this guide**.

Using the CircleCI API, you can configure your production monitoring tools to trigger CircleCI pipelines to run so that you can roll back, retrain, or redeploy models to rapidly respond to incidents.

Automation significantly reduces the amount of time your team spends operating and monitoring your ML systems, freeing you to spend more time building and less time on administrative overhead.

## Running your CI/CD workflows in the cloud using CircleCI's managed cloud compute resources

This example used a self-hosted runner to execute commands in a local environment. This is advantageous when dealing with privileged data that you do not want to leave your network but requires that you have your own local machines for the task.

You can run ML tasks (or any CI/CD task) directly on CircleCI's managed compute resources by specifying a Docker, Linux VM (virtual machine), macOS, Windows, GPU, or Arm execution environment in your CircleCI configuration. You can also use your own Docker images with **authenticated pulls**. Hosted environments are run on CircleCI's managed cloud compute, so you don't need your own hardware — your workflows will be run on demand with automatically provisioned compute resources.

For example, to execute your jobs in a pre-built Python Docker container, you would replace the `machine` and `resource_class` options in the job with the following configuration code:

```
docker:
    - image: cimg/python:3.11.4
```

When using cloud compute, you will need to provide the execution environment access to your data. This can be done by using **SSH tunneling**, configuring a VPN, or using CircleCi's **orbs** to access resources stored on public clouds such as AWS, Google Cloud Platform, or Azure. One common use case is to share ML models and data in an **AWS S3 bucket**, which can be authenticated and accessed by on-site infrastructure and CircleCI using **OIDC**.

Workspaces can also be used to transfer local data to cloud workloads. With minimal additional configuration, CircleCI workflows can run jobs that are configured to run on local runners or CircleCI's managed cloud compute, with data persisted between them.

## Using GPU resources for ML tasks in the cloud and locally

Along with specifying the CPU and memory available to your CircleCI cloud compute using resource classes, you can also run your ML tasks in cloud-hosted GPU execution environments. Training and retraining ML models is a compute-intensive task. Leveraging GPU processing power will greatly speed up the process, allowing you to train faster or train and test multiple datasets in parallel.

To use GPU resources in CircleCI, specify a GPU-enabled machine image in your configuration:

```
machine:
  image: ubuntu-2004-cuda-11.4:202110-01
```

If you have large data processing requirements that make the cost of using cloud resources prohibitive, you can use your own self-hosted runners with your own physical GPUs.

Once you have an environment with **GPU resources available**, and if your ML platform supports it, you can configure your ML package to utilize them. See **this guide** on how to do this with TensorFlow. **GPUs make short work** of compute-intensive applications and are therefore extremely well suited for processing ML models.

By combining local runners with cloud GPU resources and CircleCI's **workspace** functionality, you can access and prepare your ML training data on-site and then use it in the cloud without having to set up complex infrastructure for granting cloud resources access to your internal data stores. Conversely, if you are concerned about your cloud-compute costs, you can move data from the cloud on-site and execute your ML tasks on local GPUs using CircleCI's self-hosted runners.

## Conclusion

If you followed the steps in this guide, you now have a fully functional MLOps pipeline capable of continuously building, testing, training, deploying, and retraining your machine learning models. This frees your ML experts from manually running pipelines, testing data, and deploying vetted models, allowing them to focus on building more accurate models and features.

This MLOps pipeline will also enable your organization to efficiently track model performance and update models as needed to ensure that the models are performing optimally for the given data. By leveraging automation and tracking performance, your teams can improve the development cycle of ML models, reducing the amount of time it takes for models to go from development to production. Ultimately, by utilizing this MLOps pipeline, your organization will be able to make more accurate decisions faster, with higher confidence.

CircleCI can do much more than we can demonstrate in the space of this guide. It provides a flexible, scalable platform for accomplishing fully bespoke ML workflows to suit any use case. You can **get started today with a free account**, or **reach out** to our team for personalized help finding the right plan and setting up customizable CI/CD pipelines tailored to your specific ML project needs.