Valohai

ENGINEERING PRACTICES
for Data Scientists

Preview Edition

Valohai

Contents

More chapters coming soon

About the author

Juha Kiili

Senior Software Developer, Product Owner at Valohai

Senior Software Developer with gaming industry background shape-shifted into a full-stack ninja. I have the biggest monitor.

Valohai

Foreword

Software engineering has come a long way. It's no longer just about getting a functioning piece of code on a floppy disk; it's about the craft of making software. There's a good reason for it too. Code lives for a long time.

Thus there are a lot of strong opinions about good engineering practices that make developing software for the long haul possible and more enjoyable. I think enjoyability is an important word here because most software developers know the pain of fixing poorly developed and poorly documented legacy software.

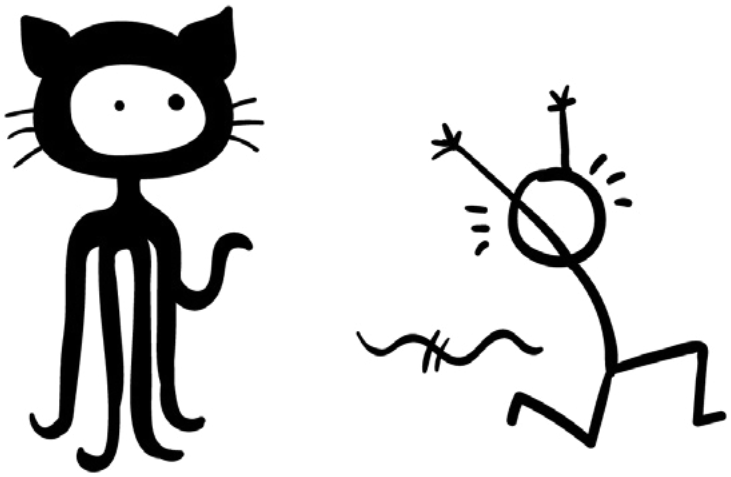
Data scientists are also entering this world because machine learning is becoming a core part of many products. While a heterogenous bunch with various backgrounds, data scientists are more commonly from academia and research than software engineering. The slog of building and maintaining software isn't as familiar as it is to most developers, but it will be soon enough. It's better to be prepared with a solid foundation of best practices, so it'll be easier to work with software engineers, and it'll be easier to maintain what you build.

This eBook is to help pick up engineering best practices with simple tips. I hope that we can teach even the most seasoned pros something new and get you talking with your team on how you should be building things. Remember, as machine learning becomes a part of software products, it too will live for a long time.

This eBook isn't about Valohai – although there is a section about our MLOps platform at the end – but good engineering is close to our heart.

Valohai

What every data scientist should know about Git



What is Git?

Git is a version control system designed to track changes in a source code over time.

When many people work on the same project without a version control system it's total chaos. Resolving the eventual conflicts becomes impossible as none has kept track of their changes and it becomes very hard to merge them into a single central truth. Git and higher-level services built on top of it (like Github) offer tools to overcome this problem.

Usually, there is a single central repository (called "origin" or "remote") which the individual users will clone to their local machine (called "local" or "clone"). Once the users have saved meaningful work (called "commits"), they will send it back ("push" and "merge") to the central repository.

Valohai

What is the difference between Git & GitHub?

Git is the underlying technology and its command-line client (CLI) for tracking and merging changes in a source code.

GitHub is a web platform built on top of git technology to make it easier. It also offers additional features like user management, pull requests, automation. Other alternatives are for example GitLab and Sourcetree.

Terminology

- **Repository** - "Database" of all the branches and commits of a single project
- **Branch** - Alternative state or line of development for a repository.
- **Merge** - Merging two (or more) branches into a single branch, single truth.
- **Clone** - Creating a local copy of the remote repository.
- **Origin** - Common alias for the remote repository which the local clone was created from
- **Main / Master** - Common name for the root branch, which is the central source of truth.
- **Stage** - Choosing which files will be part of the new commit
- **Commit** - A saved snapshot of staged changes made to the file(s) in the repository.
- **HEAD** - Shorthand for the current commit your local repository is currently on.
- **Push** - Pushing means sending your changes to the remote repository for everyone to see
- **Pull** - Pulling means getting everybody else's changes to your local repository
- **Pull Request** - Mechanism to review & approve your changes before merging to main/master

Valohai

Basic commands

- `git init` (Documentation) - Create a new repository on your local computer.
- `git clone` (Documentation) - Start working on an existing remote repository.
- `git add` (Documentation) - Choose file(s) to be saved (staging).
- `git status` (Documentation) - Show which files you have changed.
- `git commit` (Documentation) - Save a snapshot (commit) of the chosen file(s).
- `git push` (Documentation) - Send your saved snapshots (commits) into the remote repository.
- `git pull` (Documentation) - Pull recent commits made by others into your local computer.
- `git branch` (Documentation) - Create or delete branches.
- `git checkout` (Documentation) - Switch branches or undo changes made to local file(s).
- `git merge` (Documentation) - Merge branches to form a single truth.

Valohai

Rules of thumb for Git

Don't push datasets

Git is a version control system designed to serve software developers. It has great tooling to handle source code and other related content like configuration, dependencies, documentation. It is not meant for training data. Period. Git is for code only.

In software development, code is king and everything else serves the code. In data science, this is no longer the case and there is a duality between data and code. It doesn't make sense for the code to depend on data any more than it makes sense for data to depend on code. They should be decoupled and this is where the code-centric software development model fails you. **Git shouldn't be the central point of truth for a data science project.**

There are extensions like LFS that refer to external datasets from a git repository. While they serve a purpose and solve some of the technical limits (size, speed), they do not solve the core problem of a code-centric software development mindset rooted in git.

You will always have datasets floating around in your local directory though. It is quite easy to accidentally stage and commit them if you are not careful. The correct way to make sure that you don't need to worry about datasets with git is to use the `.gitignore` config file. Add your datasets or data folder into the config and never look back.

Example:

```
# ignore archives
*.zip
*.tar
*.tar.gz
*.rar

# ignore dataset folder and subfolders
datasets/
```

Don't push secrets

This should be obvious, yet the constant real-world mistakes prove to us it is not. It doesn't matter if the repository is private either. In no circumstances should anyone commit any username, password, API token, key code, TLS certificates, or any other sensitive data into git.

Even private repositories are accessible by multiple accounts and are also cloned to multiple local machines. This gives the hypothetical attacker exponentially more targets. Remember that private repositories can also become public at some point.

Decouple your secrets from your code and pass them using the environment instead. For Python, you can use the common `.env` file with which holds the environment variables, and the `.gitignore` file which makes sure that the `.env` file doesn't get pushed to the remote git repository. It is a good idea to also provide the `.env.template` so others know what kind of environment variables the system expects.

.env:

```
API_TOKEN=98789fsda789a89sdafsa9f87sda98f7sda89f7
```

.env.template:

```
API_TOKEN=
```

.gitignore:

```
.env
```

hello.py:

```
from dotenv import load_dotenv
load_dotenv()
api_token = os.getenv('API_TOKEN')
```

This still requires some manual copy-pasting for anyone cloning the repository for the first time. For more advanced setup, there are encrypted, access-restricted tools that can share secrets through the environment, such as Vault.

Note: If you already pushed your secrets to the remote repository, do not try to fix the situation by simply deleting them. It is too late as git is designed to be immutable. Once the cat is out of the bag, the only valid strategy is to change the passwords or disable the tokens.

Valohai

Don't push notebook outputs

Notebooks are cool because they let you not only store code but also the cell outputs like images, plots, tables. The problem arises when you commit and push the notebook with its outputs to git.

The way notebooks serialize all the images, plots, and tables is not pretty. Instead of separate files, it encodes everything as JSON gibberish into the `.ipynb` file. This makes git confused.

Git thinks that the JSON gibberish is equally important as your code. The three lines of code that you changed are mixed with three thousand lines that were changed in the JSON gibberish. Trying to compare the two versions becomes useless due to all the extra noise.

Source: [ReviewNB Blog]

It becomes even more confusing if you have changed some code after the outputs were generated. Now the code and outputs that are stored in the version control do not match anymore.

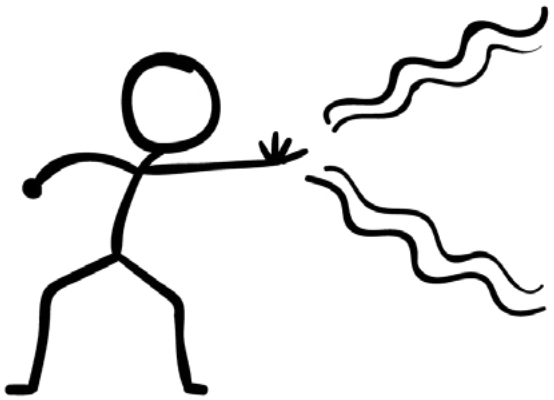
There are two options at your disposal.

You can manually clear the outputs from the main menu (Cells -> All Output -> Clear) before creating your git commit.

You can set up a pre-commit hook for git that clears outputs automatically

We highly recommend investing to option #2 as manual steps that you need to remember are destined to fail eventually.

Don't use the --force



Sometimes when you try to push to the remote repository, git tells you that something is wrong and aborts. The error message might offer you an option to "use the force" (the `-f` or `--force`). Don't do it! Even if the error message calls for your inner Jedi, just don't. It's the dark side.

Obviously, there are reasons why the `--force` exists and it serves a purpose in some situations. None of those arguments apply to you young padawan. Whatever the case, read the error message, try to reason what could be the issue, ask someone else to help you if needed, and get the underlying issue fixed.

Do small commits with clear descriptions

Inexperienced users often fall into the trap of making huge commits with nonsensical descriptions. A good rule of thumb for any single git commit is that it should only do one thing. Fix one bug, not three. Solve one issue, not twelve. Remember that issues can often be split into smaller chunks, too. The smaller you can make it, the better.

The reason you use version control is that someone else can understand what has happened in the past. If your commit fixes twelve bugs and the description says "Model fixed", it is close to zero value two months later. The commit should only do one thing and one thing only. The description should communicate the thing was. You don't need to make the descriptions long-winded novels if the commits are small. In fact, a long description for a commit message implies that the commit is too big and you should split it into smaller chunks!

Valohai

Example #1: a bad repository

```
* 39b54d7 - new stuff (3 months ago) <Juha Kiili>
* 16ec3bf - New stuffy (3 months ago) <Juha Kiili>
* 07ab2e4 - Boy 2! (4 months ago) <Juha Kiili>
* c8ef0db - new stuffs again (4 months ago) <Juha Kiili>
* b04b15c - ML! (4 months ago) <Juha Kiili>
* 2bdcc65 - First ML stuff (4 months ago) <Juha Kiili>
* 3378d9d - New stuff (4 months ago) <Juha Kiili>
* 319d616 - Fixes (4 months ago) <Juha Kiili>
```

Example #2: a good repository

```
* ce12c69c0 - Use `config_dict_or_path` for deepspeed.zero.Init (#13614) (2 months ago) <Alex Hedges>
* 0eb02871d - Removed console spam from misfiring warnings (#13625) (2 months ago) <Matt>
* da8beaaf7 - Fix special tokens not correctly tokenized (#13489) (2 months ago) <Li-Huai (Allan) Lin>
* 1f9dcfc1e - [Trainer] Add nan/inf logging filter (#13619) (2 months ago) <Patrick von Platen>
* cac7a90b7 - Optimize Token Classification models for TPU (#13096) (2 months ago) <Ibraheem Moosa>
* e02ed0ee7 - XLMR tokenizer is fully picklable (#13577) (3 months ago) <Benjamin Davidson>
* af5c6ae5e - Properly use test_fetcher for examples (#13604) (3 months ago) <Sylvain Gugger>
```

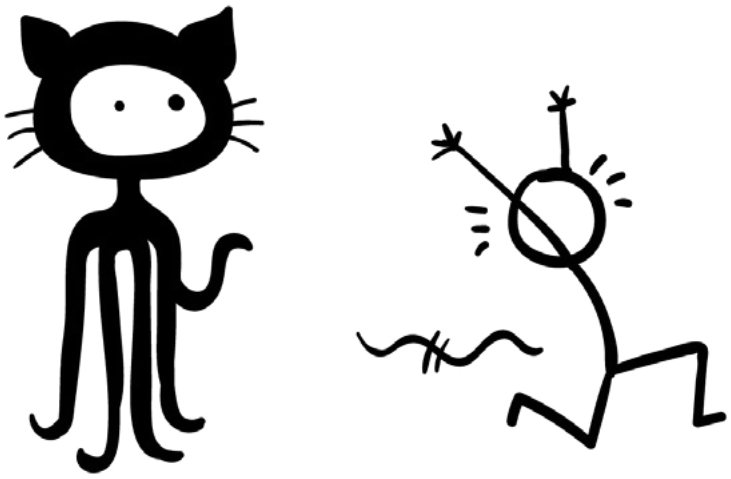
In real life you often make all kinds of ad-hoc things and end up in the situation #1 on your local machine. If you haven't pushed anything to the public remote yet, you can still fix the situation. We recommend learning how to use the interactive rebase.

Simply use:

```
git rebase –i origin/main
```

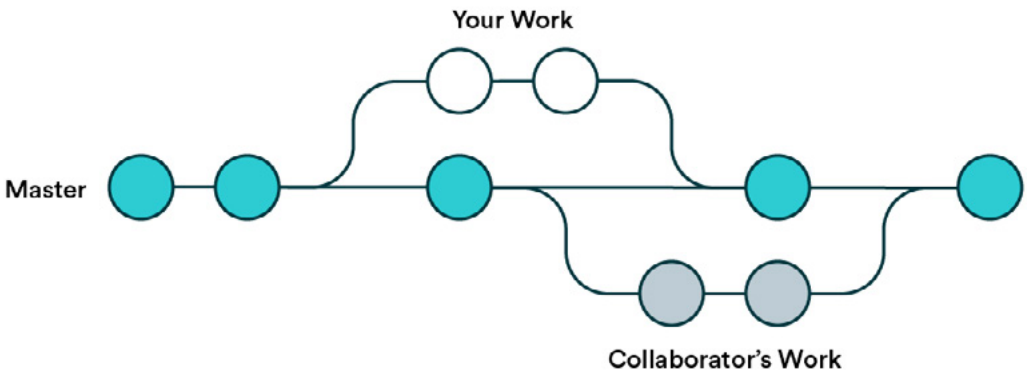
The interactive mode offers many different options for tweaking the history, rewording commit messages, and even changing the order. Learn more about the interactive rebase from here.

Don't be afraid of branching & pull requests

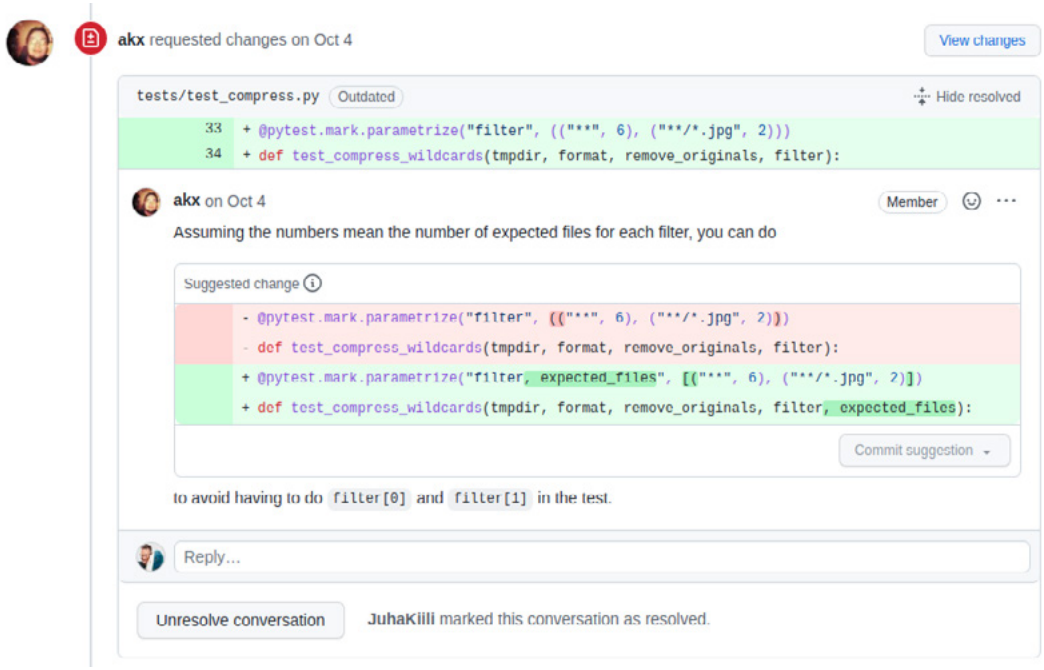


Branching and especially pull requests are slightly more advanced and not everyone's cup of tea, but if your data science project is mature, in production, and constantly touched by many different people, pull requests may be just the thing that is missing from your process.

When you create a new git repository, it will start with just a single branch called main (or master). The main branch is considered as the "central truth". Branching means that you will branch out temporarily to create a new feature or a fix to an old one. In the meantime, someone else can work in parallel on their own branch. This is commonly referred to as feature branch workflow.



The idea with branches is to eventually merge back to the `main` branch and update "the central truth". This is where pull requests come into play. The rest of the world doesn't care about your commits in your own branch, but merging to `main` is where your branch becomes the latest truth. That is when it's time to make a pull request.



Pull requests are not a git concept, but a GitHub concept. They are a request for making your branch the new central truth. Using the pull request, other users will check your changes before they are allowed to become the new central truth. GitHub offers great tools to make comments, suggest their modifications, signal approval, and finally apply the merge automatically.

Valohai

What every data scientist should know about Python dependencies

What is dependency management anyway?

Dependency management is the act of managing all the external pieces that your project relies on. It has the risk profile of a sewage system. When it works, you don't even know it's there, but it becomes excruciating and almost impossible to ignore when it fails.

Every project is built on someone else's sweat and tears. Those days when an engineer woke up, made coffee, and started a new project by writing a bootloader – the program that boots up your computer from scratch – are history. There are massive stacks of software and libraries beneath us. We are simply sprinkling our own thin layer of sugar on top.

My computer has a different stack of software than yours. Not only are the stacks different, but they are forever changing. It is amazing how *anything* works, but it does. All thanks to the sewage system of dependency management and a lot of smart people abstracting the layers so that we can just call our favorite pandas function and get predictable results.

Basics of Python dependency management

Let's make one thing clear. Simply Installing and upgrading Python packages is not dependency management. Dependency management is documenting the required environment for your project and making it easy and deterministic for others to reproduce it.

You could write installation instructions on a piece of paper. You could write them in your source code comments. You could even hardcode the install commands straight into the program. Dependency management? Yes. Recommended? Nope.

The recommended way is to decouple the dependency information from the code in a standardized, reproducible, widely-accepted format. This allows version pinning and easy deterministic installation. There are many options, but we'll describe the classic combination of pip and requirements.txt file in this article.

But before we go there, let's first introduce the atomic unit of Python dependency: the package.

What is a package?

"Package" is a well-defined term in Python. Terms like library, framework, toolkit are not. We will use the term "package" for the remainder of this article, even for the things that some refer to as libraries, frameworks, or toolkits.

A module is everything defined in a single Python file (classes, functions, etc.).

A package is a collection of modules.

Pandas is a package, Matplotlib is a package, print()-function is not a package. The purpose of a package is to be an easily distributable, reusable, and versioned collection of modules with well-defined dependencies to other packages.

You are probably working with packages every day by referring to them in your code with the Python `import` statement.

The art of installing packages

While you could install packages by simply downloading them manually to your project, the most common way to install a package is via PyPi (Python Package Index) using the famous `pip install` command.

Note: Never use `sudo pip install`. Never. It is like running a virus. The results are unpredictable and will cause your future self major pain.

Never install Python packages globally either. Always use virtual environments.

What are virtual environments?

Python virtual environment is a safe bubble. You should create a protective bubble around all the projects on your local computer. If you don't, the projects will hurt each other. Don't let the sewage system leak!

If you call `pip install pandas` outside the bubble, it will be installed globally. This is bad. The world moves forward and so do packages. One project needs the Matplotlib of 2019 and the other wants the 2021 version. A single global installation can't serve both projects. So protective bubbles are a necessity. Let's look at how to use them.

Go to your project root directory and create a virtual environment:

```
python3 -m venv mybubble
```

Now we have a bubble, but we are not in it yet. Let's go in!

```
source mybubble/bin/activate
```

Now we are in the bubble. Your terminal should show the virtual environment name in parenthesis like this:

```
(mybubble) johndoe@hello:~/myproject$
```

Now that we are in the bubble, installing packages is safe. From now on, any pip install command will only have effects inside the virtual environment. Any code you run will only use the packages inside the bubble.

If you list the installed packages you should see a very short list of currently installed default packages (like the pip itself).

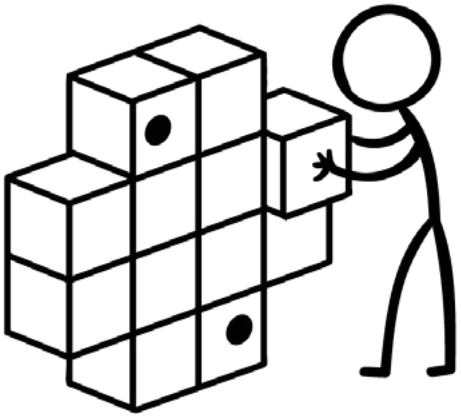
```
pip list

Package        Version
-------------- -------
pip            20.0.2
pkg-resources  0.0.0
setuptools     44.0.0
```

This listing is no longer for all the Python packages in your machine, but all the Python packages inside your virtual environment. Also, note that the Python version used inside the bubble is the Python version you used to create the bubble.

To leave the bubble, simply call `deactivate` command.

Always create virtual environments for all your local projects and run your code inside those bubble(s). The pain from conflicting package versions between projects is the kind of pain that makes people quit their jobs. Don't be one of those people.

What is version pinning?



Imagine you have a project that depends on Pandas package and you want to communicate that to the rest of the world (and your future self). Should be easy, right?

First of all, it is risky to just say: "You need Pandas".

The less risky option is "You need Pandas 1.2.1", but even that is not always enough.

Let's say you are correctly pinning the Pandas version to 1.2.1. Pandas itself has a dependency for numpy, but unfortunately doesn't pin the dependency to an exact numpy version. Pandas itself just says "You need numpy" and does not pin to an exact version.

At first, everything is fine, but after six months, a new numpy version 1.19.6 is released with a showstopper bug.

Now if someone installs your project, they'll get pandas 1.2.1 with buggy numpy 1.19.6, and probably a few gray hairs as your software spits weird errors. The sewage system is leaking. The installation process was not deterministic!

The most reliable way is to pin everything. Pin the dependencies of the dependencies of the dependencies of the dependencies, of the... You'll get the point. Pin'em as deep as the rabbit hole goes. Luckily there are tools that make this happen for you.

Note: If you are building a reusable package and not a typical project, you should not pin it so aggressively (this is why Pandas doesn't pin to the exact Numpy version). It is considered best practice for the end-user of the package to decide what and how aggressively pin. If you as a package creator pin everything, then you close that door from the end-user.

How do I pin Python dependencies?

Whenever you call `pip install` to get some hot new package into your project, you should stop and think for a second. This will create a new dependency for your project. How do I document this?

You should write down new libraries and their version number to a requirements.txt file. It is a format understood by pip to install multiple packages in one go.

 # requirements.txt

```
pandas==1.2.1
matplotlib==3.4.0
```

 # Install

```
pip install -r requirements.txt
```

This is already much better than most data science projects that one encounters, but we can still do better. Remember the recursive dependency rabbit hole from the previous chapter about version pinning. How do we make the installation more deterministic?

The answer is `pip-compile` command and `requirements.in` text file.

 # Requirements.in

```
matplotlib==3.4.0
```

 # Auto-generate requirements.txt

```
pip-compile requirements.in
```

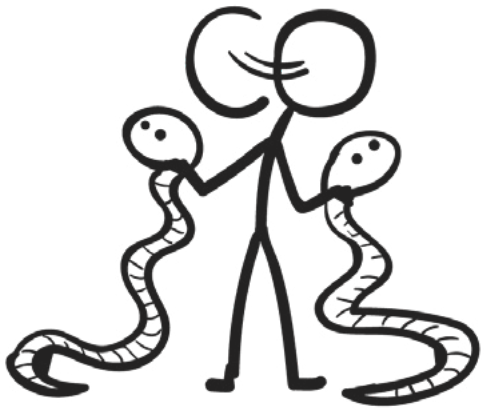
 # Generated requirements.txt

```
cycler==0.11.0
    # via matplotlib
kiwisolver==1.3.2
    # via matplotlib
matplotlib==3.4.0
    # via -r requirements.in
numpy==1.22.0
    # via
    #   matplotlib
    #   pandas
pandas==1.2.1
    # via -r requirements.in
pillow==9.0.0
    # via matplotlib
pyparsing==3.0.6
    # via matplotlib
python-dateutil==2.8.2
    # via
    #   matplotlib
    #   pandas
pytz==2021.3
    # via pandas
six==1.16.0
    # via python-dateutil
```

In the requirements.in you should only put your direct dependencies.

The pip-compile will then generate the perfect pinning of all the libraries into the requirements.txt, which provides all the information for a deterministic installation. Easy peasy! Remember to commit both files into your git repository, too.

How to pin the Python version?



Pinning the Python version is tricky. There is no straightforward way to pin the version dependency for Python itself (without using e.g conda).

You could make a Python package out of your project, which lets you define the Python version in the `setup.py` or `setup.cfg` with the key `python_requires>=3.9`, but that is overkill for a typical data science project, which usually doesn't have the characteristics of a reusable package anyway.

If you are really serious about pinning to specific Python, you could also do something like this in your code:

```
import sys

if sys.version_info < (3,9):
    sys.exit("Python >= 3.9 required.")
```

The most bullet-proof way to force the Python version is to use Docker containers, which we will talk about in the next chapter!

Main takeaways

Don't avoid dependency management - Your future self will appreciate the documented dependencies when you pour coffee all over your MacBook.

Always use virtual environments on your local computer - Trying out that esoteric Python library with 2 GitHub stars is no big deal when you are safely inside the protective bubble.

Pinning versions is better than not pinning - Version pinning protects from packages moving forward when your project is not.

Packages change a lot, Python not so much - Even a single package can have dozens of nested dependencies and they are constantly changing, but Python is relatively stable and future-proof.

What about the cloud?

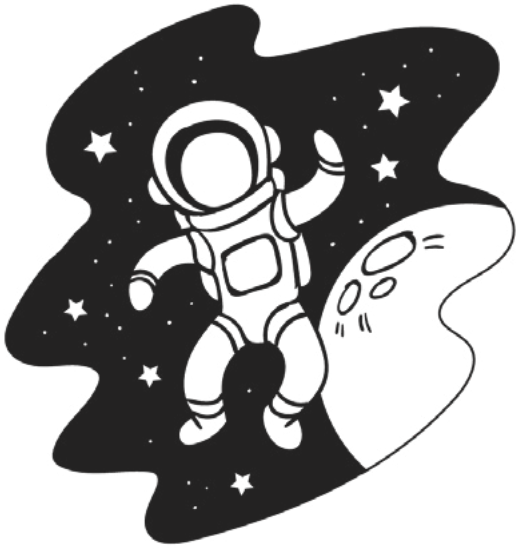
When your project matures enough and elevates into the cloud and into production, you should look into pinning the entire environment and not just the Python stuff.

This is where Docker containers are your best friend as they not only let you pin the Python version but anything inside the operating system. It is like a virtual environment but on a bigger scale.

Valohai

What every data scientist should know about Docker

What is Docker?

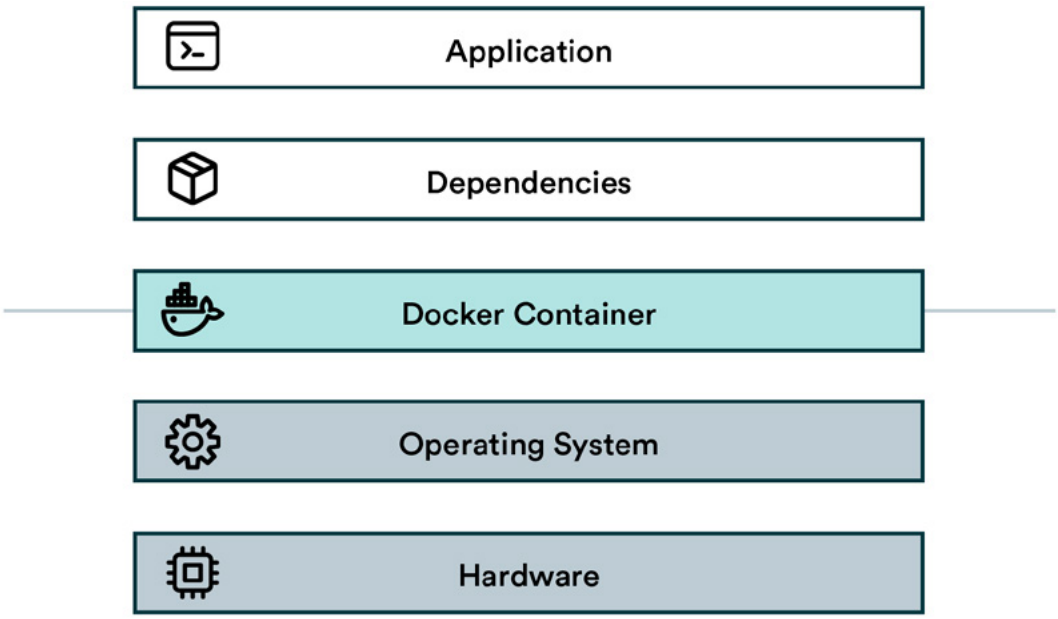


Imagine being an astronaut on a space station and planning to go outside and enjoy the view. You'd be facing hostile conditions. The temperature, oxygen, and radiation are not what you were built for. Human beings require a specific environment to thrive. To properly function in any other scenario, like deep in the sea or high up in space, we need a system to reproduce that environment. Whether it is a spacesuit or a submarine, we need isolation and something that ensures the levels of oxygen, pressure, and temperature we depend on.

In other words, we need a container.

Any software faces the same problem as the astronaut. As soon as we leave home and go out into the world, the environment gets hostile, and a protective mechanism to reproduce our natural environment is mandatory. The Docker container is the spacesuit of programs.

Docker isolates the software from all other things on the same system. A program running inside a “spacesuit” generally has no idea it is wearing one and is unaffected by anything happening outside.



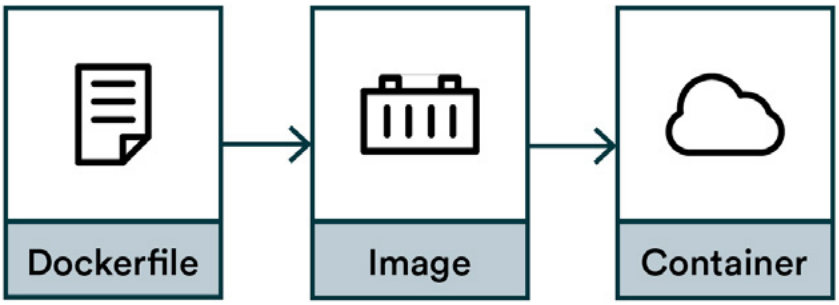
The containerized stack

- **Application:** High-level application (your data science project)
- **Dependencies:** Low-level generic software (think Tensorflow or Python)
- **Docker container:** The isolation layer
- **Operating system:** Low-level interfaces and drivers to interact with the hardware
- **Hardware:** CPU, Memory, Hard disk, Network, etc.

The fundamental idea is to package an application and its dependencies into a single reusable artifact, which can be instantiated reliably in different environments.

How to create a container?

The flow to create Docker containers:



1. **Dockerfile:** Instructions for compiling an image
2. **Image:** Compiled artifact
3. **Container:** An executed instance of the image

Dockerfile

First, we need instructions.

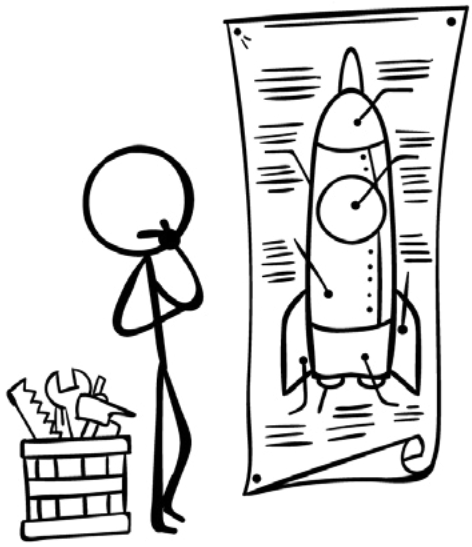
We could define the temperature, radiation, and oxygen levels for a spacesuit, but we need instructions, not requirements. Docker is instruction-based, not requirement-based. We will describe the *how* and not the *what*. To do that, we create a text file and name it `Dockerfile`.

 # Dockerfile

```
FROM python:3.9
RUN pip install tensorflow==2.7.0
RUN pip install pandas==1.3.3
```

The `FROM` command describes a base environment, so we don't need to start from scratch. A treasure trove of base images can be found from the DockerHub or via google searches.

The `RUN` command is an instruction to change the environment.



Note: While our example installs Python libraries one by one, that is not recommended. The best practice is to utilize `requirements.txt`, which defines the Python dependencies. Follow the best practices from our previous chapter to create one.

 # Dockerfile with requirements.txt

```
FROM python:3.9
COPY requirements.txt /tmp
RUN pip install -r /tmp/requirements.txt
```

The COPY command copies a file from your local disk, like the `requirements.txt`, into the image. The `RUN` command here installs all the Python dependencies defined in the `requirements.txt` in one go.

Note: All the familiar Linux commands are at your disposal when using RUN.

Docker image

Now that we have our `Dockerfile`, we can compile it into a binary artifact called an image.

The reason for this step is to make it faster and reproducible. If we didn't compile it, everyone needing a spacesuit would need to find a sewing machine and painstakingly run all the instructions for every spacewalk. That is too slow but also indeterministic. Your sewing machine might be different from mine. The tradeoff for speed and quality is that images can be quite large, often gigabytes, but a gigabyte in 2022 is peanuts anyway.

To compile, use the build command:

```
docker build . -t myimage:1.0
```

This builds an image stored on your local machine. The `-t` parameter defines the image name as "myimage" and gives it a tag "1.0". To list all the images, run:

This builds an image stored on your local machine. The -t parameter defines the image name as "myimage" and gives it a tag "1.0". To list all the images, run:

```
docker image list

REPOSITORY  TAG      IMAGE ID      CREATED      SIZE
<none>      <none>   85eb1ea6d4be  6 days ago   2.9GB
myimagename 1.0      ff732d925c6e  6 days ago   2.9GB
myimagename 1.1      ff732d925c6e  6 days ago   2.9GB
myimagename latest   ff732d925c6e  6 days ago   2.9GB
python      3.9      f88f0508dc46  13 days ago  912MB
```

Docker container

Finally, we are ready for our spacewalk. Containers are the real-life instances of a spacesuit. They are not really helpful in the wardrobe, so the astronaut should perform a task or two while wearing them.

The instructions can be baked into the image or provided just in time before starting the container. Let's do the latter.

```
docker run myimagename:1.0 echo "Hello world"
```

This starts the container, runs a single `echo` command, and closes it down.

Now we have a reproducible method to execute our code in any environment that supports Docker. This is very important in data science, where each project has many dependencies, and reproducibility is at the heart of the process.

Containers close down automatically when they have executed their instructions, but containers can run for a long time. Try starting a very long command in the background (using your shell's & operator):

```
docker run myimagename:1.0 sleep 100000000000 &
```

You can see our currently running container with:

```
docker container list
```

To stop this container, take the container ID from the table and call:

```
docker stop <CONTAINER ID>
```

This stops the container, but its state is kept around. If you call

```
docker ps –a
```

You can see that the container is stopped but still exists. To completely destroy it:

```
docker rm  <CONTAINER ID>
```

The single command combining both stopping and removing:

```
docker rm –f <CONTAINER_ID>
```

To remove all stopped leftover containers:

```
docker container prune
```

Tip: You can also start a container with an interactive shell:

```
$ docker run –it myimagename:1.0 /bin/bash
root@9c4060d0136e:/# echo "hello"
hello
root@9c4060d0136e:/# exit
exit
$ <back in the host shell>
```

It is great for debugging the inner workings of an image when you can freely run all the Linux commands interactively. Go back to your host shell by running the `exit` command.

Terminology and Naming

Registry = Service for hosting and distributing images. The default registry is the Docker Hub.

Repository = Collection of related images with the same name but different tags. Usually, different versions of the same application or service.

Tag = An identifier attached to images within a repository (e.g., 14.04 or stable)

ImageID = Unique identifier hash generated for each image

The official documentation declares:

An image name is made up of slash-separated name components, optionally prefixed by a registry hostname.

It means that you can encode registry hostname and a bunch of slash-separated “name components” into the name of your image. Honestly, this is quite convoluted, but such is life.

The fundamental format is:

```
<name>:<tag>
```

But in practice it is:

```
<registry>/<name-component-1>/
<name-component-2>:<tag>
```

It may vary per platform. For Google Cloud Platform (GCP) the convention is:

```
<registry>/<project-id>/
<repository-name>/<image>@<image-digest>:<tag>
```

It is up to you to figure out the correct naming scheme for your case.

Note: The `latest` tag will be used if you pull an image without any tags. Never use this `latest` tag in production. Always use a tag with a unique version or hash instead since someone inevitably will update the “latest” image and break your build. What is the latest today is no longer the latest tomorrow! The astronaut doesn't care about the latest bells and whistles. They just want a spacesuit that fits them and keeps them alive.

Docker images and secrets

Just like it is a terrible practice to push secrets into a git repository, you shouldn’t bake them into your Docker images either!

Images are put into repositories and passed around carelessly. The correct assumption is that whatever goes into an image may be public at some point. It is not a place for your username, password, API token, key code, TLS certificates, or any other sensitive data.

There are two scenarios with secrets and docker images:
1. You need a secret at build-time
2. You need a secret at runtime

Neither case should be solved by baking things permanently into the image. Let’s look at how to do it differently.

Build-time secrets

If you need something private - say a private GitHub repository - to be pulled into the image at build time, you need to make sure that the SSH keys you are using do not leak into the image.

Do NOT use COPY instruction to move keys or passwords into the image! Even if you remove them afterward, they will still leave a trace!

Quick googling will give you many different options to solve this problem, like using multi-stage builds, but the best and most modern way is to use BuildKit. BuildKit ships with Docker but needs to be enabled for builds by setting up the environment variable `DOCKER_BUILDKIT`.

For example:

```
DOCKER_BUILDKIT=1 docker build .
```

BuildKit offers a mechanism to make secret files safely available for the build process.

Let's first create `secret.txt` with the contents:

```
TOP SECRET ASTRONAUT PASSWORD
```

Then create a new `Dockerfile`:

```
FROM alpine
RUN --mount=type=secret,
id=mypass cat /run/secrets/mypass
```

`The --mount=type=secret,id=mypass` is informing Docker that for this specific command, we need access to a secret called `mypass` (the contents of which we'll tell the Docker build about in the next step). Docker will make this happen by temporarily mounting a file `/run/secrets/mypass`.

The `cat /run/secrets/mypass` is the actual instruction, where `cat` is a Linux command to output the contents of a file into the terminal. We call it to validate that our secret was indeed available.

Let's build the image, adding `--secret` to inform `docker build` about where to find this secret:

```
DOCKER_BUILDKIT=1 docker build . -t myimage --secret
id=mypass,src=secret.txt
```

Everything worked, but we didn't see the contents of secret.txt printed out in our terminal as we expected. The reason is that BuildKit doesn't log every success by default.

Let's build the image using additional parameters. We add `BUILDKIT_PROGRESS=plain` to get more verbose logging and `--no-cache` to make sure caching doesn't ruin it:

```
DOCKER_BUILDKIT=1 BUILDKIT_PROGRESS=plain docker
build . --no-cache --secret id=mypass,src=secret.txt
```

Among all the logs printed out, you should find this part:

```
#5 [2/2] RUN --mount=type=secret,id=mypass cat /run/
secrets/mypass
#5 sha256:7fd248d616c172325af799b6570d2522d3923638ca
41181fab438c29d0aea143
#5 0.248 TOP SECRET ASTRONAUT PASSWORD
```

It is proof that the build step had access to `secret.txt`.

With this approach, you can now safely mount secrets to the build process without worrying about leaking keys or passwords to the resulting image.

Runtime secrets

If you need a secret - say database credentials - when your container is running in production, you should use environment variables to pass secrets into the container.

Never bake any secrets straight into the image at build time!

```
docker run --env MYLOGIN=johndoe --env
MYPASSWORD=sdf4otwe3789
```

These will be accessible in Python like:

```
os.environ.get('MYLOGIN')
os.environ.get('MYPASSWORD')
```

Tip: You can also fetch the secrets from a secret store like Hashicorp Vault!

GPU support

Docker with GPUs can be tricky. Building an image from scratch is beyond the scope of this article, but there are five prerequisites for a modern GPU (NVIDIA) container.

Image:

- CUDA/cuDNN libraries
- GPU versions of your framework like Tensorflow (when needed)

Host machine:

- GPU drivers
- NVidia Docker Toolkit
- Docker `run` executed with `--gpus all`

The best approach is finding a base image with most prerequisites already baked in. Frameworks like Tensorflow usually offer images like `tensorflow/tensorflow:latest-gpu`, which are a good starting point.

When troubleshooting, you can first try to test your host machine:

```
nvidia-smi
```

Then run the same command inside the container:

```
docker run --gpus all tensorflow/tensorflow:latest-gpu
nvidia-smi
```

You should get something like this for both commands:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0  On |                  N/A |
| 21%   56C    P5    19W / 163W |    358MiB /  4096MiB |      1%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A       928      G   /usr/lib/xorg/Xorg                 32MiB |
|    0   N/A  N/A      1677      G   /usr/lib/xorg/Xorg                 69MiB |
|    0   N/A  N/A      1805      G   /usr/bin/gnome-shell              101MiB |
|    0   N/A  N/A      3022      G   ...90696857644243528 1,131072     138MiB |
+-----------------------------------------------------------------------------+
```

If you get an error from either, you'll have an idea whether the problem lies inside or outside the container.

It's also a good idea to test your frameworks. For example Tensorflow:

```
docker run --gpus all -it --rm tensorflow/
tensorflow:latest-gpu python -c "import
tensorflow as tf;print(tf.reduce_sum
(tf.random.normal([1000, 1000])))"
```

The output may be verbose and have some warnings, but it should end with something like:

```
Created device /job:localhost/replica:0/task:0/
device:GPU:0 with 3006 MB memory:  -> device:
0, name: NVIDIA GeForce GTX 970, pci bus id:
0000:01:00.0, compute capability: 5.2
tf.Tensor(-237.35098, shape=(), dtype=float32)
```

Docker containers vs. Python virtual environments

Our last chapter about Python dependency management talked about Python virtual environments and how they create a safety bubble between different Python projects in your local development environment. Docker containers solve a similar problem but on a different layer.

While a Python virtual environment creates the isolation layer between all Python-related things, a Docker container achieves this for the entire software stack. The use-cases for Python virtual environments and

Docker containers are different. As a rule of thumb, virtual environments are enough for developing things on your local machine while Docker containers are built for running production jobs in the cloud.

To put it another way, for local development virtual environments are like wearing sunscreen on the beach, while Docker containers are like wearing a spacesuit – usually uncomfortable and mostly impractical.

Valohai

Build in Production

The infrastructure stack for machine learning development

You don't need to bridge the gap between ML and Ops when every model and pipeline runs on Valohai. Our platform brings powerful cloud infrastructure to every data scientist and ML engineer and unlocks their full creativity in building machine learning solutions.

AI TRAILBLAZER

PRELIGENS

Preligens recorded that their average model development time has went down 30-40% during the first few months.

"Building a barebones infrastructure layer for our use case would have taken months, and that would just have been the beginning. The challenge is that with a self-managed platform, you need to build and maintain every new feature, while with Valohai, they come included."

Renaud Allioux
Co-Founder & CTO | Preligens

DEVELOPER-FIRST MLOPS

Ship faster and more frequently

Experimentation and productization on the same platform makes shipping easy.

› SCALE YOUR OUTPUT

Empower the whole team

Valohai democratizes the access to experiments, models, metrics and compute.

› SCALE YOUR TEAM

Build without arbitrary limitations

Every production system is different which is why Valohai can integrate with any system.

› SCALE YOUR AMBITION

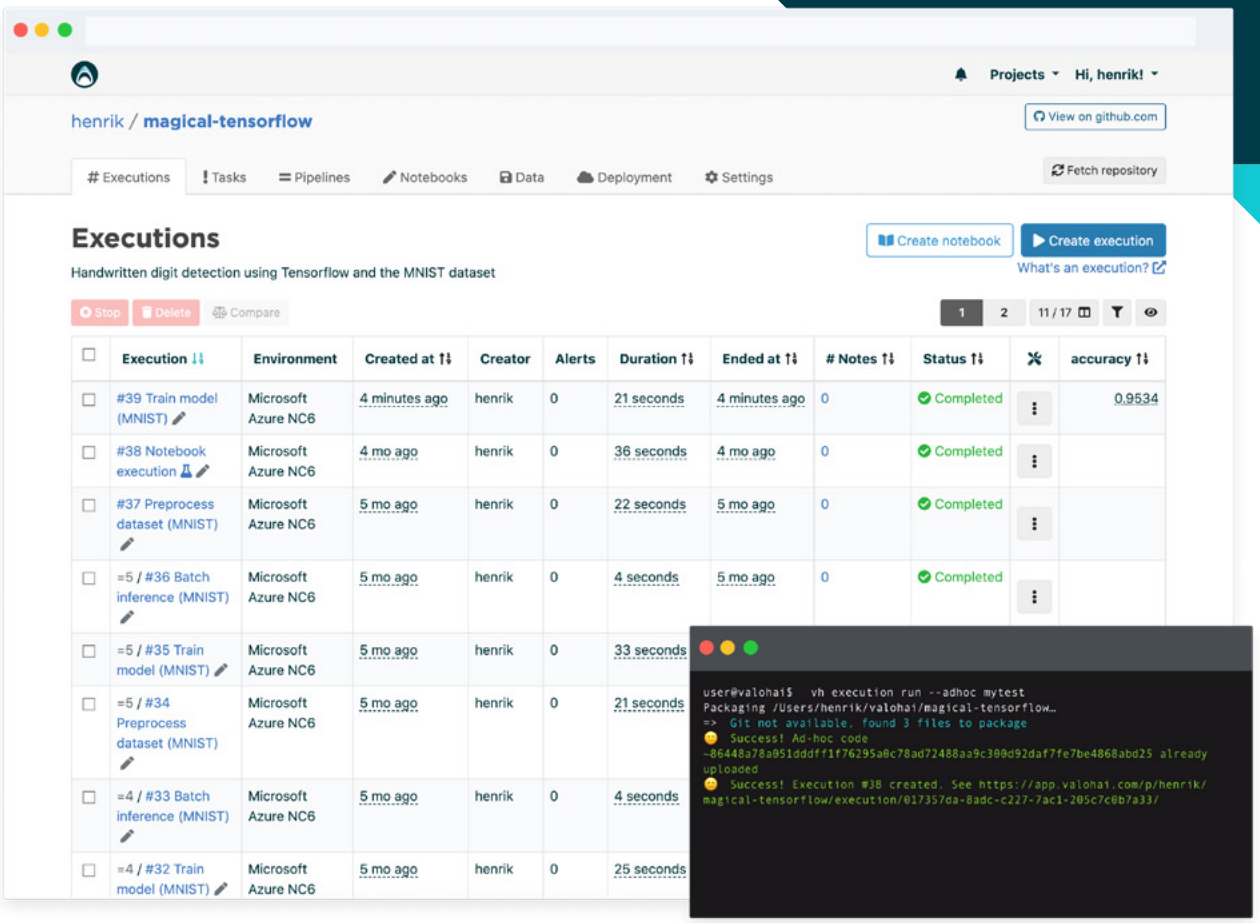
WHY CUSTOMERS USE VALOHAI

Cut down training time from weeks to minutes and get to market faster

Spend less time on infrastructure by automating version control & workflows

Reduce compliance risk through a standardized way of working

Increase team collaboration through a transparent toolchain and audit trail



Machine Orchestration

- Automatically launch VMs, set up the environment, run your code and shut down when done
- Cut down manual DevOps work from 80% to 20%
- Get your product to market faster

Automatic Version Control

- Automatically track every experiment from code & hyper parameters to training data & costs
- Go from tracking 10% to 100% of your experiment
- Ensures compliance, reproducibility and collaboration

Pipeline Management

- Build automated model pipelines that re-train themselves on demand
- Cut down manual work by adding model creation to your CI process
- Seamless integration between teams from feature extraction to inference

---

It's no secret; trailblazers use Valohai to build and deploy machine learning solutions that are changing the world.

KONUX · syngenta · JFrog · maytronics · neonode · sama · lifebit · OMHU · GREENSTEAM · PRELIGENS · yousician