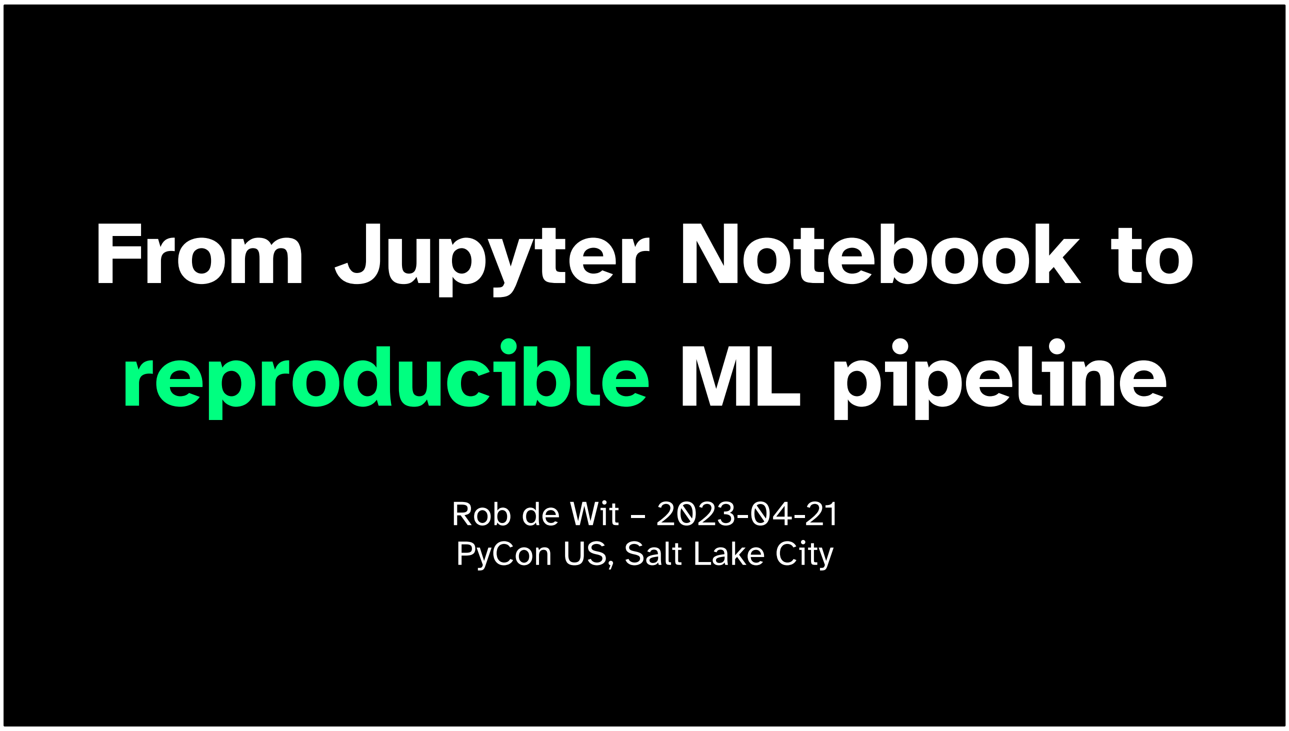


From Jupyter Notebook to reproducible ML pipeline

Rob de Wit – 2023-04-21
PyCon US, Salt Lake City

Hello and thanks for coming

Today I'll be talking about converting a Jupyter notebook into a reproducible ML pipeline with DVC

Rob de Wit

Utrecht, the Netherlands

Developer Advocate*

Water-Pokémon trainer

[Personal introduction]

A while back I spoke at PyData in Eindhoven about becoming a Pokémon master. Having successfully completed my goals of becoming the very best like no one ever was with the help of machine learning, I wanted a new pet project

I don’t need to tell any of you about the hype that text2image models have created over the past year or so, so let’s dive right in. My goal was to generate new Pokémon with stable diffusion.

Goal

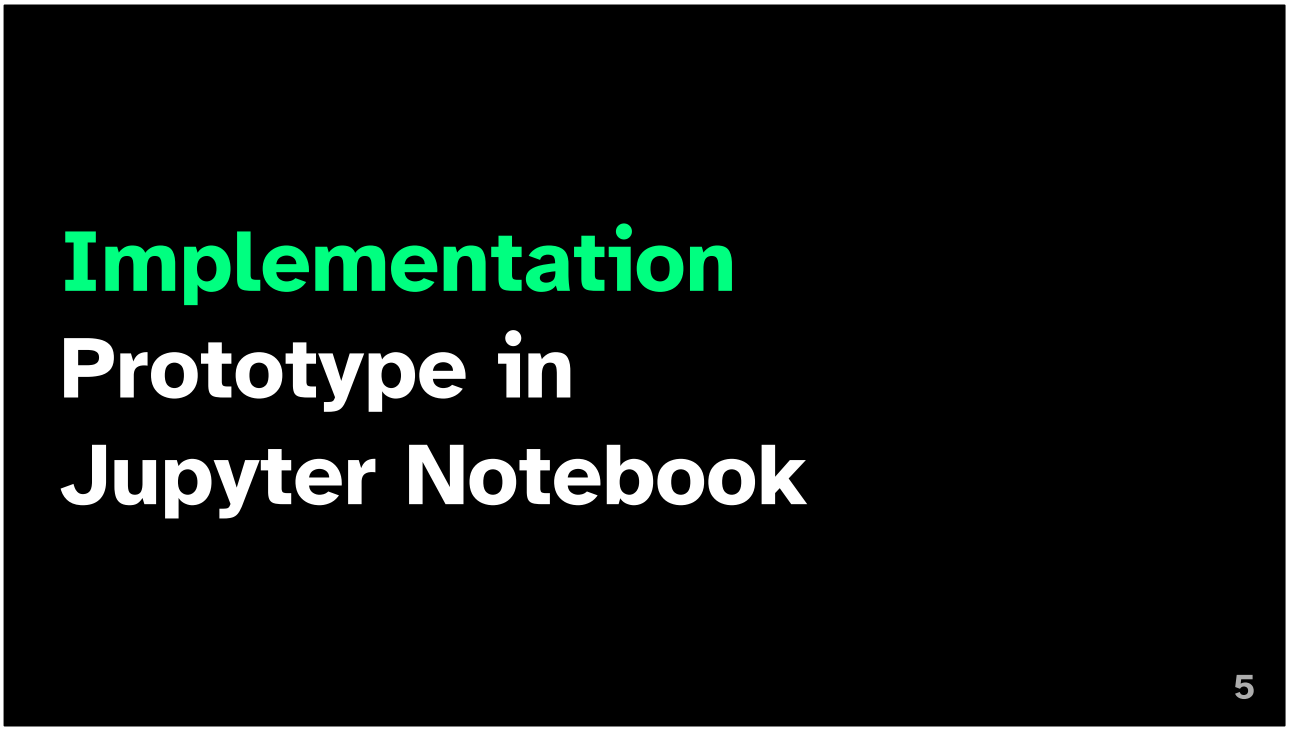
Generate Pokémon with Stable Diffusion

My goal for the project was to generate Pokémon concepts with SD

Approach
Use Stable Diffusion, train LoRA on Pokémon

The current “meta” in fine-tuning SD models is LoRA, or Low-rank adaption
They apply small changes to the most critical part of SD models: the cross-attention layers

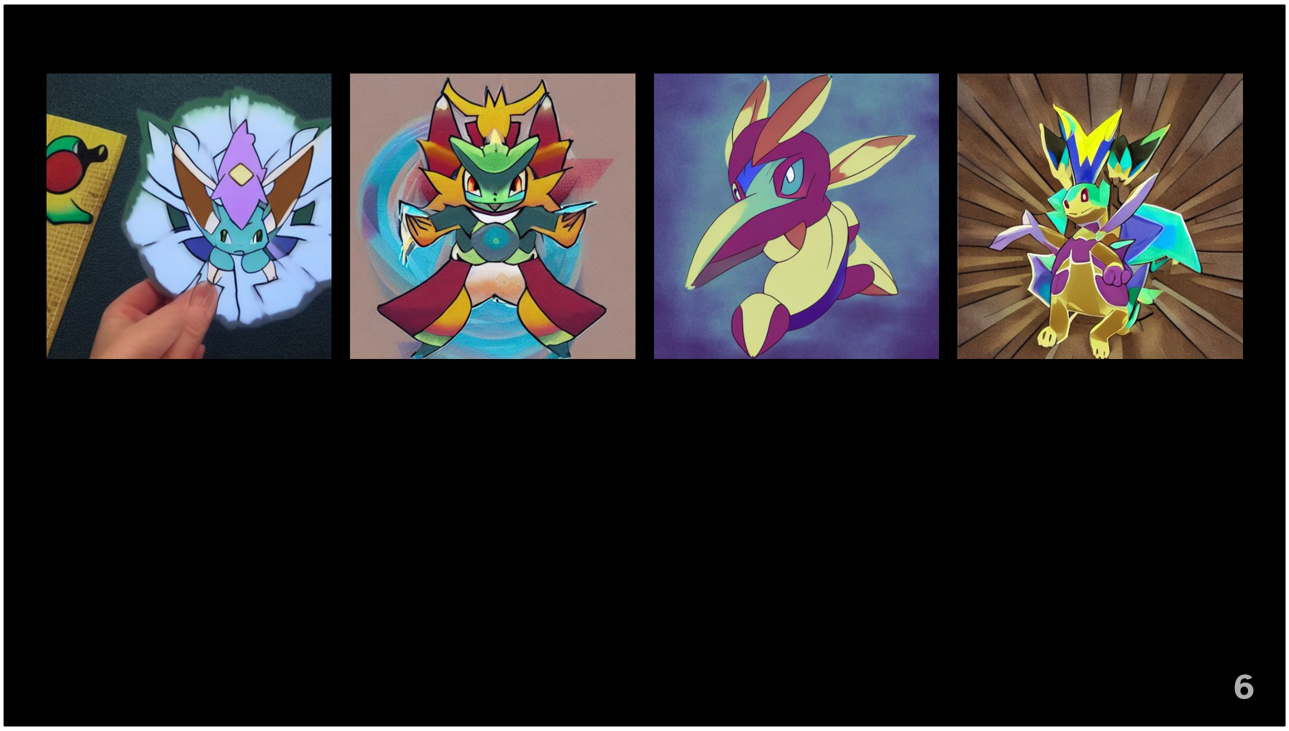
So we can take a dataset of Pokémon images, train our LoRA on those images, and use SD as a base model.
The great thing: training LoRA is really quick (and thus cheap)



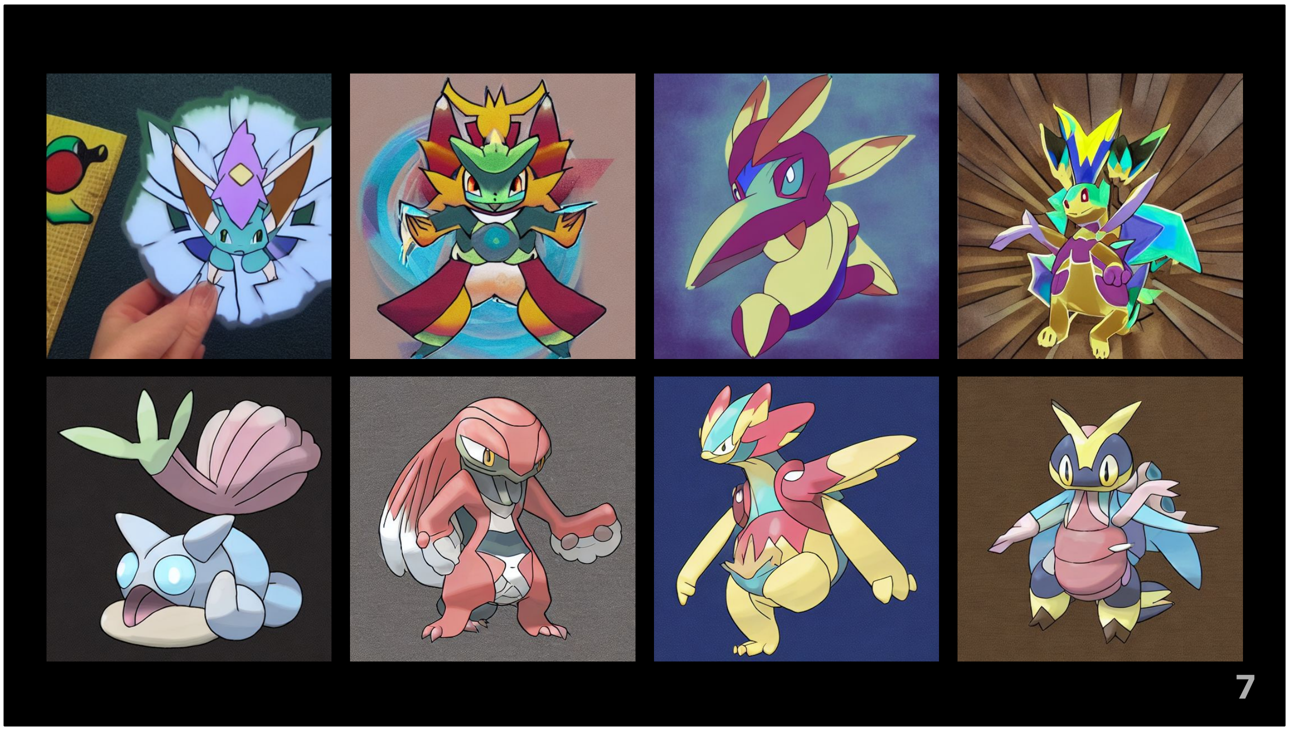
Implementation

Prototype in Jupyter Notebook

Like any data scientist, I start out in a Jupyter Notebook
[Show notebook]



First row is SD1.5 base
Second row is with custom LoRA



First row is SD1.5 base
Second row is with custom LoRA

(Cherry picked examples)

Experimentation
Find the best Pokémon

Now that I had a working prototype, it was time to do some experimenting
This is something that we do in every machine learning project:

We keep tweaking and fiddling to find the best Pokémon… Or predictions… Or generally: model

Experimentation
Find the best predictions

Now that I had a working prototype, it was time to do some experimenting
This is something that we do in every machine learning project:

We keep tweaking and fiddling to find the best Pokémon… Or predictions… Or generally: model

Experimentation
Find the best model

Now that I had a working prototype, it was time to do some experimenting
This is something that we do in every machine learning project:

We keep tweaking and fiddling to find the best Pokémon… Or predictions… Or generally: model

IRL
We experiment all the time

In real life, we rarely get to bring a model into production and just keep it there.
Changes to:

- Business requirements
- Data drift
- Port model to a different context
- Just keep optimising

But experimentation becomes real messy if we don't keep track properly of our experiments. We change a hyperparameter here, load a new dataset there, and keep tweaking to get better results

Just like in "real" science, we can't do data science without properly tracking our experiments and results
Show of hands: who here has been guilty of tracking model performance on a physical notepad? I sure have

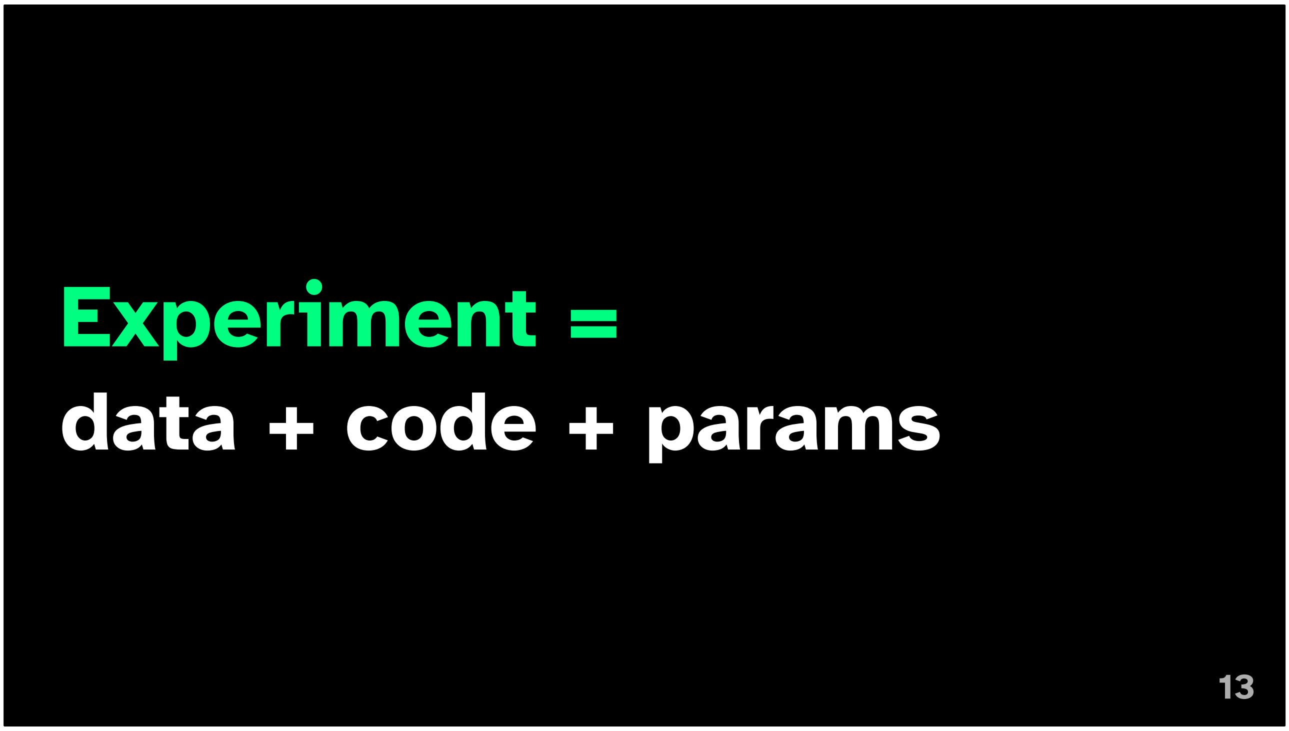
We need a more structured approach to be able to compare the results of our ML experiments.

Reproducibility
Track all components

And just keeping track of models and metrics isn’t enough. In many contexts we need to be able to explain which data and considerations went into a model.
[Anecdote about the guy in Berlin whose banking team spent two weeks reconstructing a model from four years ago]

To do proper experimentation, we need reproducibility.
We should be able to go back and look at our records to see which changes where made.

So what do we need for reproducibiltity?



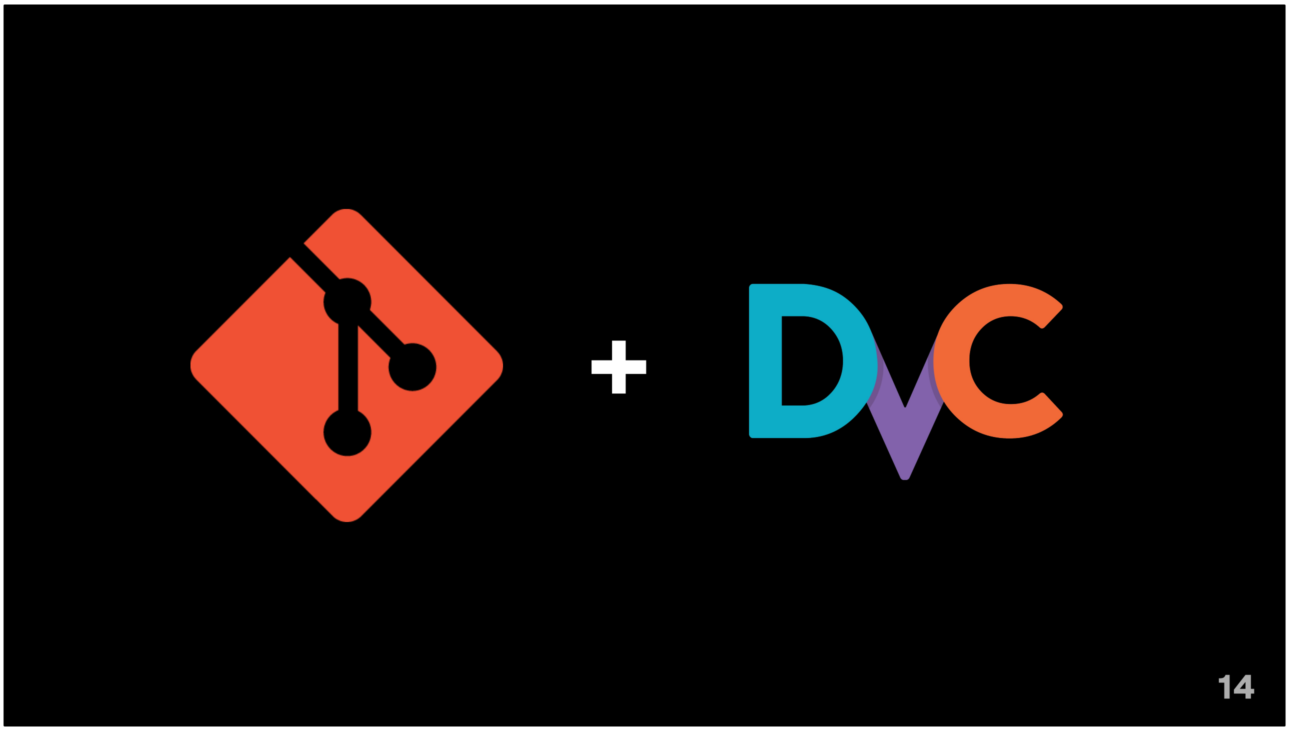
Experiment =
data + code + params

13

When we know the data, code, and parameters that went into an experiment, we should always be able to reproduce its results (i.e. the model)

While Jupyter Notebooks are great for initial prototyping, they really aren’t great for reproducibility.
Versioning them is difficult and you can jump around cells arbitrarily without keeping track of changes

Luckily we can look toward best practices from computer science for solutions.
Reproducibility for code is a solved problem: Git

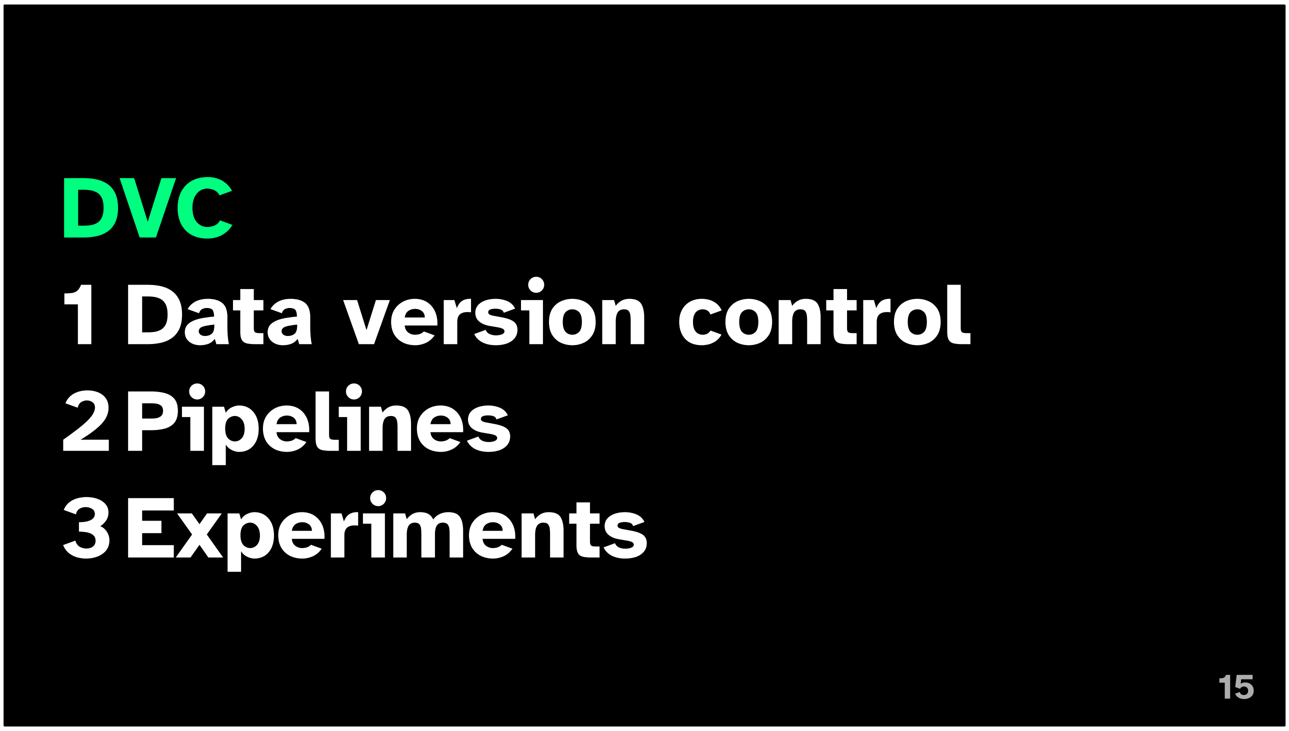


Git is great to keep a history of changes to files
But it has its limitations: it’s geared towards text/code and doesn’t work well for large files (e.g. data and artefacts)
Github file size is 50MB, for example

So can we have Git for data? Yes we can, that’s DVC!

Sidenote: git-lfs is an alternative, but DVC is geared towards ML as we’ll see shortly

So what does DVC do for you?



DVC

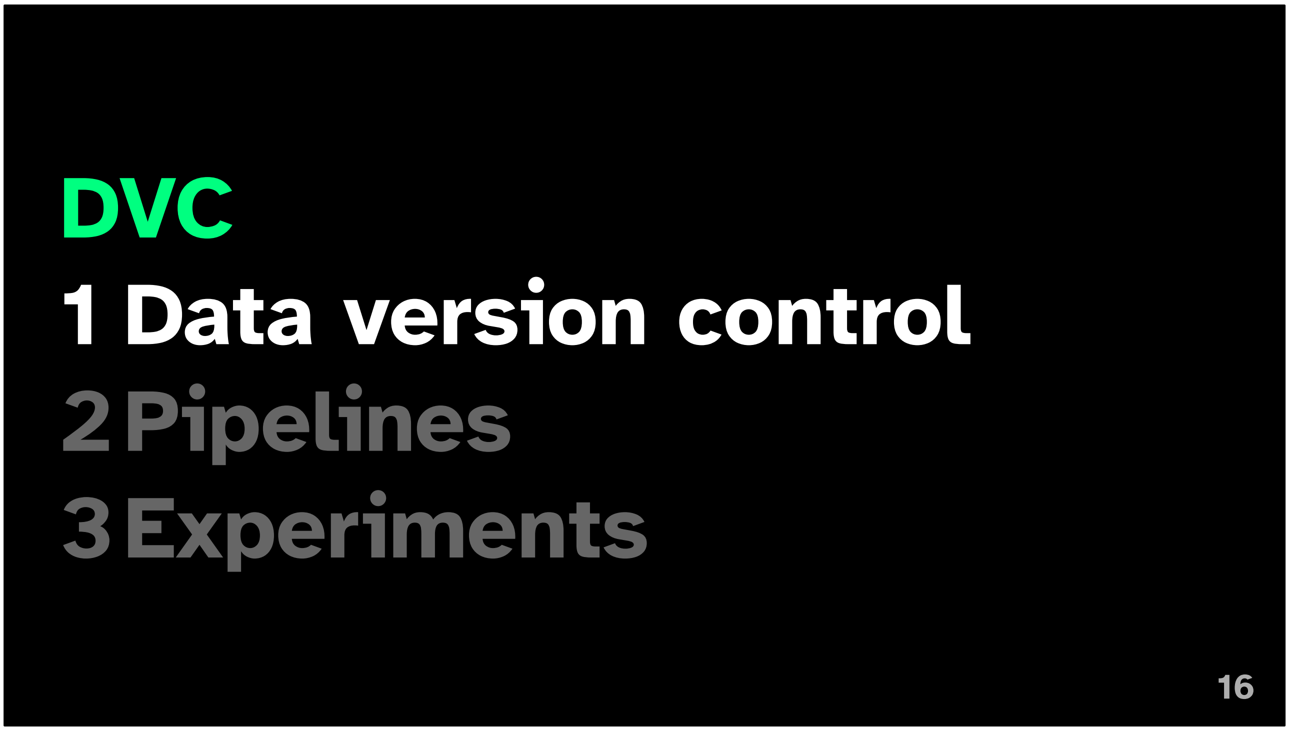
1 Data version control

2 Pipelines

3 Experiments

DVC boils down to three main features:

- Its namesake; data version control
- Pipelines
- And experiment management



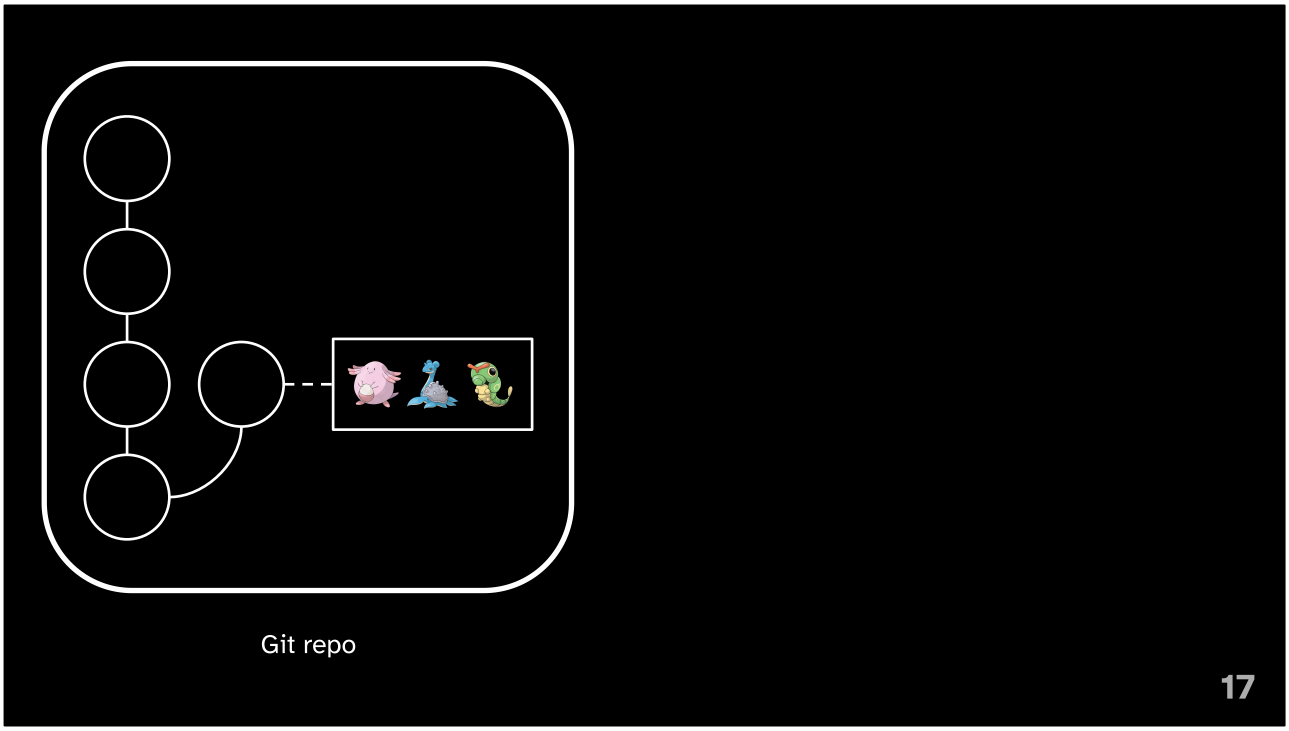
DVC

1 Data version control

2 Pipelines

3 Experiments

Let’s look at data version control first and see how DVC handles this



Here we see a Git commit tree, where a commit contains a bunch of Pokémon artwork

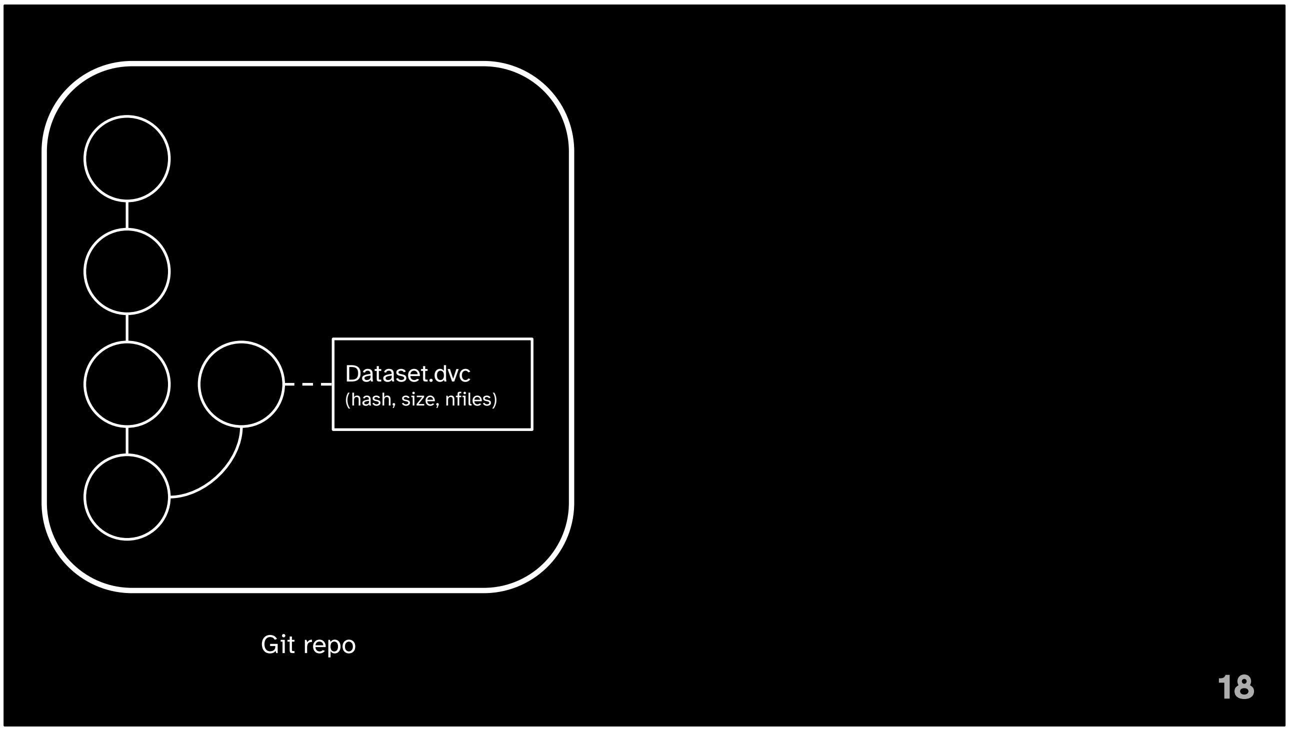
If we use DVC to track those images instead of Git, DVC creates a metadata file to replace the physical images.
The metadata file contains a bunch of info, most importantly the hash

The physical files are then transferred to the DVC cache. DVC uses the file hashes to link the files in the cache to the right location in the Git repo.
It does so using symlinks.

Now, if we make a new commit with a changed dataset, DVC can point to updated locations in its cache.
Here we remove Caterpie from the dataset and add Ampharos. The other two Pokémon are unaffected

The great thing here is that DVC doesn't duplicate the images that are unchanged. It just saves the changes

The DVC cache lives on your local system; in my case my laptop. We can also mirror it to a remote storage (such as an S3 bucket, an SFTP server, or even a Google Drive directory).



Here we see a Git commit tree, where a commit contains a bunch of Pokémon artwork

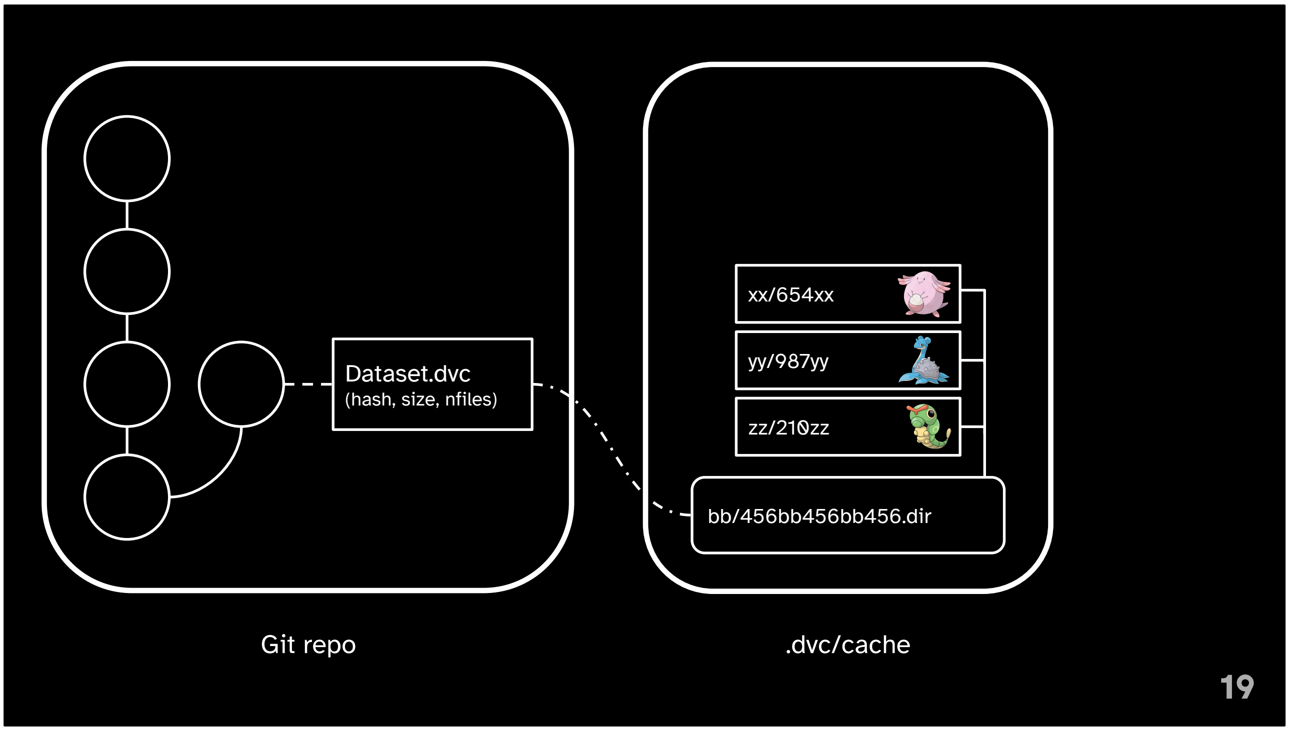
If we use DVC to track those images instead of Git, DVC creates a metadata file to replace the physical images.
The metadata file contains a bunch of info, most importantly the hash

The physical files are then transferred to the DVC cache. DVC uses the file hashes to link the files in the cache to the right location in the Git repo.
It does so using symlinks.

Now, if we make a new commit with a changed dataset, DVC can point to updated locations in its cache.
Here we remove Caterpie from the dataset and add Ampharos. The other two Pokémon are unaffected

The great thing here is that DVC doesn’t duplicate the images that are unchanged. It just saves the changes

The DVC cache lives on your local system; in my case my laptop. We can also mirror it to a remote storage (such as an S3 bucket, an SFTP server, or even a Google Drive directory).



Here we see a Git commit tree, where a commit contains a bunch of Pokémon artwork

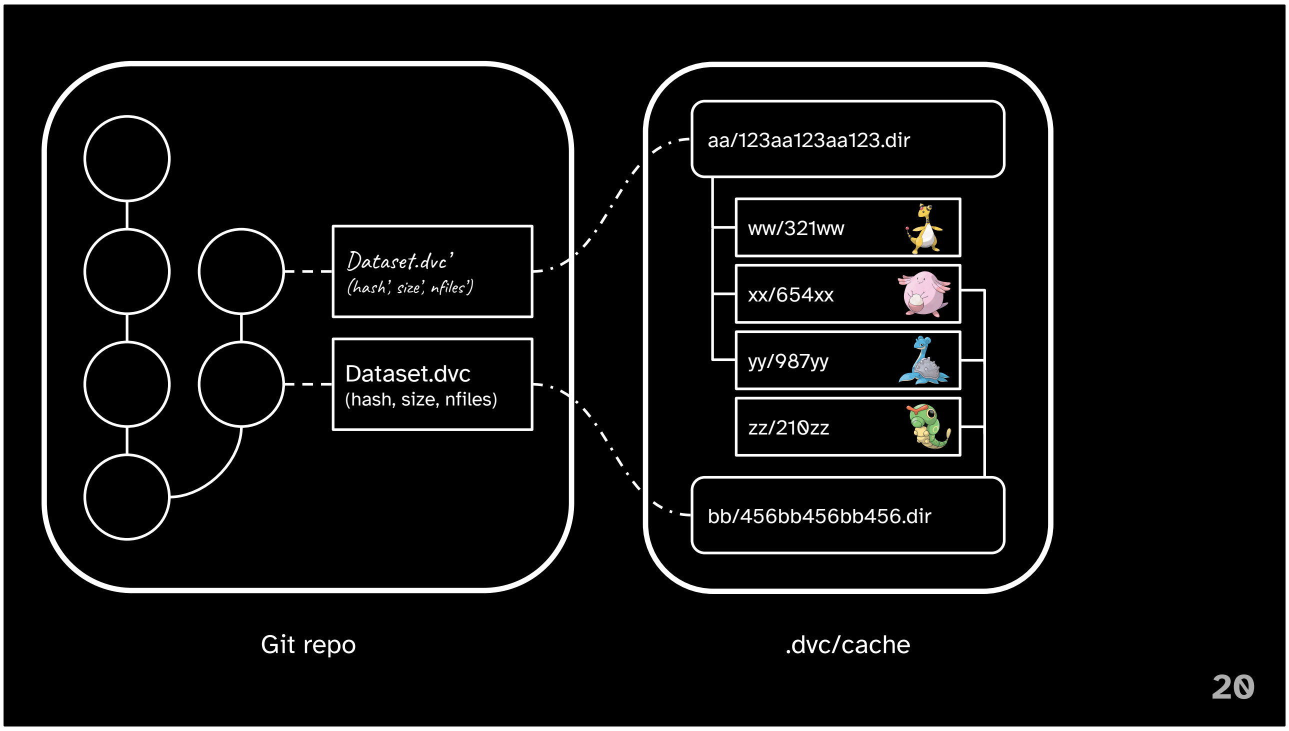
If we use DVC to track those images instead of Git, DVC creates a metadata file to replace the physical images.
The metadata file contains a bunch of info, most importantly the hash

The physical files are then transferred to the DVC cache. DVC uses the file hashes to link the files in the cache to the right location in the Git repo.
It does so using symlinks.

Now, if we make a new commit with a changed dataset, DVC can point to updated locations in its cache.
Here we remove Caterpie from the dataset and add Ampharos. The other two Pokémon are unaffected

The great thing here is that DVC doesn't duplicate the images that are unchanged. It just saves the changes

The DVC cache lives on your local system; in my case my laptop. We can also mirror it to a remote storage (such as an S3 bucket, an SFTP server, or even a Google Drive directory).



Here we see a Git commit tree, where a commit contains a bunch of Pokémon artwork

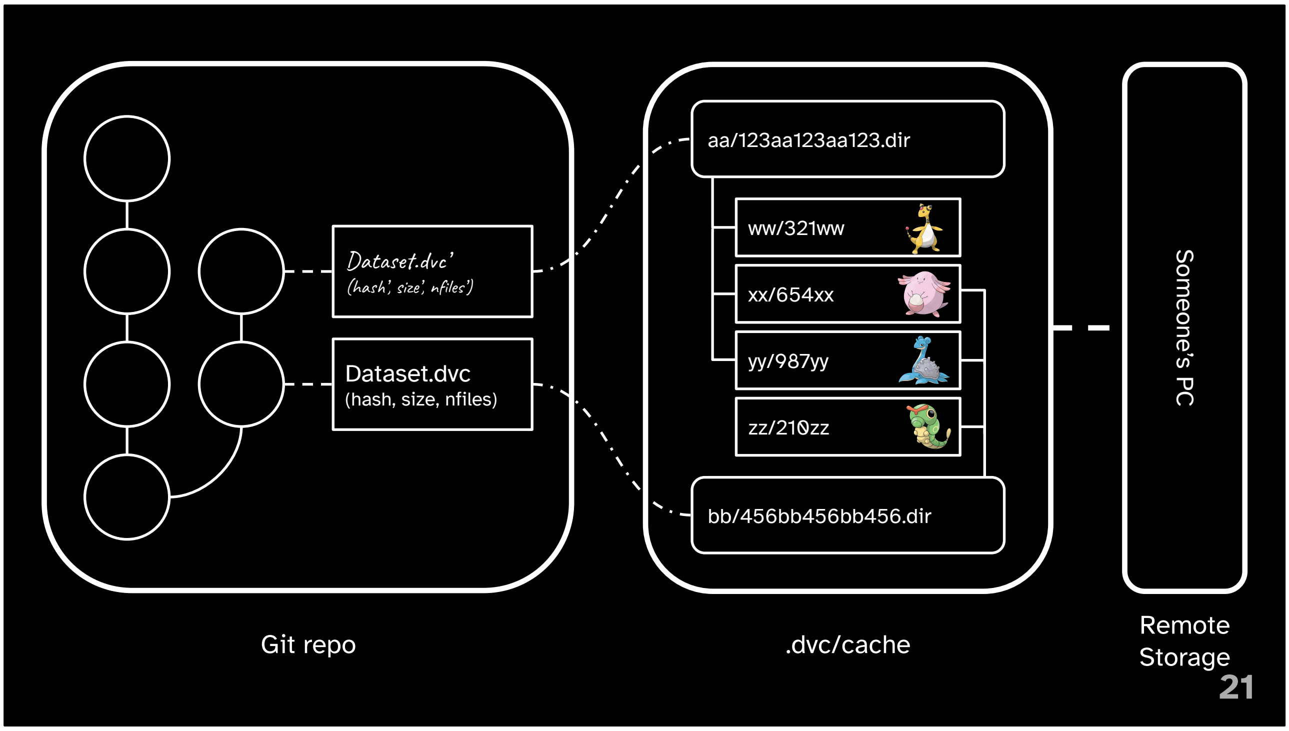
If we use DVC to track those images instead of Git, DVC creates a metadata file to replace the physical images.
The metadata file contains a bunch of info, most importantly the hash

The physical files are then transferred to the DVC cache. DVC uses the file hashes to link the files in the cache to the right location in the Git repo.
It does so using symlinks.

Now, if we make a new commit with a changed dataset, DVC can point to updated locations in its cache.
Here we remove Caterpie from the dataset and add Ampharos. The other two Pokémon are unaffected

The great thing here is that DVC doesn't duplicate the images that are unchanged. It just saves the changes

The DVC cache lives on your local system; in my case my laptop. We can also mirror it to a remote storage (such as an S3 bucket, an SFTP server, or even a Google Drive directory).



Here we see a Git commit tree, where a commit contains a bunch of Pokémon artwork

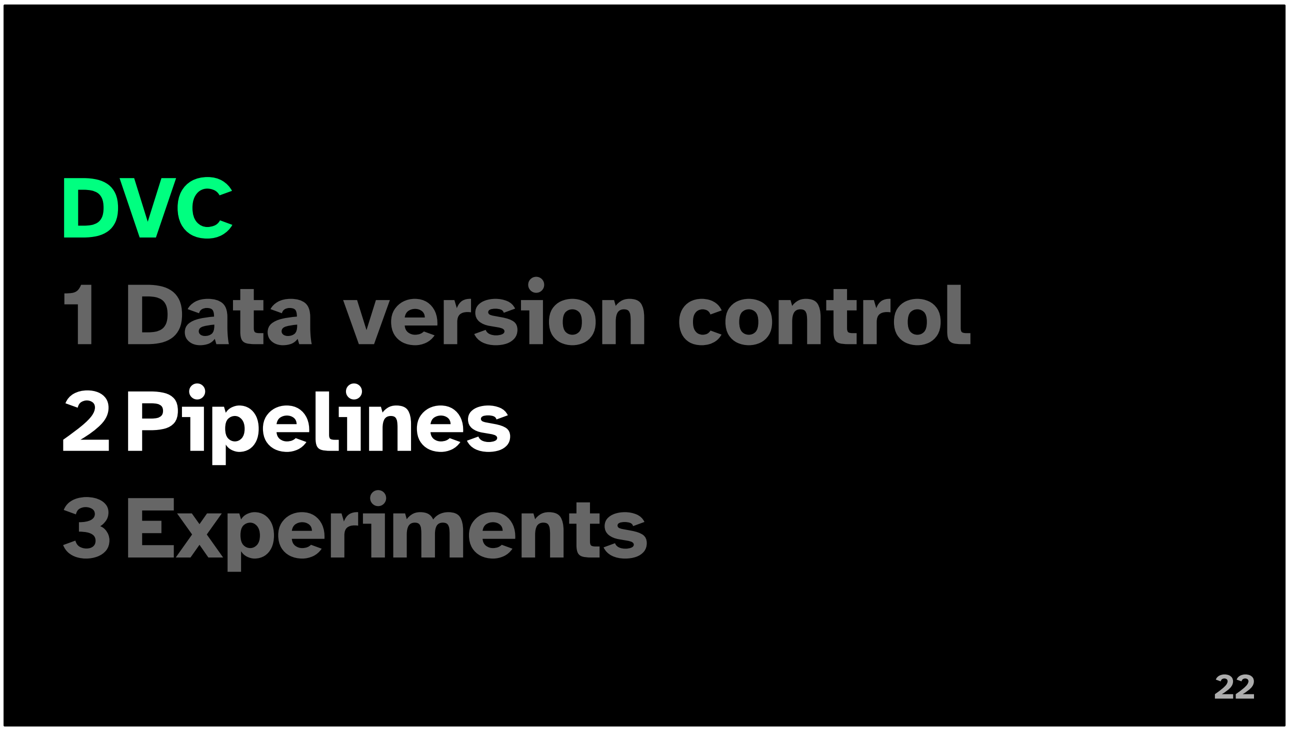
If we use DVC to track those images instead of Git, DVC creates a metadata file to replace the physical images.
The metadata file contains a bunch of info, most importantly the hash

The physical files are then transferred to the DVC cache. DVC uses the file hashes to link the files in the cache to the right location in the Git repo.
It does so using symlinks.

Now, if we make a new commit with a changed dataset, DVC can point to updated locations in its cache.
Here we remove Caterpie from the dataset and add Ampharos. The other two Pokémon are unaffected

The great thing here is that DVC doesn't duplicate the images that are unchanged. It just saves the changes

The DVC cache lives on your local system; in my case my laptop. We can also mirror it to a remote storage (such as an S3 bucket, an SFTP server, or even a Google Drive directory).



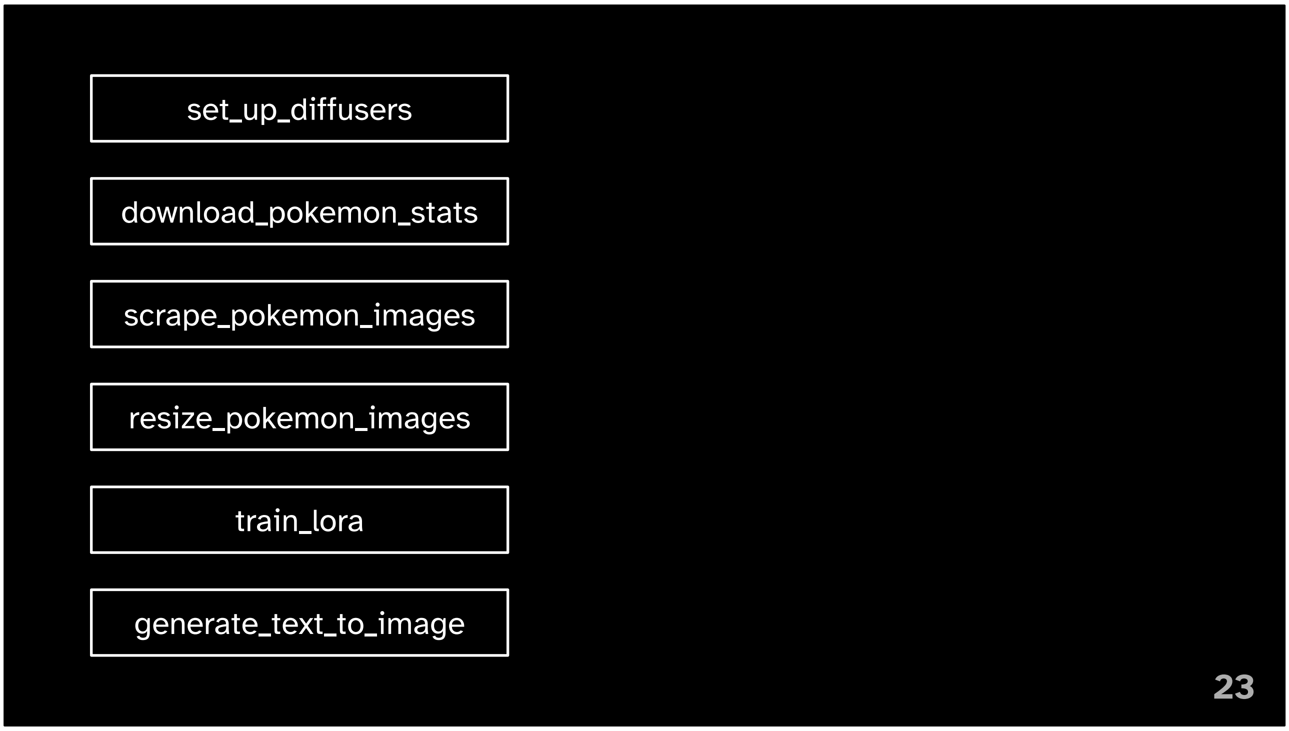
DVC

1 Data version control

2 Pipelines

3 Experiments

Now let’s take a look at Pipelines in DVC



set_up_diffusers

download_pokemon_stats

scrape_pokemon_images

resize_pokemon_images

train_lora

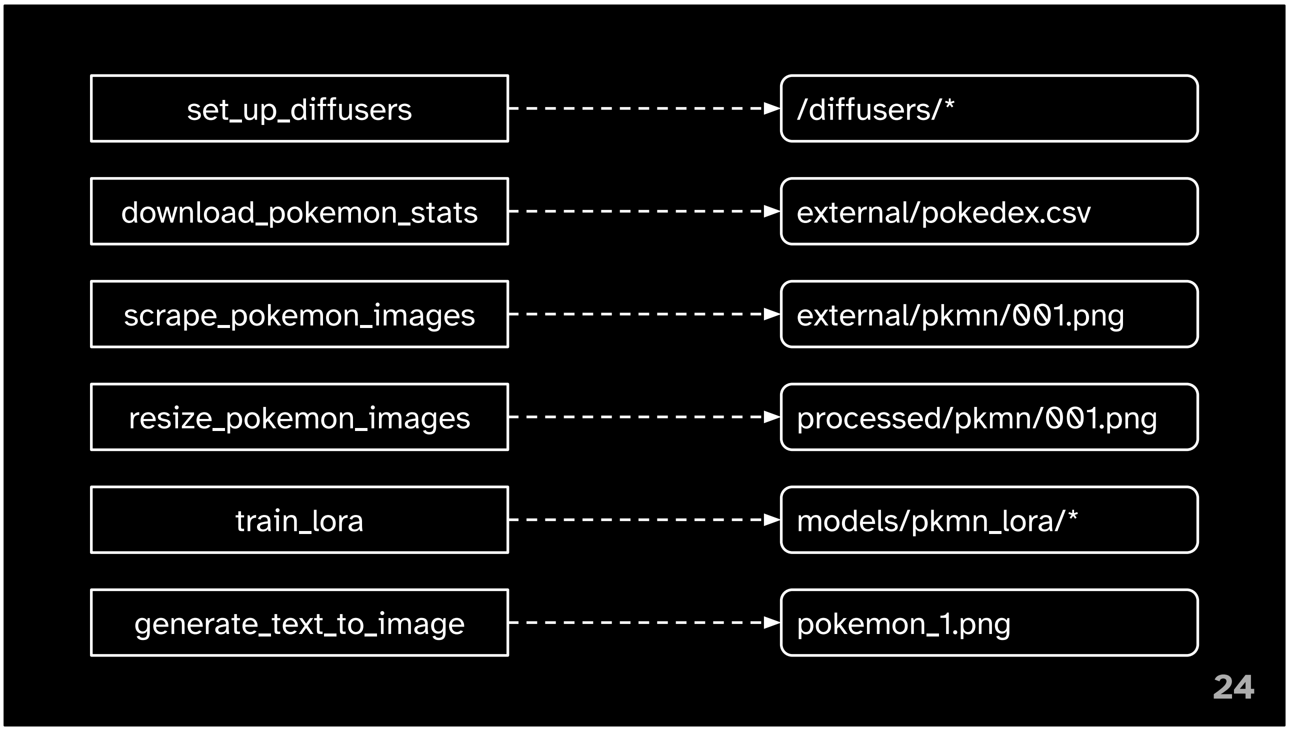
generate_text_to_image

This is broadly the set-up we saw earlier in the Notebook. I have split up the process into cohesive steps, or stages.

Each stage produces outputs, such as the image dataset for the scraping stage.

DVC allows you to specify these outputs as dependencies for downstream stages

So that same image dataset is a dependency for the resize stage, and DVC won't trigger that stage until the previous one has completed.

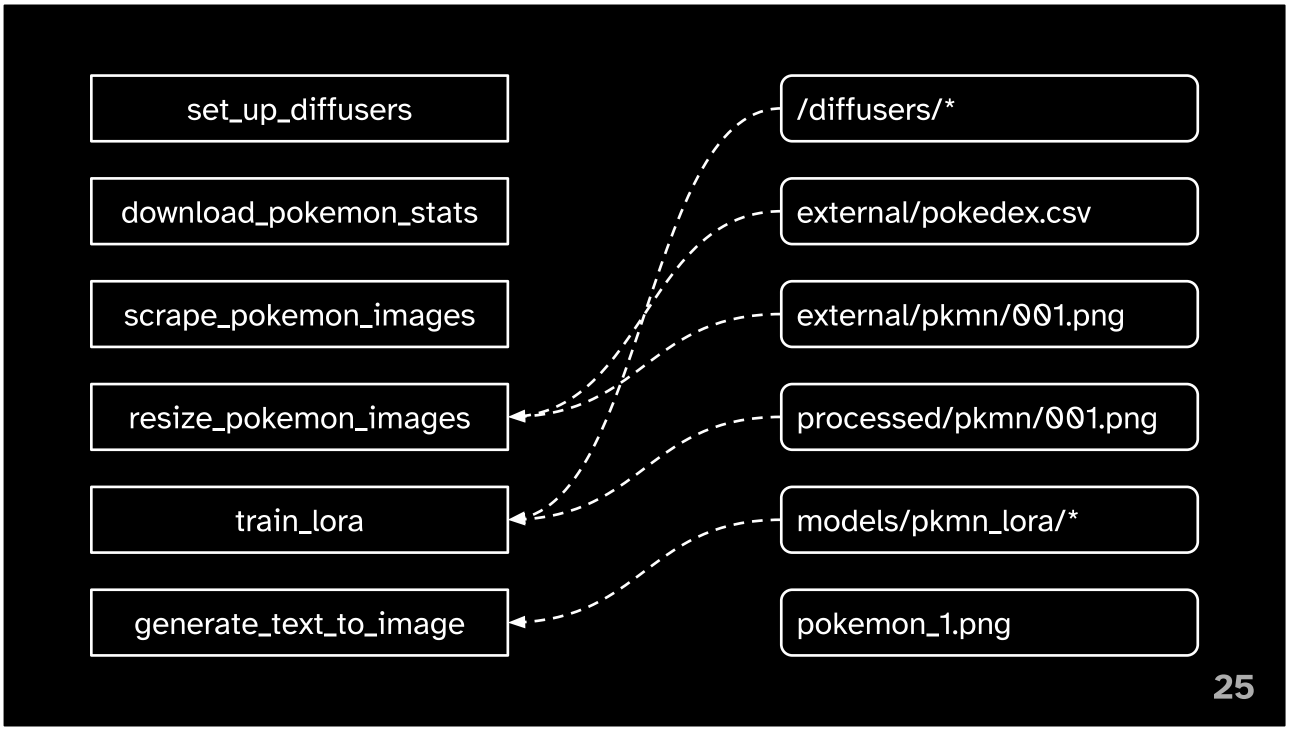


This is broadly the set-up we saw earlier in the Notebook. I have split up the process into cohesive steps, or stages.

Each stage produces outputs, such as the image dataset for the scraping stage.

DVC allows you to specify these outputs as dependencies for downstream stages

So that same image dataset is a dependency for the resize stage, and DVC won't trigger that stage until the previous one has completed.

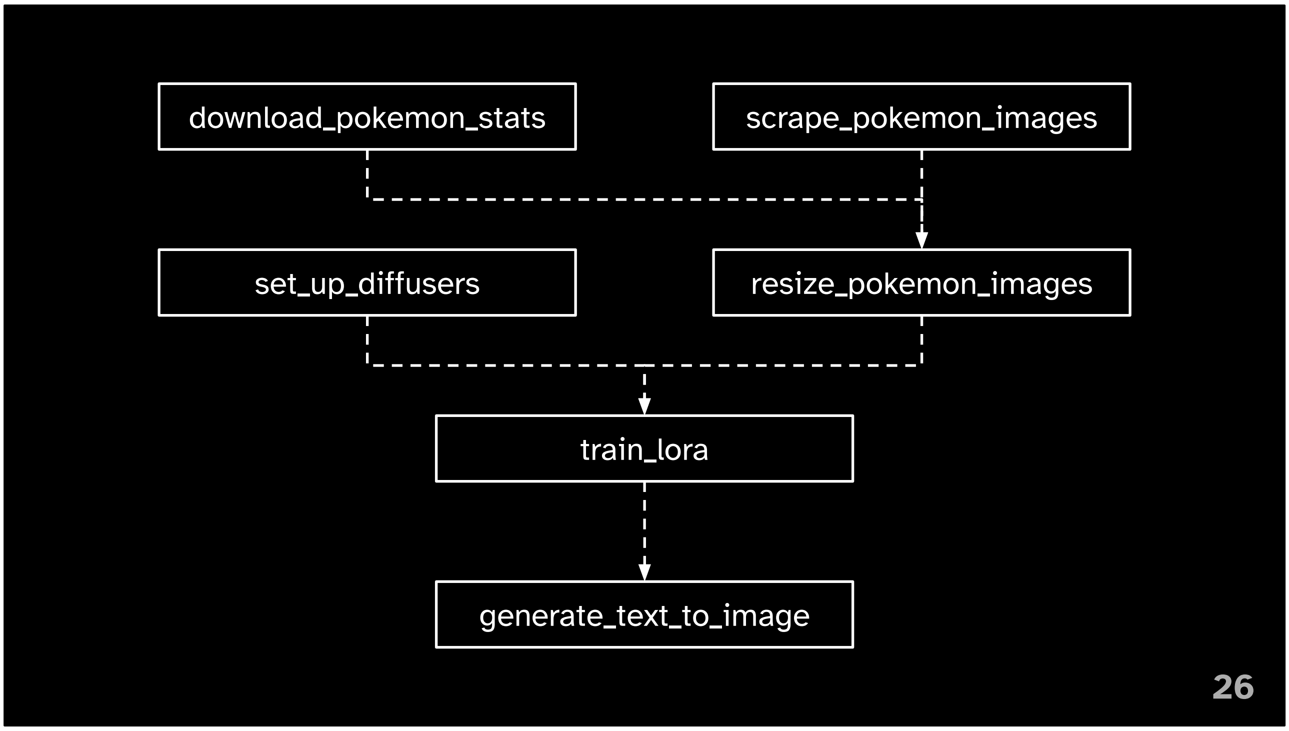


This is broadly the set-up we saw earlier in the Notebook. I have split up the process into cohesive steps, or stages.

Each stage produces outputs, such as the image dataset for the scraping stage.

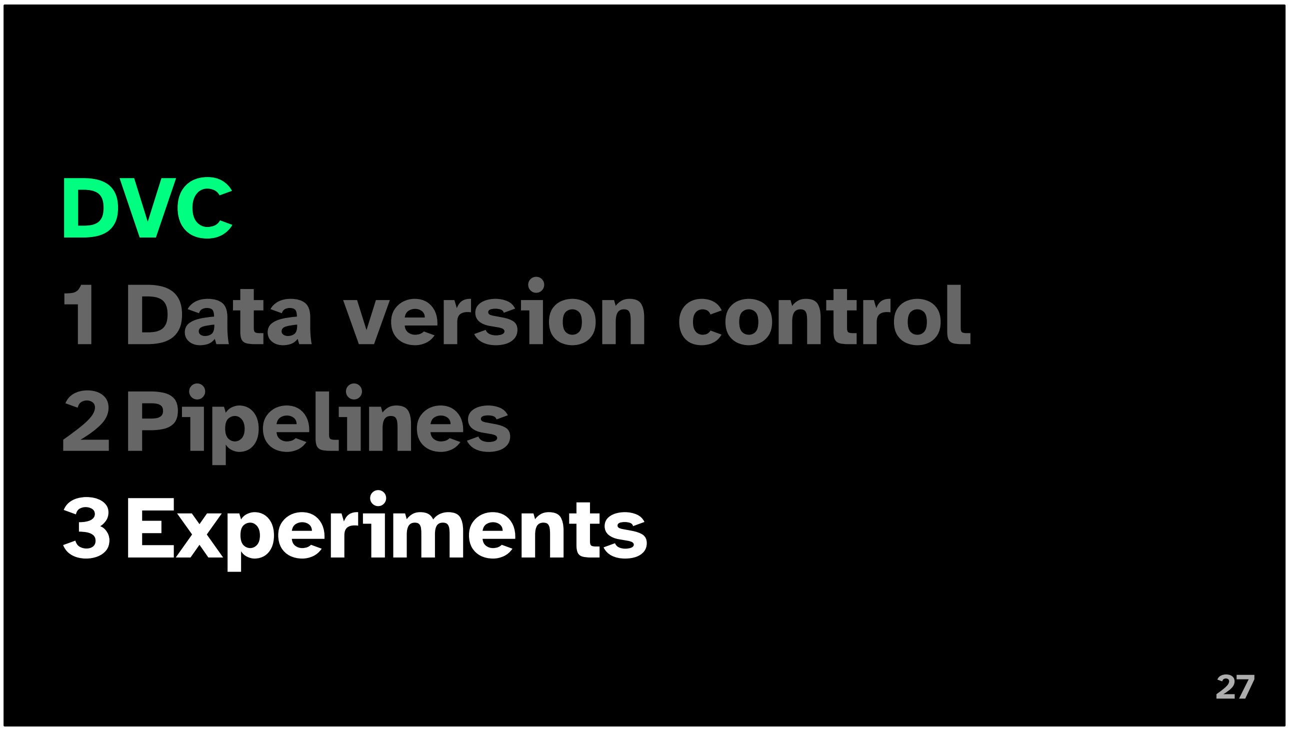
DVC allows you to specify these outputs as dependencies for downstream stages

So that same image dataset is a dependency for the resize stage, and DVC won't trigger that stage until the previous one has completed.



Some of you will probably recognise this as a DAG, or directed acyclic graph
A great feature of DVC is that it caches stage outputs just like it can cache datasets

So if we run the pipeline and DVC detects that nothing has changed up until the train_lora stage, it will skip the preceding stages.
Real time saver.



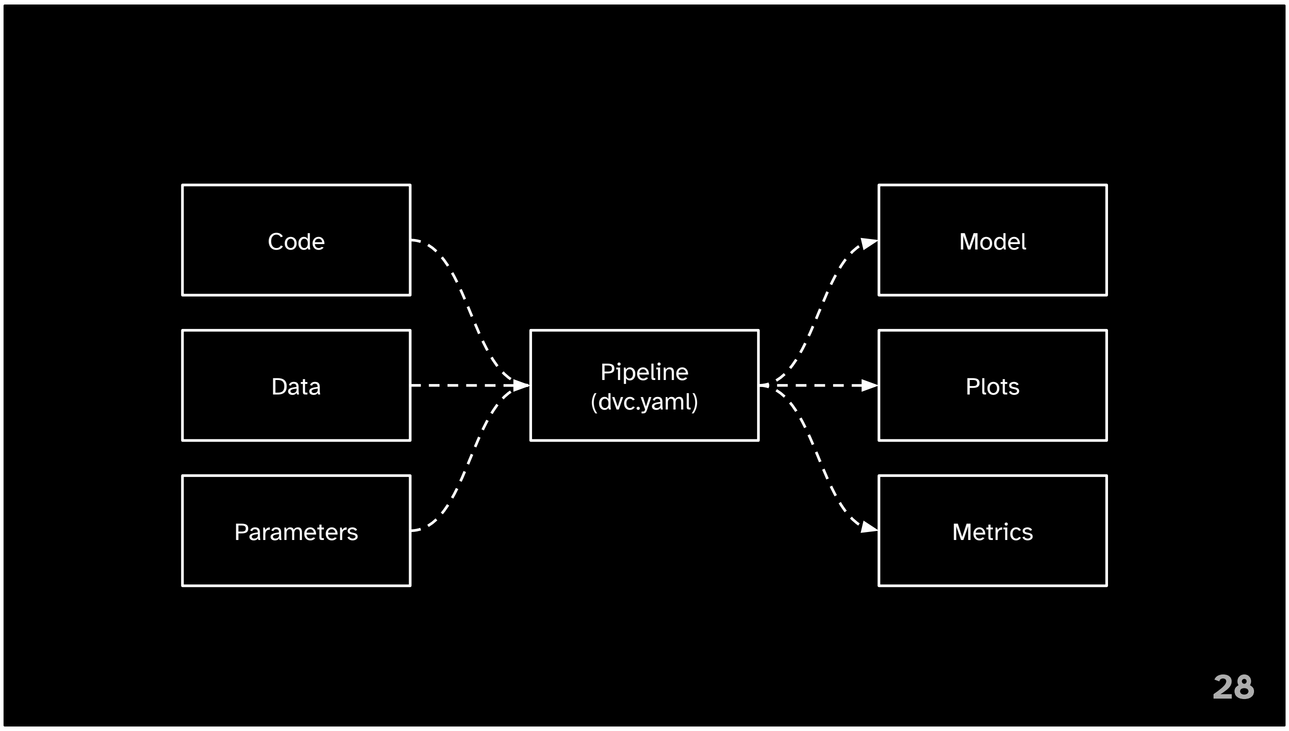
DVC

1 Data version control

2 Pipelines

3 Experiments

Now let’s see how DVC uses these foundations for ML experiment management

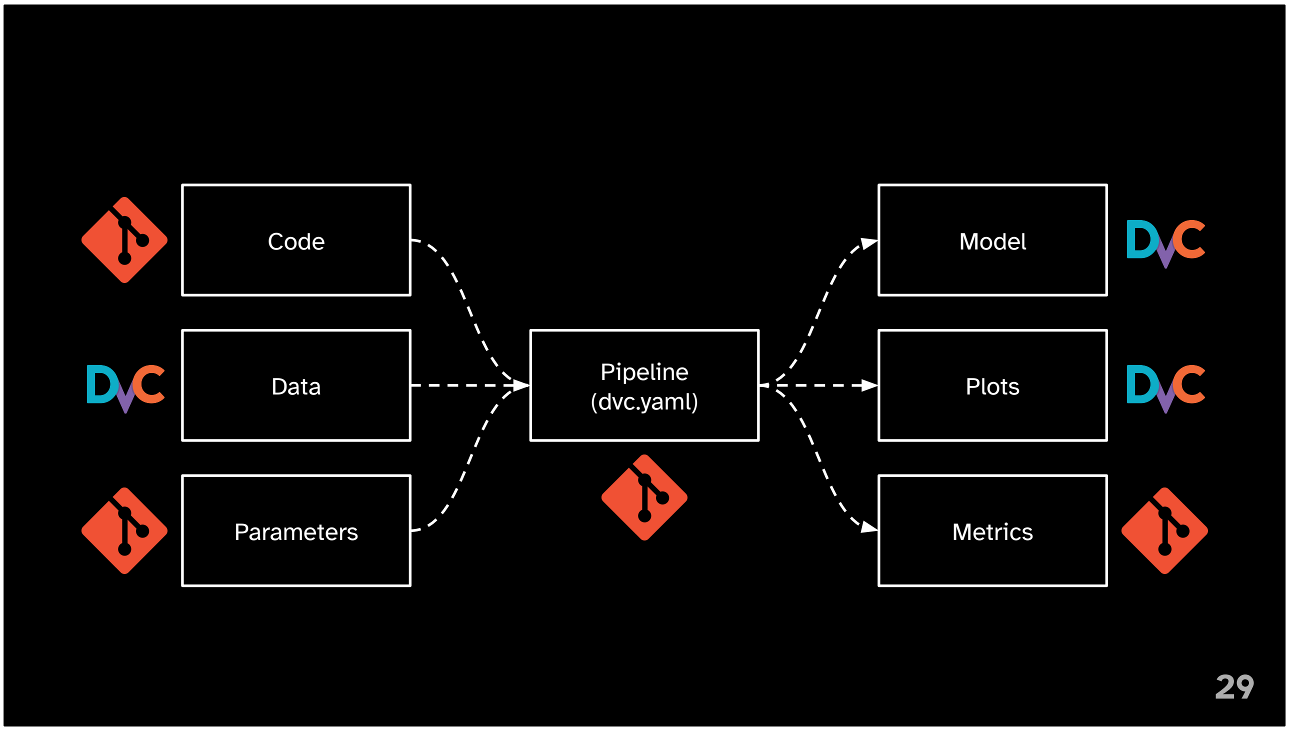


Let’s consider our pipeline, which we define in dvc.yaml. We have inputs (data, code, parameters) and outputs (model, plots, metrics)

We version each of these components with either Git or DVC
As a rule of thumb: small files with Git, larger ones with DVC

Now we can consider a pipeline run as a specific combination of these files as specified in Git; i.e. a commit.

And with two simple commands (git checkout and dvc checkout) we can switch between commits, or experiments, and compare their results and outputs.

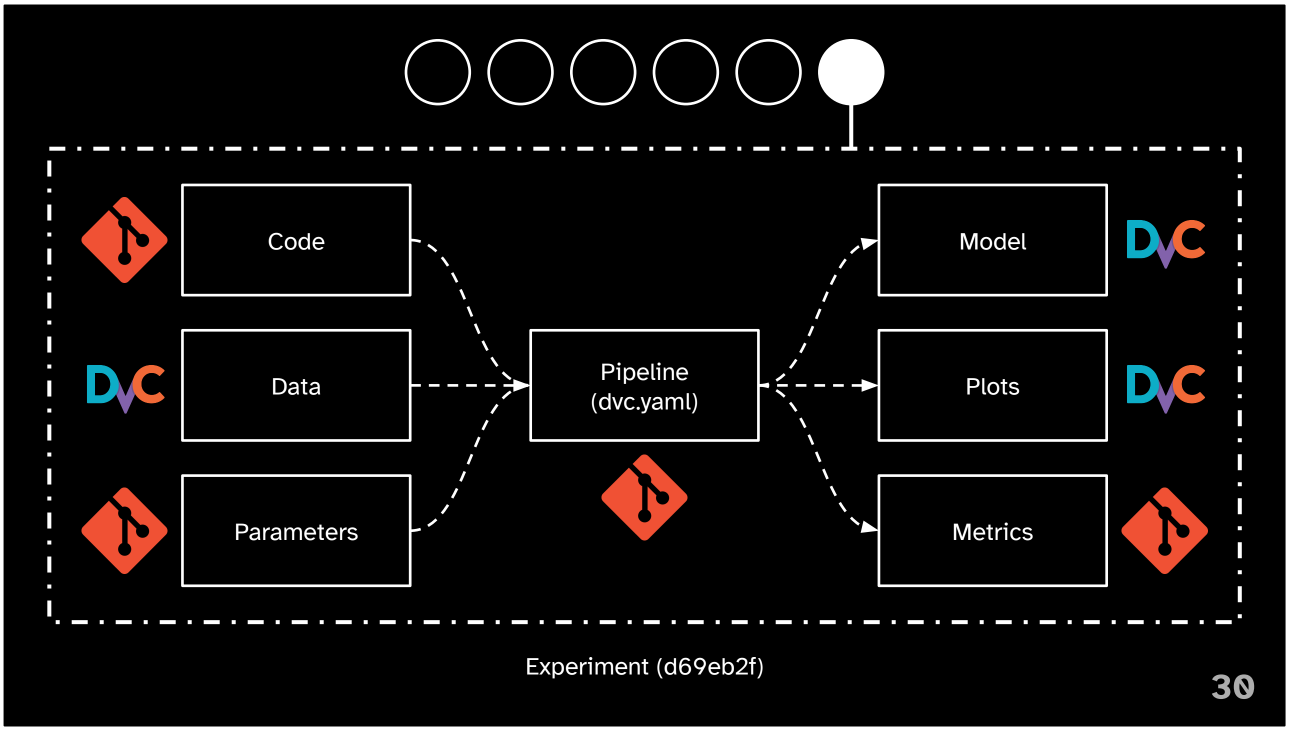


Let's consider our pipeline, which we define in dvc.yaml. We have inputs (data, code, parameters) and outputs (model, plots, metrics)

We version each of these components with either Git or DVC
As a rule of thumb: small files with Git, larger ones with DVC

Now we can consider a pipeline run as a specific combination of these files as specified in Git; i.e. a commit.

And with two simple commands (git checkout and dvc checkout) we can switch between commits, or experiments, and compare their results and outputs.

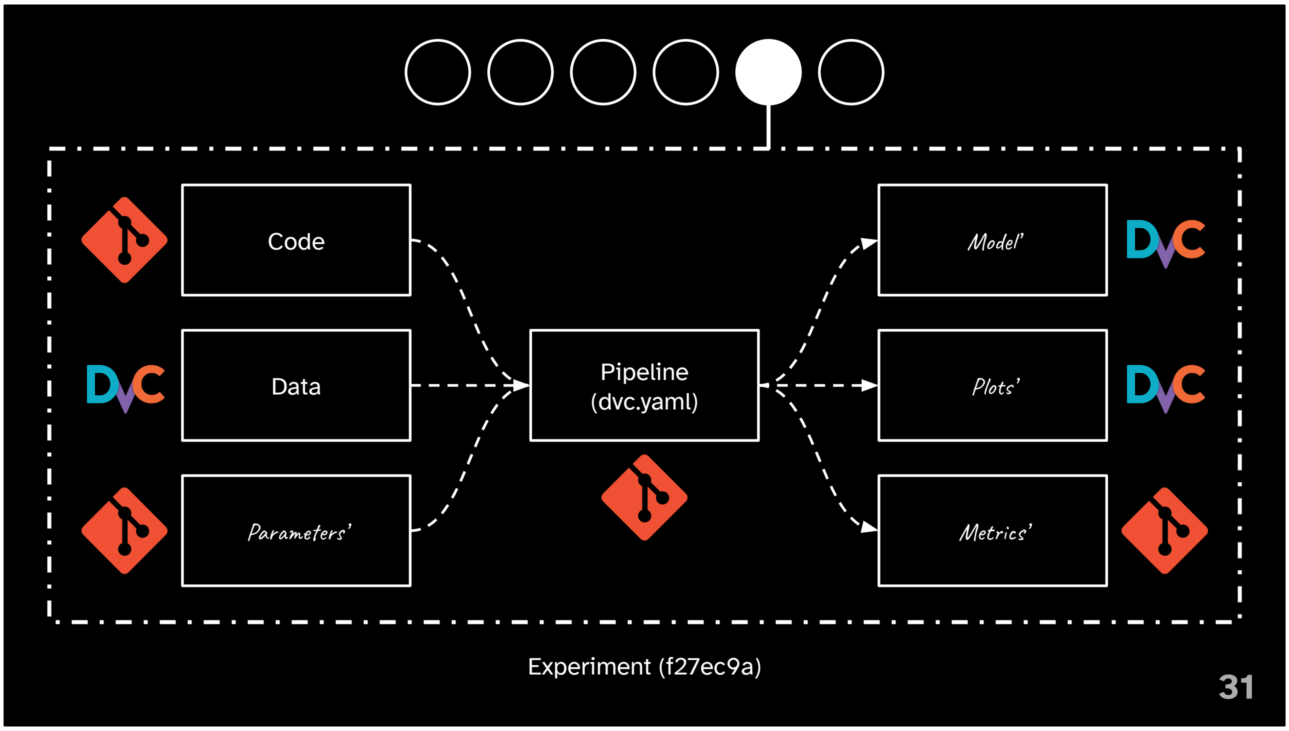


Let's consider our pipeline, which we define in dvc.yaml. We have inputs (data, code, parameters) and outputs (model, plots, metrics)

We version each of these components with either Git or DVC
As a rule of thumb: small files with Git, larger ones with DVC

Now we can consider a pipeline run as a specific combination of these files as specified in Git; i.e. a commit.

And with two simple commands (git checkout and dvc checkout) we can switch between commits, or experiments, and compare their results and outputs.



Let's consider our pipeline, which we define in dvc.yaml. We have inputs (data, code, parameters) and outputs (model, plots, metrics)

We version each of these components with either Git or DVC
As a rule of thumb: small files with Git, larger ones with DVC

Now we can consider a pipeline run as a specific combination of these files as specified in Git; i.e. a commit.

And with two simple commands (git checkout and dvc checkout) we can switch between commits, or experiments, and compare their results and outputs.
So here we see that the parameters have changed, resulting in different models/plots/metrics

Implementation 2
DVC pipeline

So what does that look like in practice? Let’s switch back to our IDE.

- Show Python modules
- Show dvc.yaml
- Show dvc dag
- Show params.yaml
- Show dvc.cache
- Show experiments management
    - Dvc exp show
    - VSCode extension

Recap and takeaways

In data science we experiment constantly

Experimentation requires reproducibility

Jupyter Notebooks are great for prototyping, not great for reproducibility

We can achieve full reproducibility with Git and DVC

Together, Git and DVC track data, code, and parameters

We can use DVC for Data version control

We can use DVC for
Pipelines

We can use DVC for
Experiment management

Thank you!

robdewit.nl
rob@binary3.dev

#opentowork

Further resources

- dvc.org
- dvc.org/chat
- github.com/RCdeWit/sd-pokemon-generator

Sources used

- dvc.org/doc/
- huggingface.co/docs/diffusers/
- github.com/cloneofsimo/lora/
- replicate.com/blog/lora-faster-fine-tuning-of-stable-diffusion/
- aituts.com/stable-diffusion-lora/
- huggingface.co/blog/lora0/
- old.reddit.com/r/StableDiffusion/comments/1171zhk/how_can_i_make_a_lora_model_on_my_m1_mac/jerageb/
- civitai.com/models/5115/pokemon-lora-ken-sugimori-style/
- kaggle.com/datasets/brdata/complete-pokemon-dataset-gen-iiv/
- https://stable-diffusion-art.com/lora/