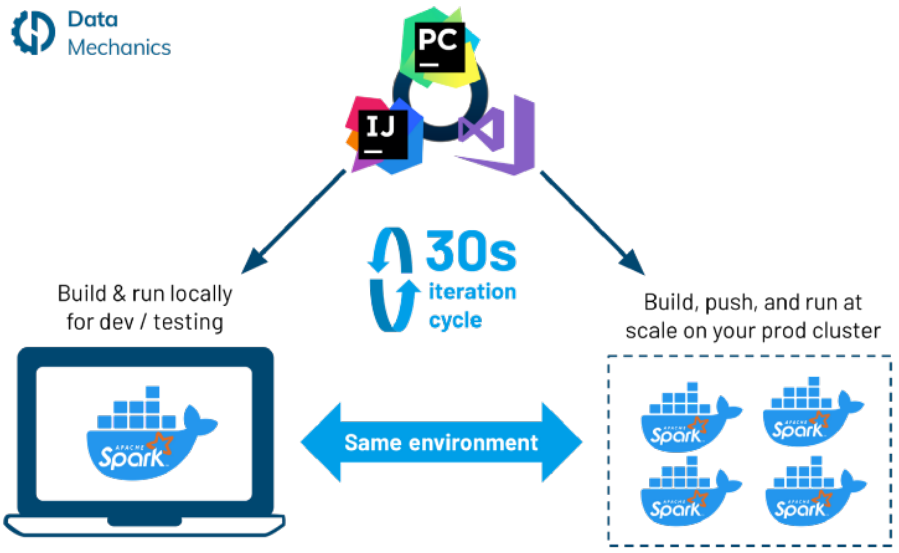
Published in Towards Data Science

Jean Yves Follow
Nov 4, 2020 · 7 min read · Listen

Spark and Docker: Your Spark development cycle just got 10x faster!



Spark & Docker Development Iteration Cycle. Image by author.

The benefits that come with using Docker containers are well known: they provide consistent and isolated environments so that applications can be deployed anywhere — locally, in dev / testing / prod environments, across all cloud providers, and on-premise — in a repeatable way.

The software engineering world has fully adopted Docker, and a lot of tools around

In the big data and Apache Spark scene, most applications still run directly on virtual machines without benefiting from containerization. The dominant cluster manager for Spark, Hadoop YARN, did not support Docker containers until recently (Hadoop 3.1 release), and even today the support for Docker remains “experimental and non-complete”.

Native support for Docker is in fact one of the main reasons companies choose to deploy Spark on top of Kubernetes instead of YARN. The Spark-on-Kubernetes project received a lot of backing from the community, until it was declared Generally Available and Production Ready as of Apache Spark 3.1 in March 2021.

In this article, we will illustrate the benefits of Docker for Apache Spark by going through the end-to-end development cycle used by many of our users at Data Mechanics.

The benefits of Docker for Apache Spark

1. Build your dependencies once, run everywhere

Your application will run the same way wherever you run it: on your laptop for dev / testing, or in any of your production environments.In your Docker image, you will:

- Package your application code
- Package all your dependencies (python: pypi, eggs, conda, scala / java: jars, maven; system dependencies)
- Define environment variables to tweak behavior at runtime
- Customize your operating system the way you want
- If running on Kubernetes: you’re free to choose your Spark version for each separate Docker image! This is in contrast to YARN, where you must use the same global Spark version for the entire cluster.

2. Make Spark more reliable and cost-efficient in production.



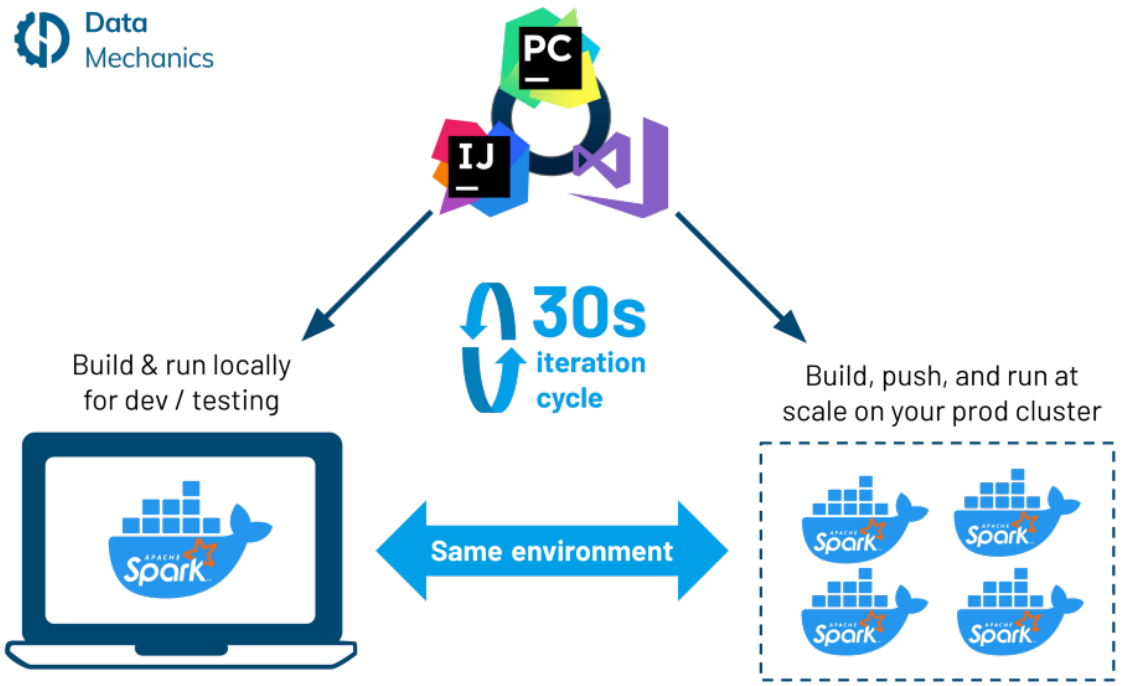

These scripts have several issues:

- They're not reliable — a library installation can fail due to a spurious network failure and be responsible for your Spark pipeline failure in production.
- They're slow — these runtime scripts can add multiple minutes to the setup time of each of your Spark nodes, which increases your costs.
- They're hard to develop, maintain, and debug.

3. Speed up your iteration cycle.

At Data Mechanics, our users enjoy a 20–30s iteration cycle, from the time they make a code change in their IDE to the time this change runs as a Spark app on our platform.

This is mostly thanks to the fact that Docker caches previously built layers and that Kubernetes is really fast at starting / restarting Docker containers.



Spark & Docker Development Iteration Cycle. Image by author.

Tutorial: How to speed up your Spark development cycle by 10x with

to work on other environments.

1. Building the Docker Image

We'll start from a local PySpark project with some dependencies, and a Dockerfile that will explain how to build a Docker image for this project. Once the Docker image is built, we can directly run it locally, or push to a Docker registry first then run it on the Data Mechanics cluster.

Thanks to Docker, there's no need to package third-party dependencies and custom modules in brittle zip files. You can just co-locate your main script and its dependencies.

In this example, we adopt a layout for our project that should feel familiar to most Python developers: a main script, a requirements file, and custom modules.

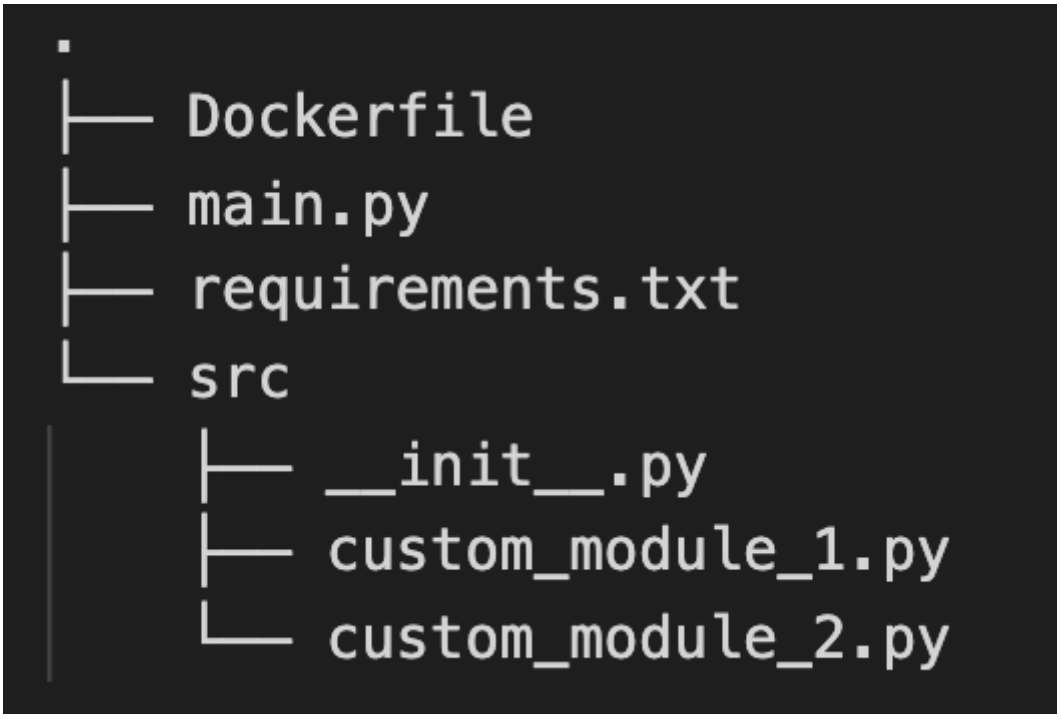


Image by author.

Let's look at the Dockerfile. In this example:

1. We start from a base image published by Data Mechanics. This image corresponds to a major Spark image (3.0 here). We discuss this choice in detail

3. We finally copy our own code and main file. Since this is the code we'll be primarily iterating on, it's recommended to do this at the end, so that when we change this code Docker doesn't need to rebuild the image from scratch.

4. It's also easy to set up environment variables at this step.

```
FROM datamechanics/spark:3.1-latest

ENV PYSPARK_MAJOR_PYTHON_VERSION=3

WORKDIR /opt/application/

COPY requirements.txt .
RUN pip3 install -r requirements.txt

COPY src/ src/

COPY main.py .
```

Example Dockerfile. Image by Author.

2. Local Run

```
docker build -t {{image_name}} .
docker run {{image_name}} driver local:///opt/application/main.py {{args}}
```

Image by author.

You can then build this image and run it locally. This means Spark will run in local mode; as a single container on your laptop. You will not be able to process large amounts of data, but this is useful if you just want to test your code correctness (maybe using a small subset of the real data), or run unit tests.

On an average laptop, this took 15 seconds. If you're going to iterate a lot at this stage and want to optimize the speed further; then instead of re-building the image, you can simply mount your local files (from your working directory on your laptop) to your working directory in the image.

3. Run at Scale

The code looks correct, and we now want to run this image at scale (in a distributed way) on the full dataset. We're going to push the image to our Docker registry and then provide the image name as a parameter in the REST API call to submit this Spark application to Data Mechanics.

```
docker build -t {{image_name}} .
docker push {{image_name}}
curl --request POST {{cluster_url}}/api/apps/ \
    --header 'Content-Type: application/json' \
    --header 'X-API-Key: {{api_key}}' \
    --data-raw '{
        "jobName": "spark-summit-demo",
        "configOverrides": {
            "type": "Python",
            "image": "{{image_name}}",
            "imagePullPolicy": "Always",
            "mainApplicationFile": "local:///opt/application/main.py",
            "arguments": [{{args}}],
        }
    }'
```

Image by author.

It's important to have a fast iteration cycle at this stage too — even though you ran the image locally — as you'll need to verify the correctness and stability of your pipeline on the full dataset. It's likely you will need to keep iterating on your code and on your infrastructure configurations.

On an average laptop, this step took 20 seconds to complete: 10 seconds to build and push the image, 10 seconds for the app to start on Data Mechanics — meaning the Spark driver starts executing our **main.py** code.

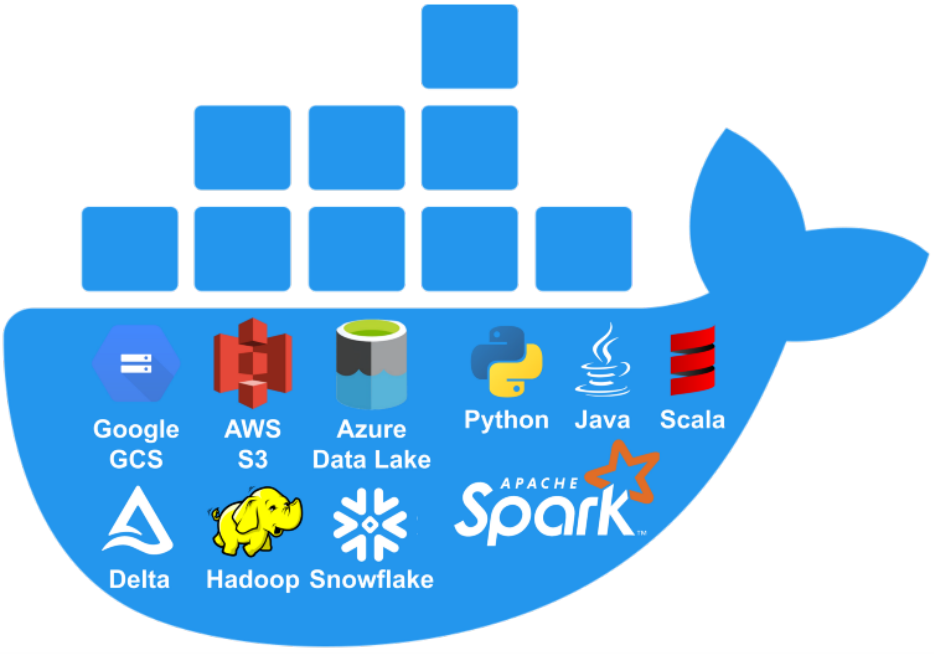
paste your code to a notebook or an interactive shell). This is a game-changer for Spark developers, a 10x speed-up compared to industry average.

> *"I really love that we can use Docker to package our code on Data Mechanics. It's a huge win for simplifying PySpark deployment. Based on the iterations I have completed it takes less than 30 seconds to deploy and run the app"*
>
> *— Max, Data Engineer at Weather2020.*

This fast iteration cycle is thanks to Docker caching the previous layers of the image, and the Data Mechanics platform (deployed on Kubernetes) optimizing fast container launches and application restarts.

How to choose your Spark base Docker image

Since April 2021, Data Mechanics maintains a public fleet of Docker Images that come built-in with Spark, Java, Scala, Python, Hadoop, and connectors with common data sources like S3, GCS, Azure Data Lake, Delta Lake, and more. We regularly push updates to these images whenever a new version of Spark or one of the included dependency is published, and test them across various workloads and virements.



- If you're a Data Mechanics customer, refer to our documentation. Our platform images have a higher availability guarantee, and a few exclusive platform improvements (like Jupyter Notebook support).
- For non Data Mechanics customers, our optimized images are now freely available on our public DockerHub repository. You can them to run Spark locally, on Kubernetes, or on other architectures.

Conclusion

- Package your dependencies and control your environment in a simple way.
- Iterate on your code from your IDE by quickly running Spark locally or at scale
- Make Spark more reliable and cost-efficient in production. Finally, you can say goodbye to slow and flaky bootstrap scripts and runtime downloads!

We've demonstrated how Docker can help you:

If this sounds like something you would like to try out on the Data Mechanics platform, book a demo with our team to start a trial of Data Mechanics — we'll help you adopt this Docker-based dev workflow as one of the first steps of our onboarding.

Running Spark on Kubernetes in a Docker native and cloud agnostic way has many other benefits. For example, you can use the steps above to build a CI/CD pipeline that automates the building and deployment of Spark test applications whenever you submit a pull request to r 166 2 ion pipelines. We will cover this in a future blog post — so stay tuned!