DVC guided tour: walkthrough extended

Avoid confusion between DVC and Git scope

DVC cannot “dvc add/track” any data file that is already staged/committed to Git, because then DVC cannot safely “take ownership” without ambiguity and duplication. So be careful not to stage to git those data files that you would want dvc to track (e.g. avoid `git add .`).

DVC may refuse to `dvc add` a data file if the data file’s path is gitignored in a way the would prevent Git from tracking dvc’s metadata file (e.g. `.dvc`) associated with said data file. This is because DVC cannot guarantee reproducibility/portability if the pointer file would not be versioned in Git. So do note add a broad gitignore of the whole data path that could prevent Git from tracking dvc metadata files.

Note: `dvc add data_file` will create a `.dvc` pointer file placed next to the data file. It will also create a target specific `.gitignore` at the same path, if not path already ignored by root `.gitignore`, making the data file invisible to `git`.

- solution A: Keep data files visible to Git (i.e. don’t gitignore), but never stage it with git before `dvc`
- Solution B: ignore raw data by default but allow (“unignore”) dvc metadata

```
# ignore all data files by default
data/**
# do not ignore the directories themselves (so negations below can work)
!data/**/
# allow DVC metadata anywhere under data/
!data/**/*.dvc
!data/**/.gitignore
```

DVC subprojects

Usually there is a single `dvc.lock` per DVC project directory (the same directory that contains `dvc.yaml`). In a monorepo with multiple DVC subprojects, each subproject can have its own `dvc.yaml` and `dvc.lock`.

DVC plots are the “visual twin” of DVC metrics.

DVC treats certain files (often CSV/TSV/JSON/YAML series data, or images) as “plots” and can generate an HTML visualization from them. Think: learning curves, ROC, confusion matrix, etc.

Like metrics, plots are just declared outputs in `dvc.yaml`, so they become versioned-by-Git via commits/experiments, and DVC can render or compare them across versions.

Two key commands: - `dvc plots show`: render current plots (typically produces an HTML you open). - `dvc plots diff [revs...]`: overlay/compare plots across Git revisions or workspace vs HEAD (similar intent to dvc metrics diff, but visual).

Mental model: metrics diff answers “what numbers changed?”, plots diff answers “how did the curve/shape change?”

DVC cache

DVC cache is not a single “state” like a checkout, and it is not a linear history like Git.

- The cache is a content-addressed object store: it keeps blobs keyed by hash. If two versions of a file have different content, they have different hashes and can coexist in the cache at the same time. If they are identical, they reuse the same cached object.

So in practice:

- DVC does not keep “one cache per `dvc.lock`” or “one cache per commit”.
- it can retain many historical objects (from many commits/locks) simultaneously, as long as they are still present and not garbage-collected.
- What determines whether old cached objects remain:
  - whether something still references them (via current workspace, experiments, commits you have checked out, etc.)
  - whether you run cache cleanup / garbage collection (commands like `dvc gc` can remove unreferenced cache).

Mental model: Git stores snapshots by commit graph; DVC cache stores a bag of hashed blobs that may include many versions, but it is not organized as a timeline.

“dvc repro” cache behavior example 1

Manually change values in metrics file

```
$ dvc metrics diff
Path                   Metric    HEAD     workspace    Change
reports/metrics.json  accuracy  0.93333  0.09333      -0.84
```

first `repro` checks out cached output

```
$ dvc repro
'data/raw/iris.csv.dvc' didn't change skipping
Stage 'prepare' didn't change, skipping
Stage 'train' didn't change, skipping
Stage 'evaluate' is cached - skipping run, checking out outputs
```

second time (normal)

```
$ dvc repro
'data/raw/iris.csv.dvc' didn't change skipping
Stage 'prepare' didn't change, skipping
Stage 'train' didn't change, skipping
Stage 'evaluate' didn't change, skipping
```

“dvc repro” cache behavior example 2

- Change test split in `params.yaml`
- `dvc repro`
- `git diff`; see that `dvc.lock` and `params.yaml` have changed
- the artifacts also changed (`models/`, `data/preprocessed`), but Git is not tracking them and won’t show as changed.
- `git reset --hard HEAD`; changes `params.yaml` and `dvc.lock` back
- `git diff`; shows no changes but `dvc status` shows the differences!
- do another `dvc repro` and every stage will be skipped, but artifacts will be overwritten from cache.

DVC cache and remote

What is in .dvc/cache, and what gets pushed to remote?

- `.dvc/cache` is a content-addressed store of the actual file/directory contents for DVC-tracked data and outputs (hashed blobs). Your workspace files are typically links/copies that correspond to objects in the cache.
- `dvc push` uploads to the remote the cached objects referenced by your tracked `.dvc` files and `dvc.yaml` (and, for pipelines, the versions pinned by `dvc.lock`).
- The "run cache" (stage execution cache) is separate and is only pushed/pulled if you use `--run-cache`.

Does data have to be inside the Git repo directory?

No. You have a few supported patterns: - Track data that lives outside the project by placing it into the workspace with `dvc add -o <path>` (copying into the repo-managed location). - Use "external dependencies/outputs" in pipelines (deps/outs that are external paths or remote URLs). - If you want to avoid keeping a local copy entirely, `dvc add --to-remote` can send data straight to the remote and rely on `dvc pull` to materialize it later. - Independently, you can move the cache location (so the heavy storage is elsewhere) while still keeping tracked paths inside the repo.

What is the point of “dvc pull” - why not just “dvc repro”?

- `dvc pull` is for getting the exact versions of data/outputs/models/metrics that are already stored remotely, without recomputing. This is especially relevant after `git clone` or on another machine.
- `dvc repro` regenerates pipeline outputs from dependencies. It requires the inputs (raw data) to be present (via local cache or remote pull), and it costs compute/time. If outputs are already available in the remote, `dvc pull` is cheaper and more faithful to the pinned state.

In practice: after a fresh clone you often do `dvc pull` first; `dvc repro` is only needed if you want to recompute (or if some outputs are missing and cannot be pulled).

DVC remote command cheatsheet

- `dvc add <path>`: Put data under DVC tracking (creates `<path>.dvc` or updates `dvc.yaml`), stores content in cache.
- `dvc add --to-remote <path>`: Add data and upload it directly to remote (skip keeping a full local copy, depending on options).
- `dvc push`: Upload required cache objects for the current workspace/commit (and pipeline outs) to the remote.
- `dvc pull`: Download required objects from remote into cache and then checkout to workspace.
- `dvc fetch`: Download required objects from remote into cache only (no workspace checkout).
- `dvc checkout`: Materialize files in the workspace to match the tracked pointers/lockfile using local cache (and links/copies); does not download from remote by itself.
- `dvc gc`: Garbage-collect unused cache objects (based on what refs/commits/experiments you keep).

List experiments (data versions included)

```
dvc exp show
```

NOTE: `dvc exp show` only shows saved experiments (things created via dvc exp run/save). It will NOT magically list every historical dvc add you did unless those states were saved as experiments.

Use DVC experiments (dvc exp show) for exploratory runs/param sweeps, not as your primary dataset version ledger.

`dvc repro` vs `dvc exp run`

- `dvc repro`: reproduces the pipeline in your current workspace based on `dvc.yaml`, re-running only the out-of-date stages and updating `dvc.lock`.
- `dvc exp run`: runs the pipeline as an "experiment" and saves the result without you needing to make a Git commit; with no args it is effectively `dvc repro` plus saving the experiment result.

Practical rule:

- Use `dvc repro` for the main, canonical pipeline state (what you would normally commit).
- Use `dvc exp run` when you want to try many parameter/code changes and compare them without polluting Git history (you then view/compare via the experiments UI/commands).

What does `dvc exp run` have that `dvc repro` does not?

- `-S/--set-param` to override params on-the-fly (Hydra override syntax)
- `--queue`, `--run-all`, `-j/--jobs` for queued/parallel experiment execution
- `--temp` to run outside your workspace
- `-n/--name` and `-m/--message` for experiment identity/message

Plus it saves the run as an experiment under `refs/exps/*`

DVC experiments are stored under `.git/refs/exps`, not a normal branch

`dvc exp run` creates custom refs under `refs/exps/*`, not `refs/heads/*`. Typically `git checkout` is not used on them; DVC keeps them "hidden / not checked out" by default.

For a normal branch from an experiment use `dvc exp branch <exp>` (it creates a Git branch).

Git stores names that point to commits under namespaces like:

```
refs/heads/* -> normal branches
refs/tags/* -> tags
refs/exps/* -> DVC experiments
```

So an experiment is still "a named pointer to a commit", but it lives in a separate namespace and is managed by DVC. By default, it is also not pushed with plain `git push`; DVC provides `dvc exp push/pull` for that.

What does `dvc exp save` do?

`dvc exp save` snapshots your current project state as an experiment (without you making normal Git commits/branches), i.e. “save an experiment after the fact”.

`exp run` ~= `repro + exp save` -> `exp run` runs the pipeline and saves the result as an experiment, whereas `exp save` saves a snapshot without necessarily running anything at that moment.

“Permanent” vs other experiment states

A useful way to think about states:

- Workspace-only: you ran something and have changes, but nothing saved as an experiment
- Saved experiment: exists under `refs/exps/*` (named, comparable via `dvc exp show`)
- Applied experiment: you restored an experiment into the workspace (`dvc exp apply`) but still not in normal Git history
- Branched experiment: converted into a normal Git branch (`dvc exp branch`)
- Committed/persistent: you commit the applied results into your normal Git history (mainline)