## System Design Document: Model Management System in Git

Drafted by Eric Oulster

*Initialization:*

To design this system from Scratch, the first step is to actually have a Github repo. Assuming that this particular git repository will be the only implementation across all companies, this step will only need to be done once, and at the beginning of the project.

While it is presumed in this document you know the basics of Github, using an account under which you want to create the repository, go to github.com (or enterprise-specific URL for Github), go under repositories, and click the green ‘new’ button to start the repository creation process.

Should the pattern outlined in this system be expected to be replicated several times, or should you feel more comfortable doing so programmatically, I would propose automating some of the repository creation process using PyGithub.

*Branches:*

I would propose leaving the default branch of ‘main’ for documentation for the solution (as I have done with this repo). While it may seem odd not to do the core of the development work on the ‘main’ branch, this is for a couple of reasons:

Using a default system name makes this particular branch subject to the idiosyncrasies of both git clients and Github. Git clients accessing a repository may default to pulling this branch (even if another branch were to be set as default in Github itself), and so having the documentation set to this branch means someone is most likely to see it after cloning the repo.

It is also possible (though unlikely) that main would be subject to change as the system default, as what happened with the previously named ‘master’ branch. Keeping this branch relatively stable helps to prevent future organization issues with regards to new system default names.

Beyond main for documentation, you will have the following branches which serve the following purposes:

‘build’ – This is the branch that each company will pull from. It is also where your developers will push stable builds of their model objects and data. Everything in this branch should be bug-free, functional, and ready for use by any companies.

Within this branch are each of the model folders, and within that the models and dependencies needed for the models themselves.

This branch is also where the Dependency Matrix is housed, a .csv file which works as a grid to show what models are associated with what product.

‘develop’ – This is the branch where your model developers track their in-progress tasks. I would generally recommend creating branches off of develop when a developer is working on a particular feature or stream of work in parallel to others, and then merge back into develop when ready.

These branches can be named with the feature/[request name], or some other systemized approach which works with your developer ceremonies. Adding ‘feature/’ to sub-branches off of develop is to prevent the unlikely possibility that a company has the same name as a developer’s branch, and causing an issue with branches. Developers should push their changes often, pull before starting new tasks, and do their work within the branch itself, instead of elsewhere on their computer.

Not everything has to run perfectly in develop, as this branch is for works-in-progress. Commits and changes to develop should be integrated into ‘build’ eventually, and when the time comes to author pull-requests to make that happen, the code should be flawless.

‘[company]\_[product]’ – Each repo would be named after the actual company in question, as well as the specific product. This is where each customer obtains the model and model dependencies, and, depending on their level of data maturity, may also use it to store some code related to their model.

Arguably, each of these is a “deployment” of the model, as it will be where the company’s developers interact with it.

Whether or not the company’s branch is purely for them to obtain the model, or to use it as a development environment as well is subject to further discussion and technical scoping of the project. If they are intended to develop in this branch, please note they will need to have access to collaborate in the repo as well.

*Branch protections:*

I generally believe it is more important to show developers the right patterns of using git, rather than hard-coding limitations in place.

As most git actions can be easily undone, under most circumstances, an error can be a learning opportunity, instead of a wild scramble to fix something broken.

Under a more permissive repository structure, the main branch protection to consider would be forcing all merges to build to require a pull request, with the reviewers either being peer developers, a product owner, or one of the more code-savvy senior developers. This encourages a culture of shared ownership over the codebase, adds a failsafe against erroneous code making it to a live system, and helps developers better understand both code style and function that their peers are implementing.

Generally, I would anticipate the only branch that would merge to the build branch would be develop. If individual customers are pushing changes though, and they are welcome changes, this allows the developers to evaluate these pull requests, too.

Any additional permissions schemes would require further scoping based on the use case, but there may be reasons to disable companies from authoring changes to other company’s repositories, and disabling allowing them to push changes to develop branches & main, too.

*Access Token:*

One additional piece required for this solution is an access token from Github.

This can be accessed under the profile icon -> settings -> developer settings -> personal access tokens -> Fine-grained tokens.

Though Fine-grained tokens are still in a ‘preview’ state at the time of writing, they provide repo-specific access, and allow you to choose only the necessary settings from a security perspective for this project to run.

Going over each of the individual permissions settings on the access token would be out of scope. For the purposes of brevity, you can grant the token read-write access on any actions specifically for this repo, but not general account-level ones.

After creating your access token, you should copy and add it to .json file on your local development environment, with the token as the value, and the key being named ‘key’.

If you store the token in the same folder as the git repo, be sure to .gitignore the key.

*The Solution – High Level:*

There is an automated tool that takes in the name of the company, as well as the specific product. The tool scans the dependency matrix by product to determine what models are needed (and not needed) and saves these as variables.

The tool then automatically creates a branch out of build, erases unneeded models in the first commit, then pushes that commit, allowing for a clean branch with only the models needed.

*The Solution – Technical:*

As of right now, the solution is fairly simple. A Python script accesses your authentication token to access the repo, then accepts the name of both the company that the repository is being built for, and the product itself. The script uses PyGithub to read the csv dependency matrix on the build branch, then uses pandas (and a call to StringIO) to parse the file for the given product.

The dependency-matrix.csv file is assumed to contain records for all models and products, as well as whether or not they are dependencies (as outlined by 0 or 1). It checks the row for a specific product (it is assumed there are no duplicate rows), then for that product, returns 1’s and 0’s for each model. The script saves the names of models not needed, to mark them for deletion.

Using PyGithub, the script ensures that the branch looking to be created doesn’t already exist. If the branch does not exist, the script creates the branch from build, naming it ‘company’ + ‘\_’ + ‘product’.

Pygithub looks up the model folders, then iteratively deletes all files within those folders, authoring a commit for each file with the file name in it. The process is then complete.

This presently exists as a python script, but depending on use case, this could be generalized to a command line interface, or executed as a trigger in response to a form submission, or triggered as a file is being entered into a storage medium.

Adding new products is done by adding the folder + files to the build branch, and editing dependency-matrix.csv accordingly.

*Why use branches instead of folders?*

The original instructions for this assignment asked to create folders in a branch, I asked (and was given the go-ahead) to use branches instead of folders.

There are a couple of reasons I proposed this change to the original instructions for designing my solution, they are as follows:

1. Less “Process” required to track & implement changes
2. Pushes from one company-branch don’t affect another one.
3. Better branch-level controls available
4. Utilizing version control on files saves space

Point 1 – Less process required to track & implement changes

If the folders, and objects inside, keep the same name from build to the company branches, it will be easier to native track them inside git. This means that propagating new changes from the build branch to the company branch is as easy as teaching the developers how to merge from build into their company branch. This saves a massive amount of time and code to configuring out a solution to mirror the changes in one folder to several other folders in the same branch, and allows git to just version control the files across several branches.

Point 2 – Pushes from one company-branch don’t affect another company

Frequently when working with developers in a shared repository, I see git issues arise where one developer has pulled a branch, has done work on gradually for a week (or longer), and then when attempting to push their changes, has realized that several commits from other developers have created a mismatch between their push and the current state of the repository. Having several companies work in the same repo, possibly authoring commits of their own, means this issue could be commonplace. Giving each company (and product for that company) its own branch ensures fewer issues within the timeline of git commits, especially given that companies are likely not communicating with one another.

Point 3 – Better branch-level controls available

While no such requirements were outlined in the initial specification, it may make sense to block developers from CompanyA from editing code intended for CompanyB, and vice-versa. It is far more difficult to incorporate write protections for an individual folder within a branch than it is to add write protections for the branch itself. That way, there is better internal protections available by partitioning the company-products into branches instead of folders.

Point 4 – Utilizing version control on files saves space

Perhaps a smaller issue, but by creating branches, we prevent duplicating files unnecessarily. Duplicating files requires git to track those files as well, rather than just tracking the *changes* on those files. This creates a bigger memory footprint on the git repo, and that impact could start being seen more clearly if the project scales to 100’s of customers.

Using branches instead of copying folders allows files to be version-controlled well in git, which makes management of those files on git more efficient from both a performance and practicality perspective.

*Additional Considerations:*

Data & Git – Generally speaking, it is a bad idea to have any personally identifiable information (PII) as data stored in git. Should the model contain information like the addresses of customers, income of certain farms, or information pertaining to cyber or physical access to any of the companies on the repo, this could cause a security incident. It is an assumption of this project that no PII is present in the repo.

Security keys & Git – In addition to the above statement, it is also worth noting that any access credentials not be stored on git. This is why previous instructions ask you to add keys to the .gitignore file, so they will not be uploaded by accident. The project I am saving in this repo is missing this auth key required to run it, and you are expected to provide this yourself.

Good faith among companies & participants in repo – What is inherent in the instructions for this solution are that your customers are not trying to actively sabotage and spy on one another. I asked a clarifying question about whether or not all those who has access to the repo were acceptably able to view data in the repo, and it was to ensure this assumption could be made.

Repo size – I had asked a clarifying question after receiving this project about the size of the files in this repo. Large files in git repositories often require ‘git-lfs’ to be saved and stored. It is assumed that no files would get to that size. Should large enough files for git-lfs be required, the solution might change.

Performance – Using the timing present on a jupyter notebook, the file goes through the process of doing branch creation, deleting unneeded files & committing in 8.0 seconds.

Code – Please note that the code presented is not the complete system written about in this document, merely a component of it to show that the trickiest piece of the solution is still easily implemented.

Lastly – This system design is an initial proposal, not a final prescription. It makes assumptions, and should be discussed further to understand specific circumstances rather than implemented verbatim.