Organizing Code, Experiments, and Research for Kaggle Competitions

Lessons and tips learnt while earning a Kaggle Competition Medal

Ibrahim Habib

2025-12-14

Table of contents

> Tell me and I forget. Teach me and I remember. Involve me and I learn.

The old adage still holds true, and *learning by doing* is one of the most instructive processes to acquire a new skill. In the field of data science and machine learning, participating in competitions is one of the most effective ways to gain hands-on experience and enhance your skills and abilities.

Kaggle is the world's largest data science community, and its competitions are highly respected in the industry. Many of the world's leading ML conferences (e.g., NeurIPS), organizations (e.g., Google), and universities (e.g., Stanford) host competitions on Kaggle.

The featured Kaggle Competitions award medals to top performers on the private leaderboard. Recently, I've participated in my very first medal-awarding Kaggle competition, and I was fortunate enough to earn a **Silver Medal**. This was the NeurIPS - Ariel Data Challenge 2025. I don't intend to share my solution here. If you're interested, you can check out my solution here.

What I didn't realize prior to participation is how much Kaggle tests besides just ML skills.

Kaggle tests one's coding and software engineering skills. It stressed one's ability to properly organize their codebase in order to quickly iterate and try new ideas. It also tested the ability to track experiments and results in a clear manner.

Being part of the NeurIPS 2025 Competition Track, a research conference, also tested the ability to research and learn about a new domain quickly and effectively.

All in all, this competition humbled me a lot and taught me many lessons besides ML.

The purpose of this article is to share some of these non-ML lessons with you. They all revolve around one principle: **organization**, **organization**, **organization**.

First, I will convince you why clear code structuring and process organization isn't *time wasting* or *nice to have*, but rather *essential* for competing in Kaggle specifically and any successful data science project in general. Then, I will share with you some of the techniques I used and lessons learned regarding code structuring and the experimentation process.

I want to start with a note of humility. By no means am I an expert in this field. I am still in the outset of my journey. All I hope for is that some readers will find some of these lessons beneficial and will learn from my pitfalls. If you have any other tips or suggestions, I urge you to share them in the **comments section** below, so that we all can learn together.

1 Science Golden Tip: Organize

It is no secret that natural scientists like to keep detailed records of their work and research process. Unclear steps may (and will) lead to incorrect conclusions and understanding. Irreproducible work is the bane of science. For us data scientists, why should it be any different?

1.1 But Speed is Important!

The common counterargument is that the nature of data science is *fast-paced* and *iterative*. Generally speaking, experimentation is cheap and quick; besides, who in the world prefers writing documentation over coding and building models?

As much as I sympathize with this thought and I love quick results, I fear that this mindset is short-sighted. Remember that the final goal of any data science project is to either deliver accurate, data-supported, and reproducible insights or to build reliable and reproducible models. If fast work compromises the end goal, then it is not worth anything.

My solution to this dilemma is to make the *mundane* parts of organization as simple, quick, and painless as possible. We shouldn't seek total deletion of the organization process, but rather fix its faults to make it as efficient and productive as possible.

1.2 Costs of Lack of Organization

Imagine with me this scenario. For each of your experiments, you have a single notebook on Kaggle that does everything from loading and preprocessing the data to training the model, evaluating it, and finally submitting it. By now, you have run dozens of experiments. You discover a small bug in the data loading function that you used in all your experiments. Fixing it will be a nightmare because you will have to go through each of your notebooks, fix the bug, ensure no new bugs were introduced, and then re-run all your experiments to get the updated results. All of this would have been avoided if you had a clear code structure and your code were reusable and modular.

Drivendata (2022) mentions a great example of the costs of an unorganized data science project. It mentions the story of a failed data science project that took months to complete and cost millions of dollars. The failure came down to an incorrect conclusion discovered early in the project. A code bug in the data cleaning polluted the data and led to wrong insights. If the team had better tracked the data sources and transformations, they would have caught the bug earlier, and money would have been saved.

If there is one lesson to take away from this section, it is that **organization is not a nice-to-have, but rather an essential part of any data science project**. Without clear code structure and process organization, we are bound to make mistakes, waste time, and produce irreproducible work.

1.3 What to track and organize?

There are three main aspects that I consider worth the effort to track:

1. Codebase
2. Experiments Results and Configurations
3. Research and Learning

2 The Codebase

After all, code is the backbone of any data science project. So, there is a lesson or two to learn from software engineers here.

2.1 Repo Structure

As long as you give much thought to the structure of your codebase, you are doing great.

There is no one universally agreed upon structure (nor will ever be). So, this section is highly subjective and opinionated. I will discuss the general structure I like and use.

I like to initialize my work with the widely popular Cookiecutter Data Science (ccds) template. When you initialize a project with ccds, it creates a folder with the following structure.[1]

```
├── LICENSE            <- Open-source license if one is chosen
├── Makefile           <- Makefile with convenience commands like `make data`
or `make train`
├── README.md          <- The top-level README for developers using this
project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default mkdocs project; see www.mkdocs.org for
details
│
├── models             <- Trained and serialized models, model predictions, or
model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number
(for ordering),
│                         the creator's initials, and a short `-` delimited
description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── pyproject.toml     <- Project configuration file with package metadata
for
│                         {{ cookiecutter.module_name }} and configuration for
tools like black
│
├── references         <- Data dictionaries, manuals, and all other
explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in
reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis
environment, e.g.
│                         generated with `pip freeze > requirements.txt`
```

[1]The folder structure was copied from ccds website on the time of article publishing.

```
│
├── setup.cfg          <- Configuration file for flake8
│
└── {{ cookiecutter.module_name }}   <- Source code for use in this project.
    │
    ├── __init__.py             <- Makes {{ cookiecutter.module_name }} a
Python module
    │
    ├── config.py               <- Store useful variables and configuration
    │
    ├── dataset.py              <- Scripts to download or generate data
    │
    ├── features.py             <- Code to create features for modeling
    │
    ├── modeling
    │   ├── __init__.py
    │   ├── predict.py          <- Code to run model inference with trained
models
    │   └── train.py            <- Code to train models
    │
    └── plots.py                <- Code to create visualizations
```

2.1.a Environment Management

When you use ccds, you are prompted to select an environment manager. I personally prefer uv by Astral. It records all the used packages in the `pyproject.toml` file and allows us to recreate the same environment by simply using `uv sync`.

Under the hood, `uv` uses venv. I find using `uv` much simpler than directly managing virtual environments because managing and reading `pyproject.toml` is much simpler than `requirements.txt`.

Moreover, I find `uv` much simpler than `conda`. `uv` is built specifically for python while `conda` is much more generic.

2.1.b The Generated Module

A great part of this template is the `{ cookiecutter.module_name }` directory. In this directory, you defined a Python package that shall contain all the important parts of your code (e.g. preprocessing functions, models definition, inference function, etc.).

I find the usage of the package quite helpful, and in Section 2.3, I will discuss what to place here and what to place in Jupyter Notebooks.

2.1.c Staying Flexible

Don't regard this structure as perfect or complete. You don't have to use everything `ccds` provides, and you may (and should) alter it if the project requires it. `ccds` provides you with a great starting point for you to tune to your exact project needs and demands.

2.2 Version Control

Git has become an absolute necessity for any project involving code. It allows us to track changes, revert to earlier versions, and, with GitHub, collaborate with team members.

When you use Git, you basically access a time machine that can remedy any faults you introduce to your code. Today, the use of Git is non-negotiable.

2.3 The Three Code Types

Choosing when to use Python scripts and when to use Jupyter Notebooks is a long-debated topic in the data science community. Here I present my stance on the topic.

I like to separate all of my code into one of three directories:

1. The Module
2. Scripts
3. Notebooks

2.3.a The Module

The module should contain all the important functions and classes you create.

Its usage helps us minimize redundancy and create a single source of truth for all the important operations happening on the data.

In data science projects, some operations will be repeated in all your training and inference workflows, such as reading the data from files, transforming data, and model definitions. Repeating all these functions in all your notebooks or scripts is difficult and extremely boring. Using a module allows us to write the code once and then import it everywhere.

Moreover, this helps reduce errors and mistakes. When a bug in the module is discovered, you fix it once in the module, and it's automatically fixed in all scripts and notebooks importing it.

2.3.b Scripts

The scripts directory contains .py files. These files are the only source of generating outputs from the project. They are the interface to interacting with our module and code.

The two main usages for these files are training and inference. All the used models should be created by running one of the scripts, and all submissions on Kaggle should be made by such files.

The usage of these scripts helps make our results reproducible. To reproduce an older result (train the same model, for example), one only has to clone the same version of the repo and run the script used to make the old results[2].

Since the scripts are run from the CLI, using a library to manage CLI arguments simplifies the code. I like using typer for simple scripts that don't have many config options and using hydra for complex ones (I will discuss hydra in more depth later).

[2]It is assumed that a seed is used in all random number generators.

2.3.c Notebooks

Jupyter Notebooks are wonderful for exploration and prototyping because of the short feedback loop they provide.

On many occasions, I start writing code in a notebook to quickly test it and figure out all mistakes. Only then would I transfer it to the module.

However, notebooks shouldn't be used to create final results. They are hard to reproduce and track changes in. Therefore, always use the scripts to create final outputs.

3 Running the Codebase on Kaggle

Using the structure discussed in the previous section, we need to follow these steps to run our code on Kaggle:

1. Clone The Repo
2. Install Required Packages
3. Run one of the Scripts

Because Kaggle provides us with a Jupyter Notebook interface to run our code and most Kaggle competitions have restrictions on internet access, submissions aren't as straightforward as running a script on our local machine. In what follows, I will discuss how to perform each of the above steps on Kaggle.

3.1 Cloning The Repo

First of all, we can't directly clone our repo from GitHub in the submission notebook because of the internet restrictions. However, Kaggle allows us to import outputs of other Kaggle notebooks into our current notebook. Therefore, the solution is to create a separate Kaggle notebook that clones our repo and installs the required packages. This notebook's output is then imported into the submission notebook.

Most likely, you will be using a private repo. The simplest way to clone a private repo on Kaggle is to use a personal access token (PAT). You can create a PAT on GitHub by following this guide. A great practice is to create a PAT specifically for Kaggle with the minimal required permissions.

In the cloning notebook, you can use the following code to clone your repo:

```
from kaggle_secrets import UserSecretsClient
user_secrets = UserSecretsClient()
github_token = user_secrets.get_secret("GITHUB_TOKEN")
user = "YOUR_GITHUB_USERNAME"
CLONE_URL = f"https://oauth2:{github_token}@github.com/{user}/
YOUR_REPO_NAME.git"
get_ipython().system(f"git clone {CLONE_URL}")
```

This code downloads your repo into the working directory of the current notebook. It assumes that you have stored your PAT in a Kaggle secret named `GITHUB_TOKEN`. Make sure that you activate the secret in the notebook settings before running it.

3.2 Installing Required Packages

In the cloning notebook, you can also install the required packages. If you are using `uv`, you can build your custom module, install it, and install its dependencies by running the following commands:[3].

```
cd ariel-2025 && uv build
```

This creates a wheel file in the `dist/` directory for your module. You can then install it and all its dependencies in a custom directory by running:[4].

```
pip install /path/to/wheel/file --target /path/to/custom/dir
```

Make sure to replace `/path/to/wheel/file` and `/path/to/custom/dir` with the actual paths. The `/path/to/wheel/file` will be the path to the .whl file inside the `REPO_NAME/dist/` directory. The `/path/to/custom/dir` can be any directory you like. Remember the custom directory path because subsequent notebooks will rely on it to import your module and your project dependencies.

I like to both download the repo and install the packages in a single notebook. I name this notebook the same name as the repo to simplify importing it later.

3.3 Running One of the Scripts

The first thing to do in any subsequent notebook is to import the notebook containing the cloned repo and installed packages. When you do this, Kaggle stores the contents of `/kaggle/working/` from the imported notebook into a directory named `/kaggle/input/REPO_NAME/`, where `REPO_NAME` is the name of the repo[5].

Many times, your scripts will create outputs (e.g., submission files) relative to their locations. By default, your code will live on /kaggle/input/REPO_NAME/, which is read-only. Therefore, you need to copy the contents of the repo to `/kaggle/working/`, which is the current working directory and is read-write. While this may be unnecessary, it is a good practice that causes no harm and prevents silly issues.

```
cp -r /kaggle/input/REPO_NAME/REPO_NAME/ /kaggle/working/
```

If you directly run your scripts from `/kaggle/working/scripts/`, you will get import errors because Python can't find the installed packages and your module. This can easily be solved by updating the `PYTHONPATH` environment variable. I use the following command to update it and then run my scripts:

---

[3] When running any bash command in a code cell in a Jupyter Notebook, you need to prefix the command with `!`.

[4] When running any bash command in a code cell in Kaggle, it is assumed that the current working directory is `/kaggle/working/`. Any previous `cd` commands are ignored. You have to specify the path from `/kaggle/working/` in all your commands or start your commands with `cd PATH &&` to change the directory before running the rest of the command.

[5] If you forget the path, you can copy it from the notebook tab in the right sidebar of the notebook.

```
! export PYTHONPATH=/kaggle/input/REPO_NAME/custom_dir:$PYTHONPATH && cd /
kaggle/working/REPO_NAME/scripts && python your_script.py --arg1 val1 --arg2
val2
```

I usually name any notebook running a script with the script name for simplicity. Moreover, when I re-run the notebook on Kaggle, I name the version with the hash of the current Git commit to keep track of which version of the code was used to generate the results.[6].

3.4 Gathering Everything Together

At the end, two notebooks are necessary:

1. The Cloning Notebook: clones the repo and installs the required packages.
2. The Script Notebook: runs one of the scripts.

You may need more script notebooks in the pipeline. For example, you may have one notebook for training and another for inference. Each of these notebooks will follow the same structure as the script notebook discussed above.

Separating each step in the pipeline (e.g. data preprocessing, training, inference) into its own notebook is useful when one step takes a long time to run and rarely changes. For example, in the Ariel Data Challenge, my preprocessing step took more than seven hours to run. If I had everything in one notebook, I would have to wait seven hours every time I tried a new idea. Moreover, time limits on Kaggle kernels would have made it impossible to run the entire pipeline in one notebook.

Each notebook would then import the previous notebook's output and run its own step, and build from there. A good advice is to make the paths of any data files or models arguments to the scripts so that you can easily change them when running on Kaggle or any other environment.

When you update your code, re-run the cloning notebook to update the code on Kaggle. Then, re-run only the necessary script notebooks to generate the new results.

3.5 Is all this Effort Worth it?

Absolutely yes!

I know that the specified pipeline will add some overhead when starting your project. However, it will save you much more time and effort in the long run. You will be able to write all your code locally and run the same code on Kaggle.

When you create a new model, all you have to do is copy one of the script notebooks and change the script. No conflicts will arise between your local and Kaggle code. You will be able to track all your changes using Git. You will be able to reproduce any old results by simply checking out the corresponding Git commit and re-running the necessary notebooks on Kaggle.

[6]Note that you will have to re-run the cloning notebook to update the code on Kaggle before re-running the script notebook. Don't forget to "Check for Updates" in the cloning notebook before re-running the script notebook.

Moreover, you will be able to develop on any machine you like. Everything is centralized on GitHub. You can work from your local machine. If you need more power, you can work from a cloud VM. If you want to train on Kaggle, you can do that too. All your code and environment are the same everywhere.

This is such a small price to pay for such a great convenience. Once the pipeline is set up, you can forget about it and focus on what matters: researching and building models!

4 Recording Learnings and Research

When diving into a new domain, a huge part of your time will be spent researching, studying, and reading papers. It is easy to get lost in all the information you read, and you can forget where you encountered a certain idea or concept. To that end, it is important to manage and organize your learning.

4.1 Readings Tracking

Rajpurkar (2023) suggests having a list of all the papers and articles you read. This allows you to quickly overview what you have read and refer back to it when needed.

Professor Rajpurkar also suggests annotating each paper with one, two, or three stars. One-star papers are irrelevant papers, but you didn't know that before reading them. Two-star papers are relevant. Three-star papers are highly relevant. This allows you to quickly filter your readings later on.

You should also take notes on each paper you read. These notes should focus on how the paper relates to your project. They should be short to be reviewed easily, but have enough details to grasp the main ideas. In the papers list, you should link reading notes to each paper for easy access.

I also like keeping notes on the papers themselves, such as highlights. If you're using a PDF reader or an e-Ink device, you should store the annotated version of the paper for future reference and link it in your notes. If you prefer reading on paper, you can scan the annotated version and store it digitally.

4.2 Tools

For most documents, I like using Google Docs because it allows me to access my notes from anywhere. Moreover, you can write on Google Docs in Markdown, which is my preferred writing format (I am using it to write this article).

Zotero is a great tool for managing research papers. It is great at storing and organizing papers. You can create a collection for each project and store all the relevant papers there. Importing papers is very easy using the browser extension, and exporting citations in BibTeX format is straightforward.

5 Experiment Tracking

In data science projects, you will often run many experiments and try many ideas. Once again, it is easy to get lost in all this mess.

We have already made a great step forward by structuring our codebase properly and using scripts to run our experiments. Nevertheless, I want to discuss two software tools that allow us to do even better.

5.1 Wandb

Weights and Biases (wandb), pronounced "w-and-b" (for weights and biases) or "wand-b" (for being magical like a wand) or "wan-db" (for being a database), is a great tool for tracking experiments. It allows us to run multiple experiments and save all their configurations and results in a central place.

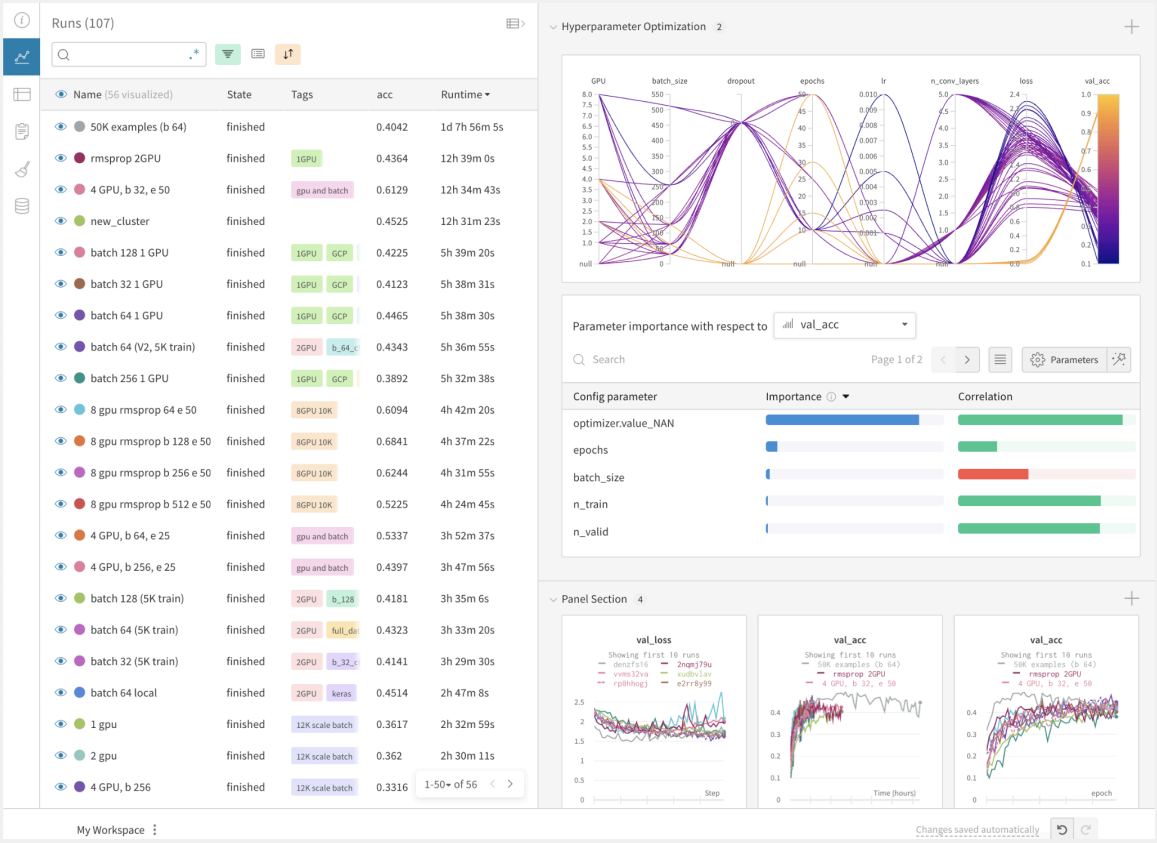

Figure 1: Wandb Dashboard Image from Adrish Dey's Configuring W&B Projects with Hydra article

Wandb provides us with a dashboard to compare the results of different experiments, the hyperparameters used, and the training curves. It also tracks system metrics such as GPU and CPU usage.

Wandb also integrates with Hugging Face libraries, making it easy to track experiments when using transformers.

Once you start using multiple experiments, wandb becomes an indispensable tool.

5.2 Hydra

Hydra is a tool built by Meta that simplifies configuration management. It allows you to define all your configuration in YAML files and easily override them from the CLI.

It is a very flexible tool and fits multiple use cases. This guide discusses how to use Hydra for experiment configuration.

6 The End-to-End Process

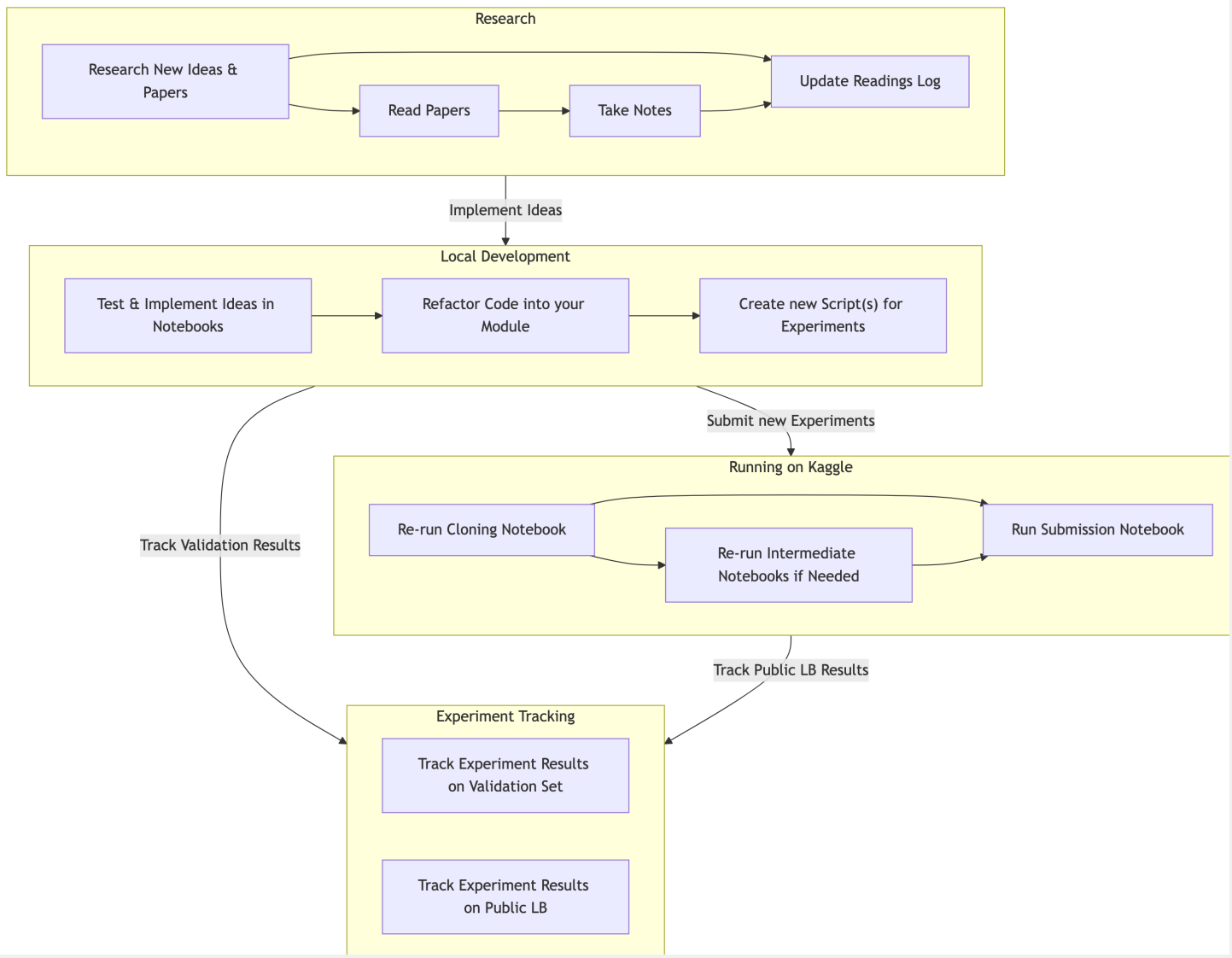

Figure 2: End-to-End Organized Kaggle Competition Process created by the Author using Mermaid.js

Figure 2 summarizes the process discussed in this article. First, we research ideas and record our learnings. Then, we experiment with these ideas on our local machines in Jupyter Notebooks. Once we have a working idea, we refactor the code into our module and create scripts to run the

experiments. We run the new experiment(s) on Kaggle. Finally, we track the results of the new experiments.

Because everything is carefully tracked, we are able to predict our shortcomings and quickly head back to the research or development phases to fix them.

7 Conclusion

Disorder is the source of all evil in data science projects. If we are to produce reliable and reproducible work, we must strive for organization and clarity in our processes. Kaggle competitions are no exception.

In this article, we discussed a technique to organize our codebase, tips to track research and learnings, and tools to track experiments. Figure 2 summarizes the proposed technique.

I hope this article was helpful to you. If you have any other tips or suggestions, please share them in the comments section below.

Best of luck in your next competition![7]

Bibliography

Drivendata. (2022). *The 10 Rules of Reliable Data Science.*

Rajpurkar, P. (2023). *Harvard CS197: AI Research Experiences.* https://www.cs197.seas.harvard.edu/

[7] This article was originaly published on Towards Data Science on November 13, 2025.