Deep Dive: Transforming your data with Code Repositories

What is Code Repositories in Foundry?

Code Repositories provides a web-based integrated development environment (IDE) for writing and collaborating on production-ready code in Foundry. With Code Repositories, data engineers can create efficient pipelines in bulk.

Example workflows that are a good fit for Code Repositories include:

- A daily pipeline at high data scale which requires incremental compute.
- A high-visibility pipeline with strict governance requirements to be able to revert to previous versions of historical code, or gate code changes on unit tests passing.

Learning objectives of the course

You will complete this course being able to create and manage code repositories, perform data transformations, and collaborate effectively.

By the end of this course, you will have

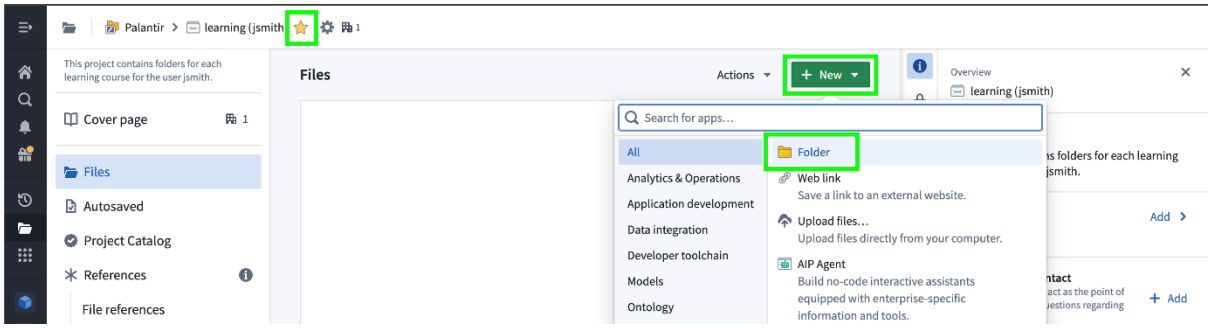
- created a PySpark Transform including casting and filtering,
- used joins and aggregations,
- worked collaboratively with other people using Branching,
- seen how Code Repositories interact with other Foundry tools like Data Lineage and Job Tracker.

Setting up your Project and Folder

Create a Course-Specific Training Folder

Step 1: Create your Folder

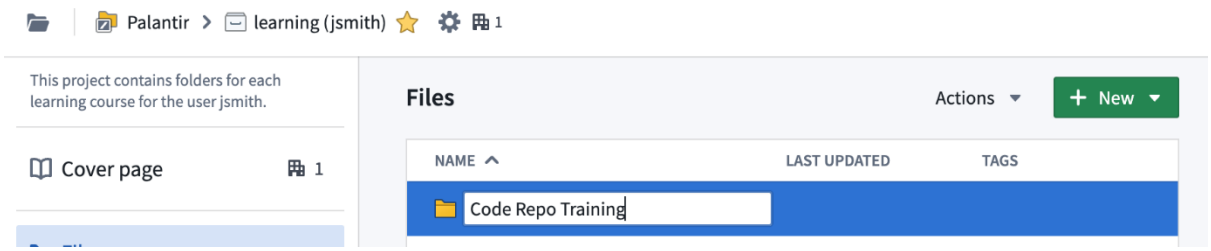
1. In the top left, click on the star to favorite your Project. This will allow you to find it quickly later on.
2. Click on **New.**
3. Click on **Folder.**



Step 2: Name your Folder

1. Name the new folder so that you can allocate it to the current learning course

    - For example: Code Repo Training



Data Transformation

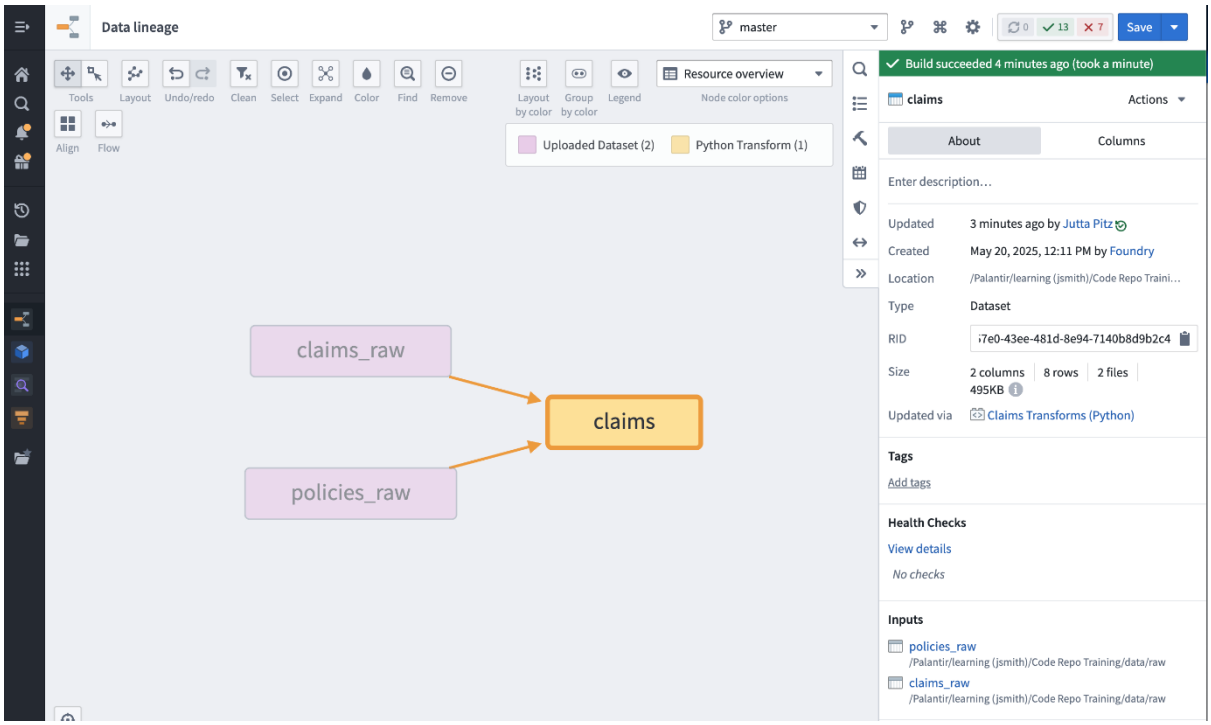
Introduction

In this section you will upload the data and create your Code Repository including the cleaning transformations to prepare your datasets. All the code you will be writing in this tutorial will be located in this repository.

Scenario

For this tutorial we assume the role of claims handler in a global insurance company. Your task is to present to the Chief Financial Officer the annual results of each line of business. Using code repository, you will process two main datasets, one containing each historic claim over the past years, and the other containing the corresponding policies, including the lines of business.

At the end of this section you will have implemented a cleaning pipeline using Python Transforms.



Download the Data Sources

Below are the different files you'll need to download and then upload into Foundry. Click the names of the files below to download them to your device.

- claims_raw.csv: A raw csv file containing all claims submitted by insurance customers in the years 2023 and 2024.
- policies_raw.csv: A raw csv file containing all policies the insurance has held in the years 2023 and 2024 together with the allocation to the respective line of business.

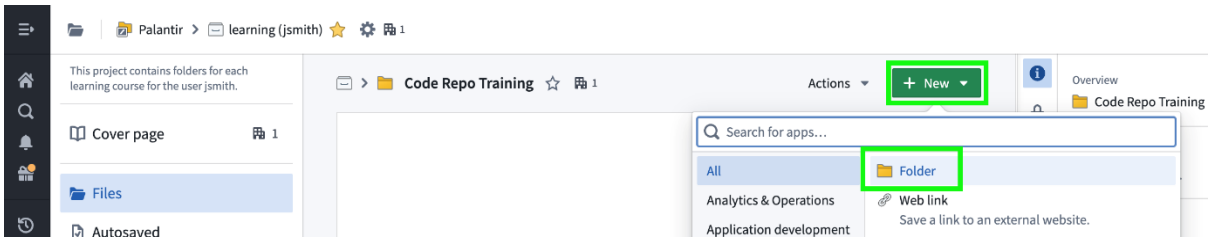
Important Downloading Tip: Your browser might open the files themselves instead of downloading them, use Ctrl + S or cmd + S to save the file.

Upload the Raw Datasets

In this section, you will upload your downloaded datasets.

Step 1: Create subfolders

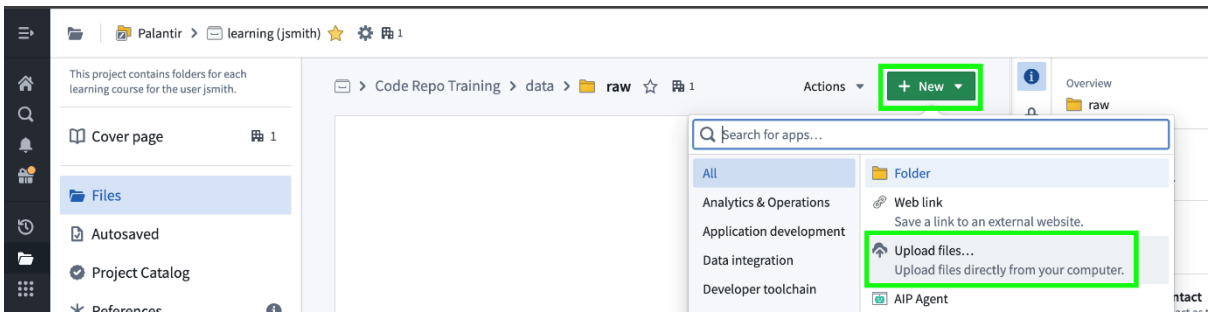
1. Open the training folder you created above: Code Repo Training
2. Create a new dedicated folder for your input datasets.
    1. Click **+New** in your project folder.
    2. Choose **Folder.**



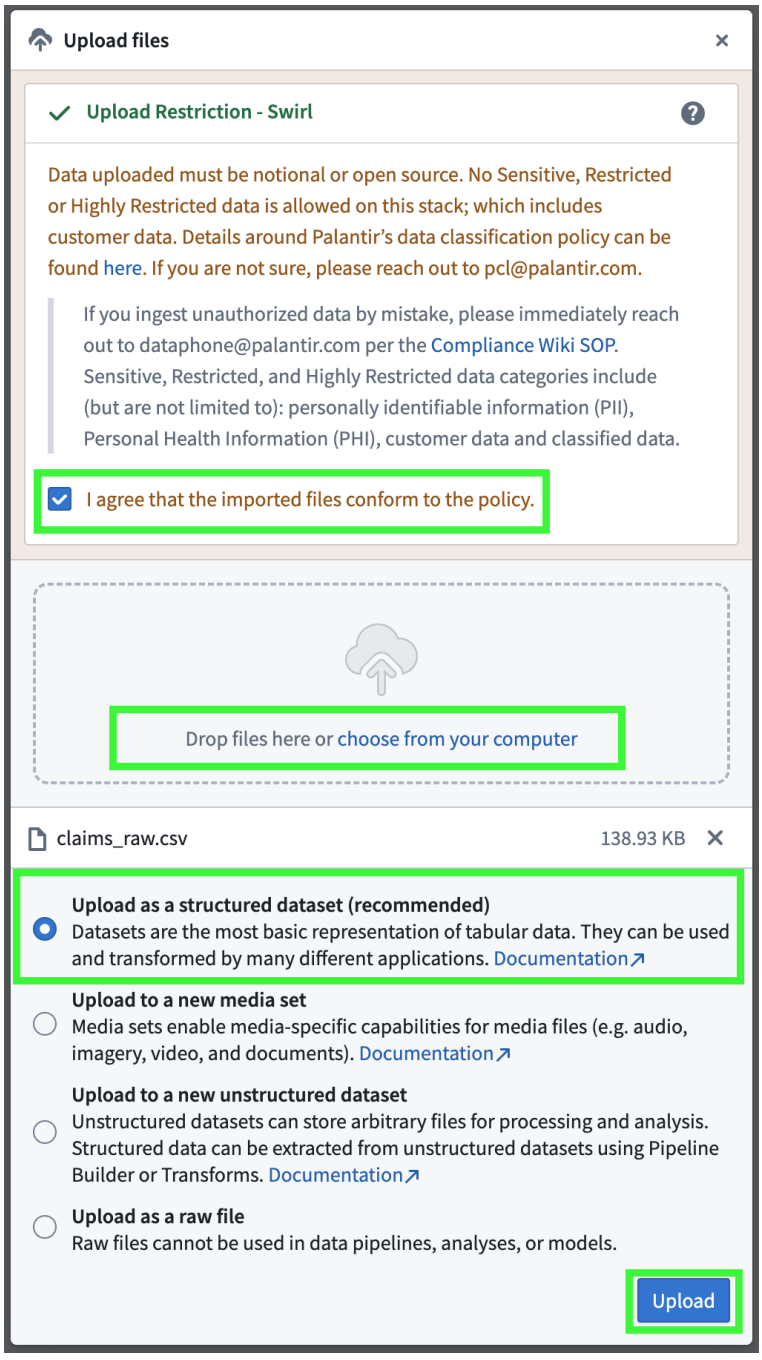
    3. Name the folder **data**.
3. <u>Within</u> the **data** folder, repeat the previous steps:
    1. Click **+New** in your **data** folder.
    2. Choose **Folder.**
    3. Name the folder “**raw**”.

Step 2: Upload your datasets into the raw folder

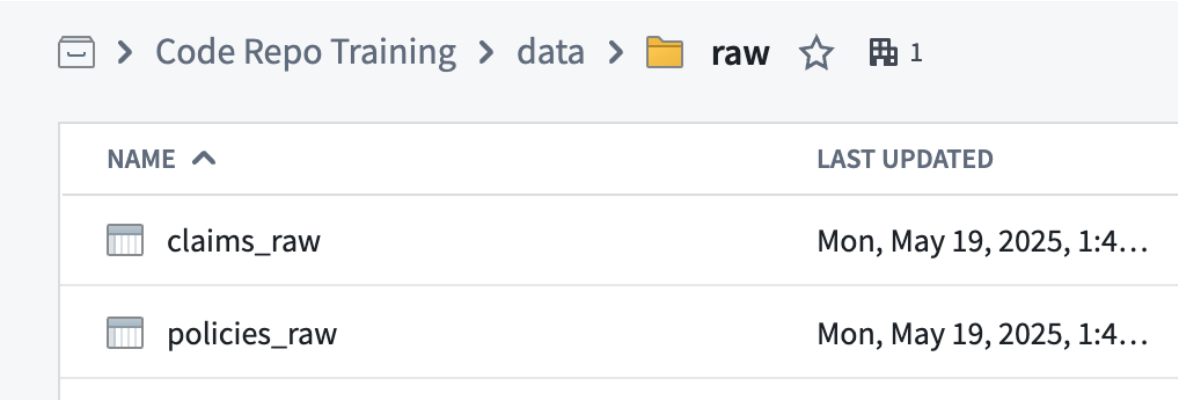
1. Open the new **raw** folder.
    1. Click **+New**.
    2. Choose “**Upload files...**” in the dropdown.



3. Choose the first file from your computer: *instructor_5lgholhrburismcba795c5iu_public_1747647595_claims_raw+(3)*
4. Rename this to **claims_raw.csv**
5. Select **“Upload as a structured dataset (recommended)”.**
6. Verify and tick confirmation checkboxes where required.
7. Click **Upload**.

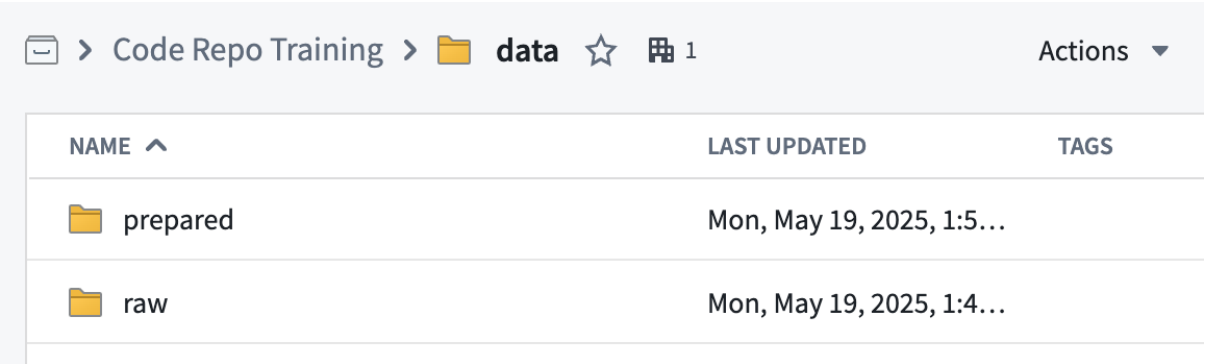


2. Repeat this process with the second file: *instructor_5lgholhrburismcba795c5iu_public_1747647748_policies_raw+(1)*
    1. In the process, rename this file to **policies_raw.csv**
3. You should now see the following two datasets in you **raw** folder.
    1. If your datasets still have the long complex name, select them and rename them to **claims_raw** and **policies_raw**, respectively.



Step 3: Create the dedicated folder for your output datasets

1. Open the **data** folder, which is the parent folder to your raw folder.
2. Within the **data** folder:
    1. Click **+New** in your project folder.
    2. Choose **Folder.**
    3. Name the folder "**prepared**".
    4. Your data folder should now contain two subfolders like shown below:

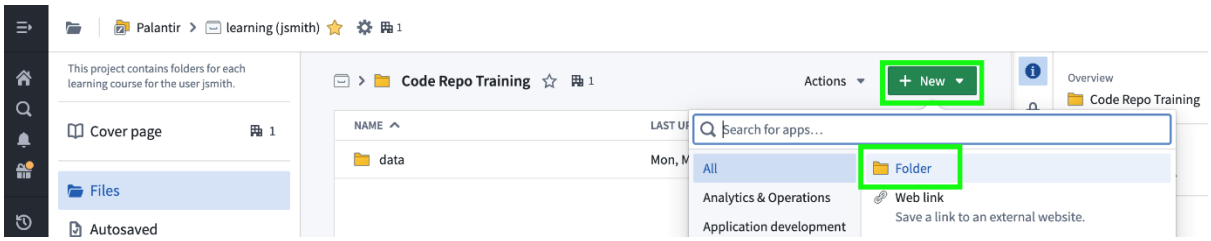


Create a New Repository

In your training folder Code Repo Training, you will first create a dedicated folder called **logic** where your Code Repository will live, followed by the Code Repository itself.

Step 1: Create a subfolder for your repository

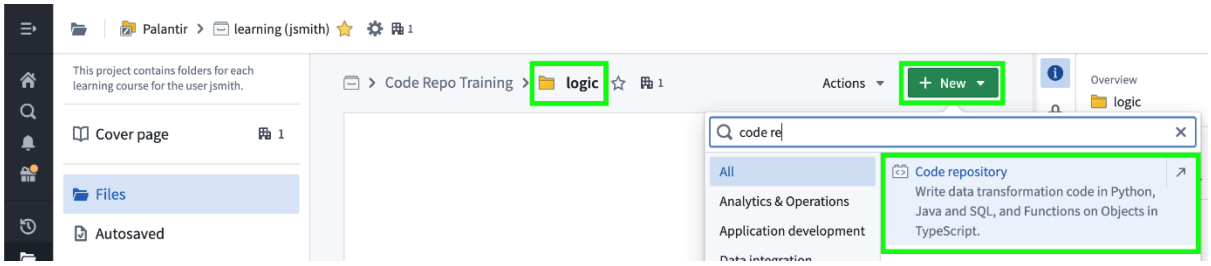
1. Navigate to your training folder. This folder should currently contain one folder called **data**.
2. Create a dedicated folder for your repo called "**logic".**
    1. click **+New** in your training folder.
    2. Choose **Folder.**



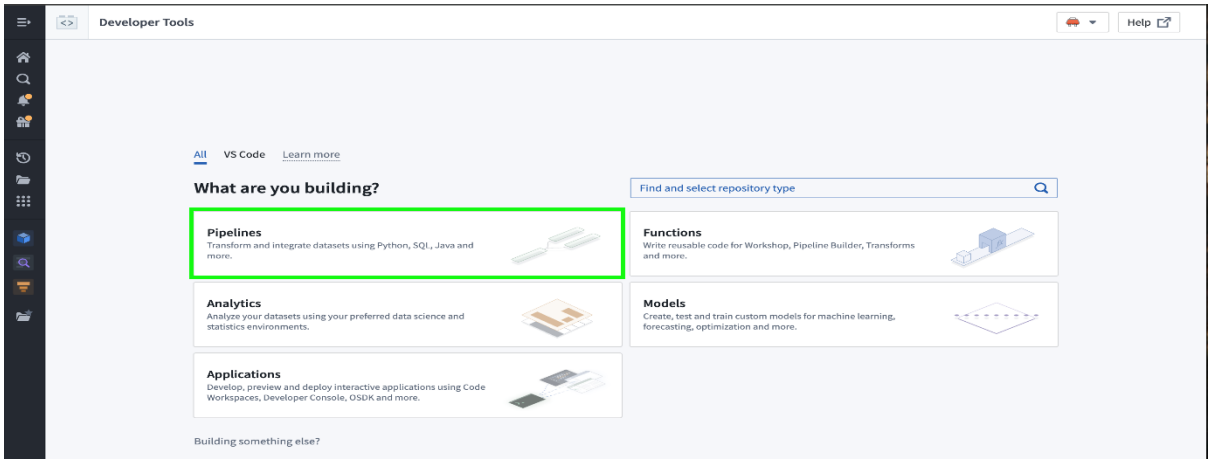
    3. Name the folder “**logic**”.
    4. Open the new **logic** folder.

Step 2: Create your repository

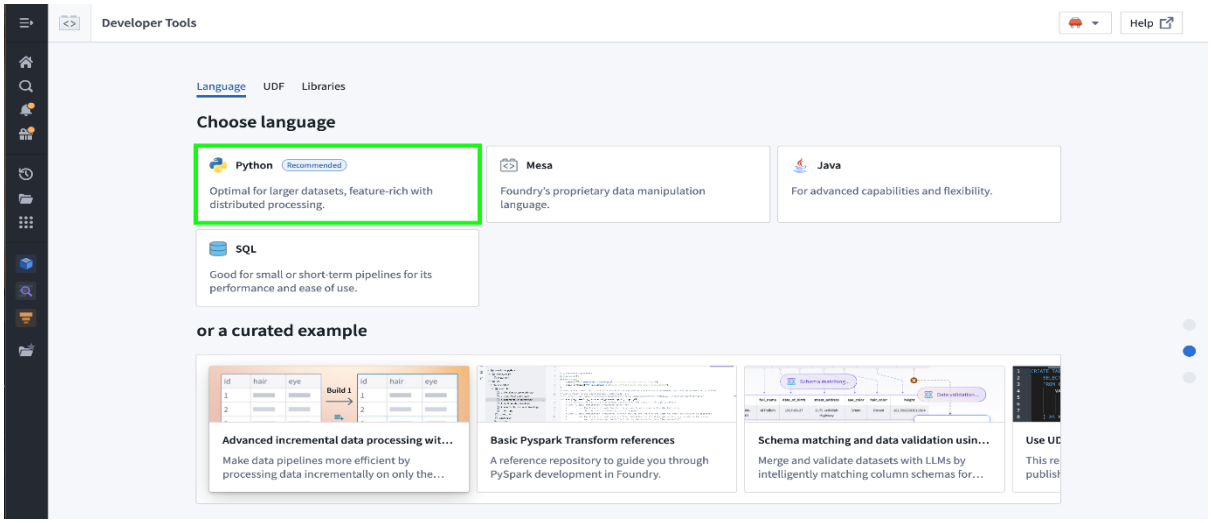
1. Click on ➕ New and choose **Code Repository** from the drop-down list.



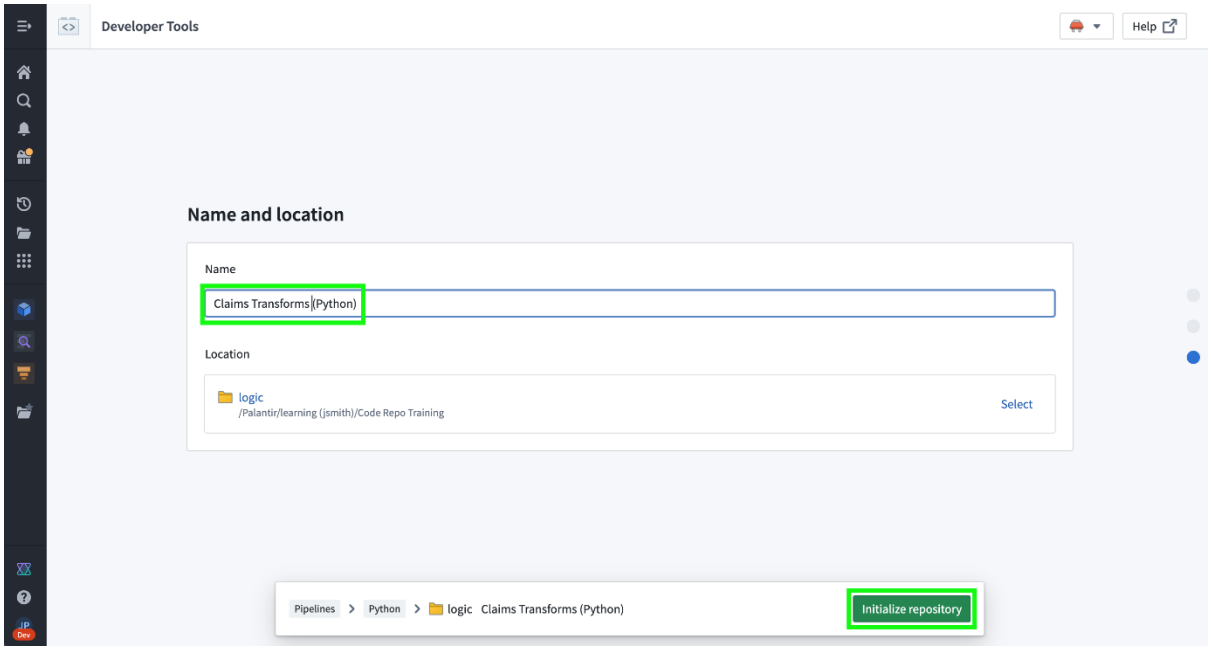
2. Choose **Pipelines**.



3. Choose **Python.**

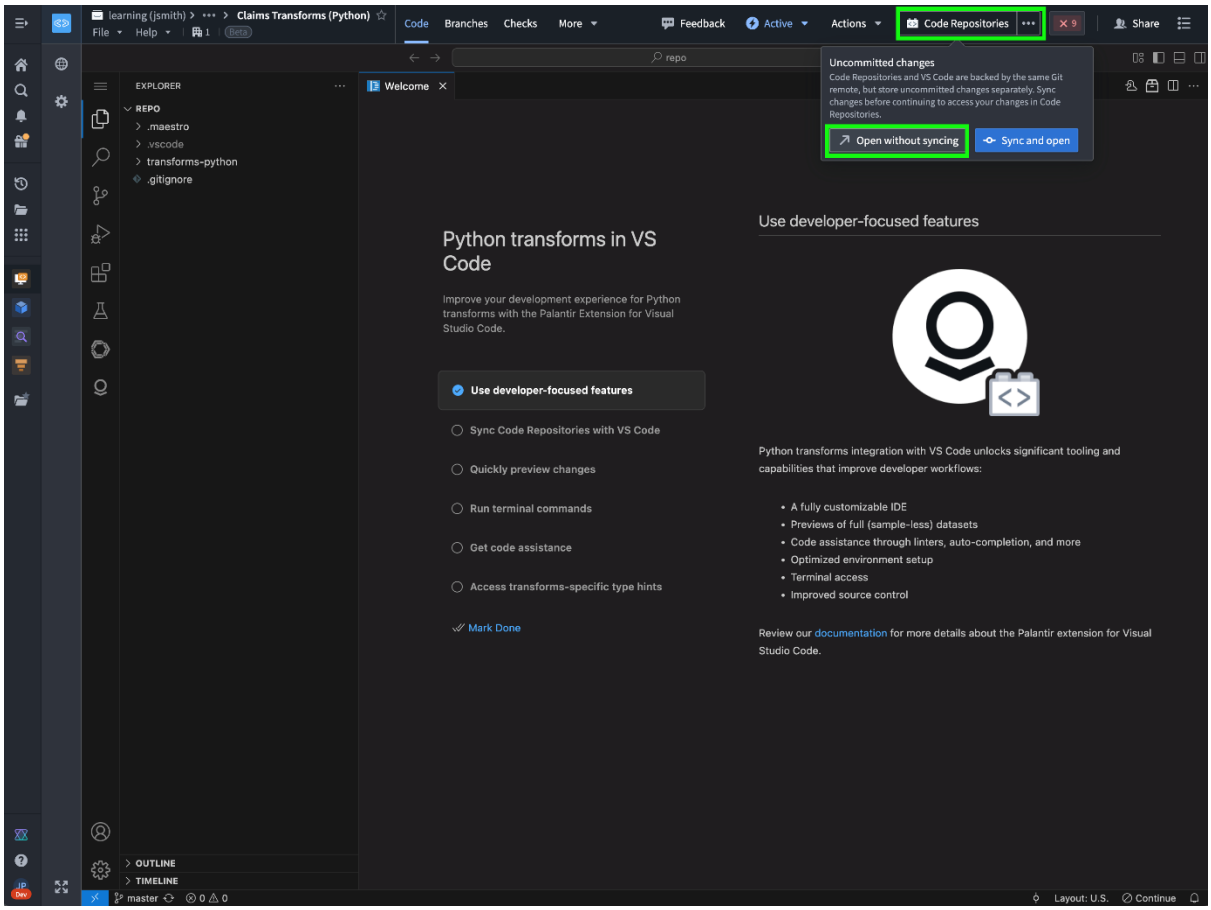


4. Name your repository Claims Transforms (Python).
5. Leave the location as is (this should be your logic folder created above).
6. Click **Initialize repository**.

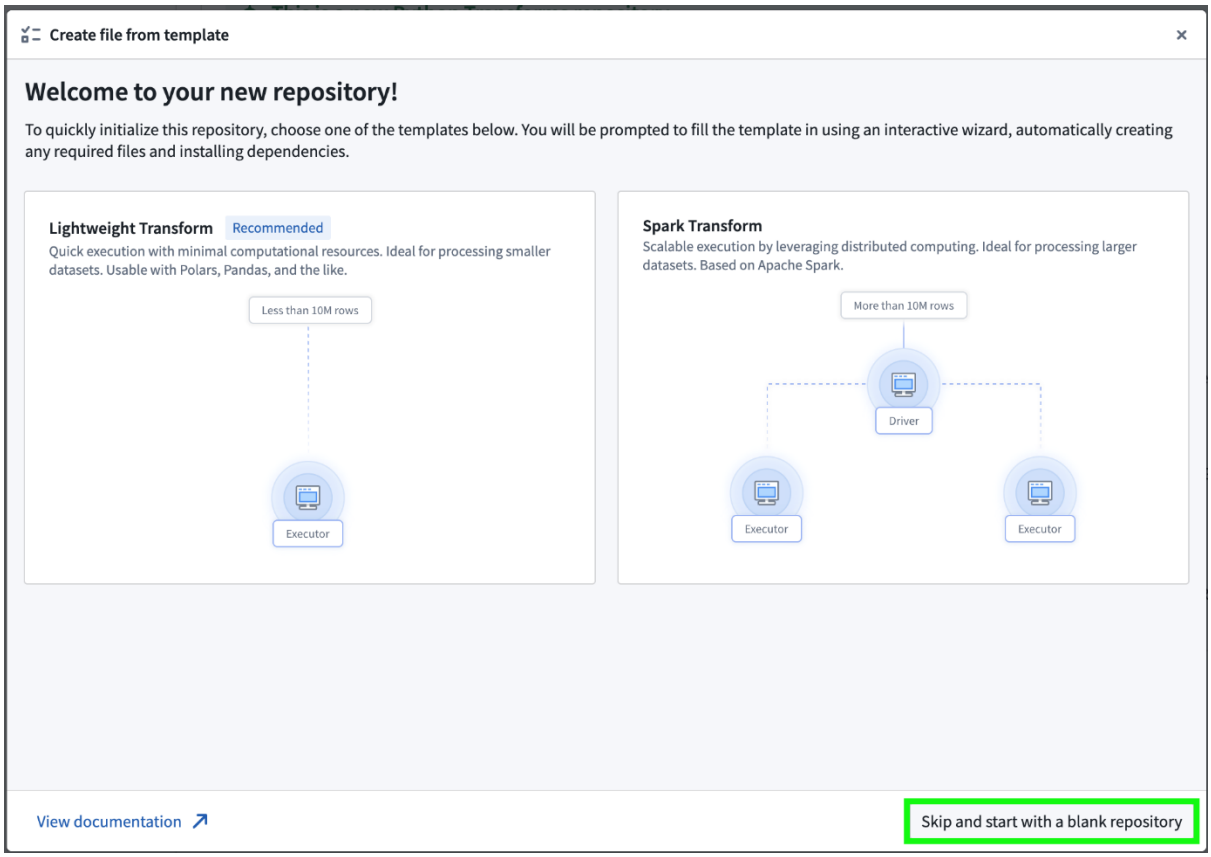


7. Sometimes, at this point, Foundry informs you about Code Repositories news. You can ignore this here and close a potential dialogue.

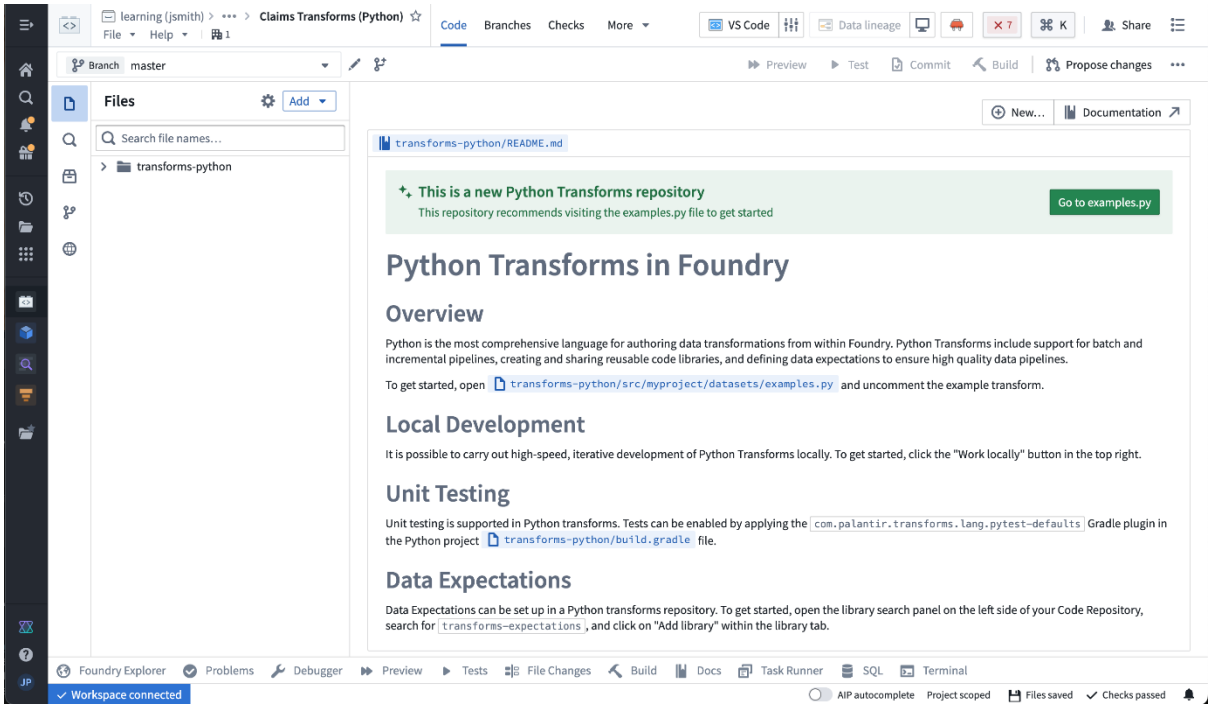
8. Depending on your Foundry instance, you might now land on the following Code Workspaces screen. If you do, click on **Code Repositories** in the top and then **Open without syncing**.



9. Click **Skip and start with a blank repository** in the next screen:



Your code repository is now set up and open.

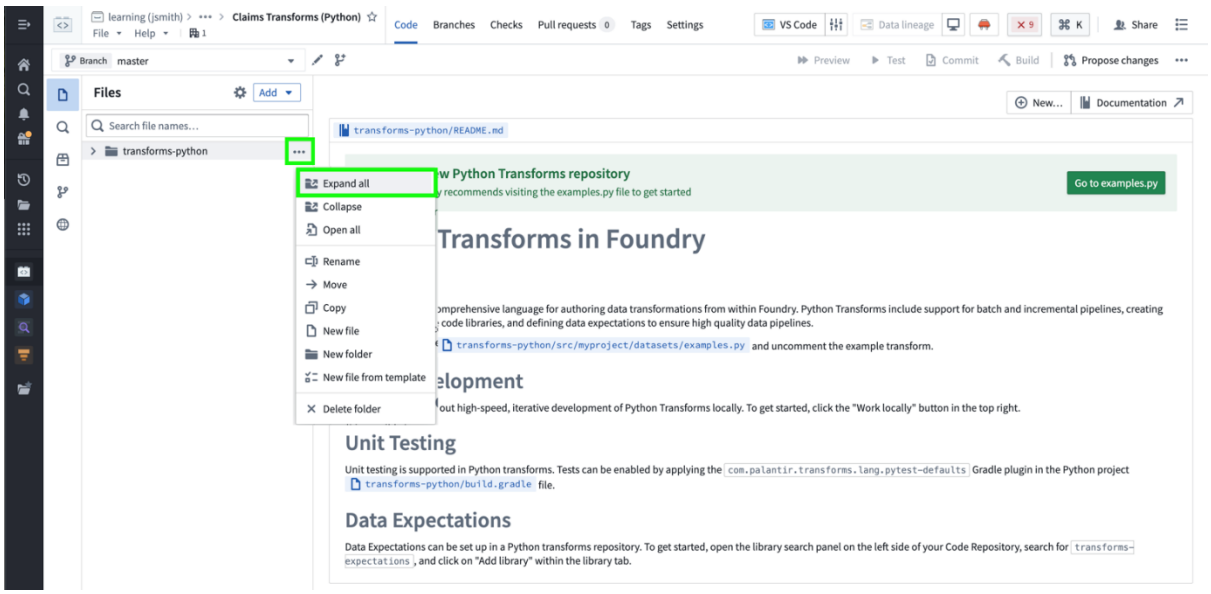


Create Your First Transform

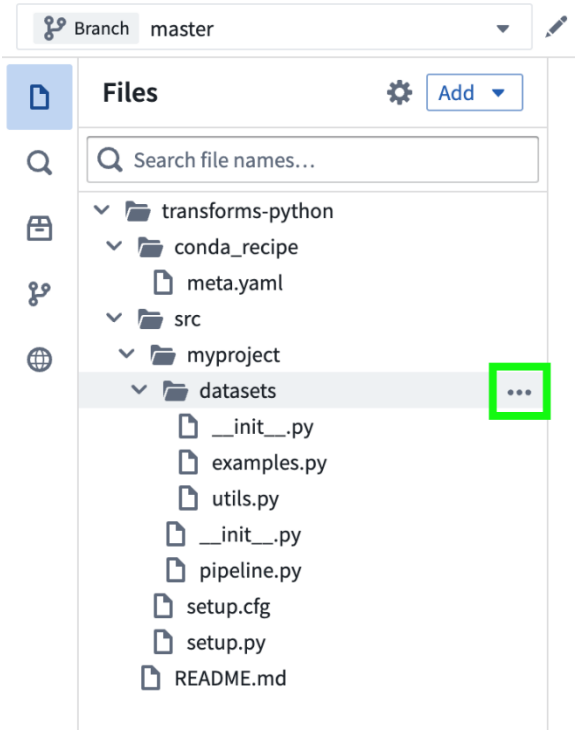
In this step, you will create your first data transformation using Code Repositories. You will add the claims_raw dataset as your input and create an identity transform. You will preview your dataset at the end of this lesson.

Step 1: Create a new Code Repository file

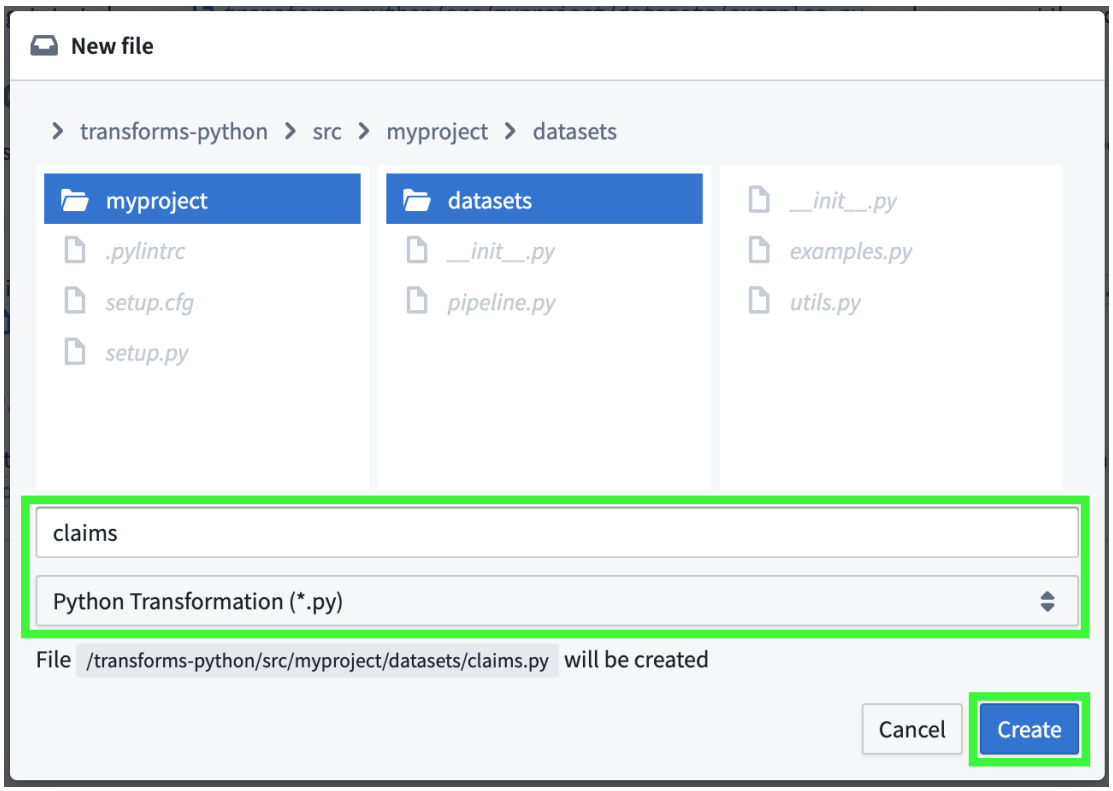
1. If the file structure on the left hand side is not yet expanded, hover over the **transforms-python** line, click the **three dots** and choose **Expand all**.



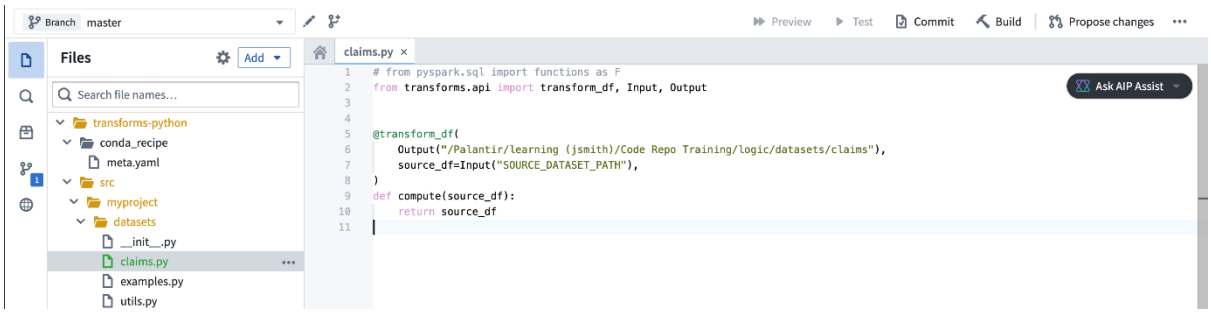
2. Hover over the **datasets** line in the file directory
3. Click the **three dots**.



4. Select **New File**.
5. Name the file “claims”.
6. Choose the **Python Transformation(*.py)** ending.
7. Click **Create**.



Code Repository will have created a simple Python file with an example input and output file.



Step 2: Define your input and output datasets

Below is a comparison of your code before and after the instructions laid out in this step.

Before this step:

from pyspark.sql import functions as F

from transforms.api import transform_df, Input, Output

@transform_df(

Output("/Palantir/learning (jsmith)/Code Repo Training/logic/datasets/claims"),

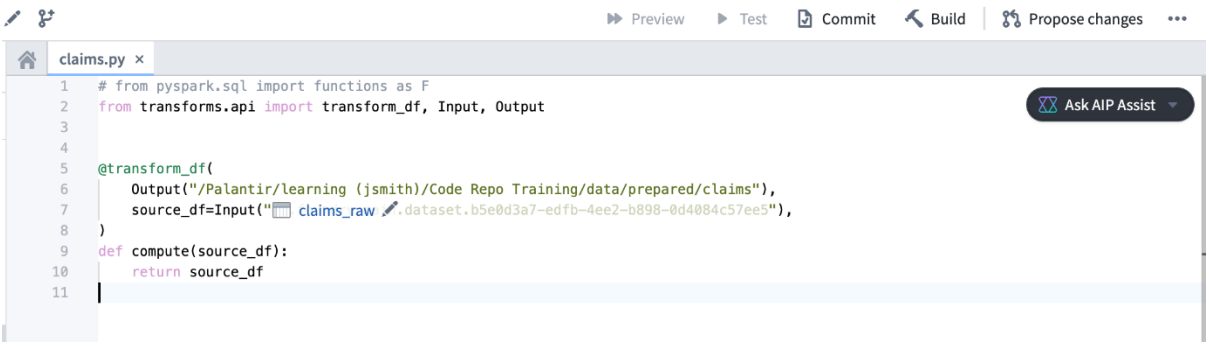
source_df=Input("SOURCE_DATASET_PATH"),

)

def compute(source_df):

return source_df

After this step:



from pyspark.sql import functions as F

from transforms.api import transform_df, Input, Output

@transform_df(

Output("/Palantir/learning (jsmith)/Code Repo Training/data/prepared/claims"),

source_df=Input("/Palantir/learning (jsmith)/Code Repo Training/data/raw/claims_raw"),

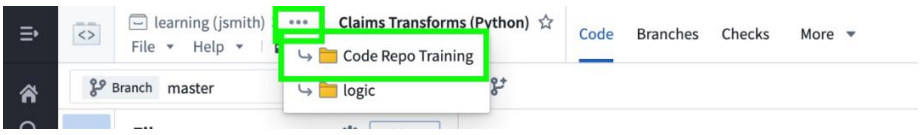
)

def compute(source_df):

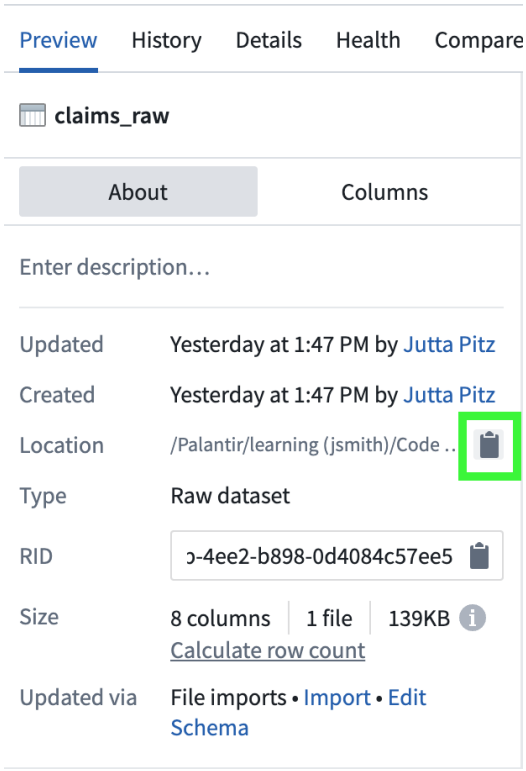
return source_df

1. First, we need to define the **input dataset** in our file.
    - In line 7, replace SOURCE_DATASET_PATH with the path of your claims input dataset, previously saved in your raw folder.

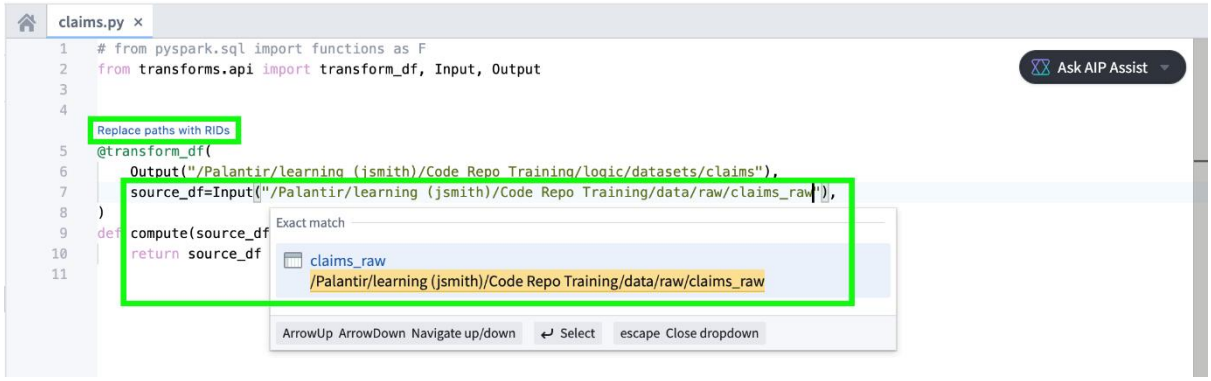
        1. Open the **claims_raw** dataset in a new tab
            1. Hold cmd/ctrl and click the **Code Repo Training** folder in your file path at the top



            2. Click on **data**
            3. Click on **raw**
            4. Click on **claims_raw**
        2. copy the **Location**

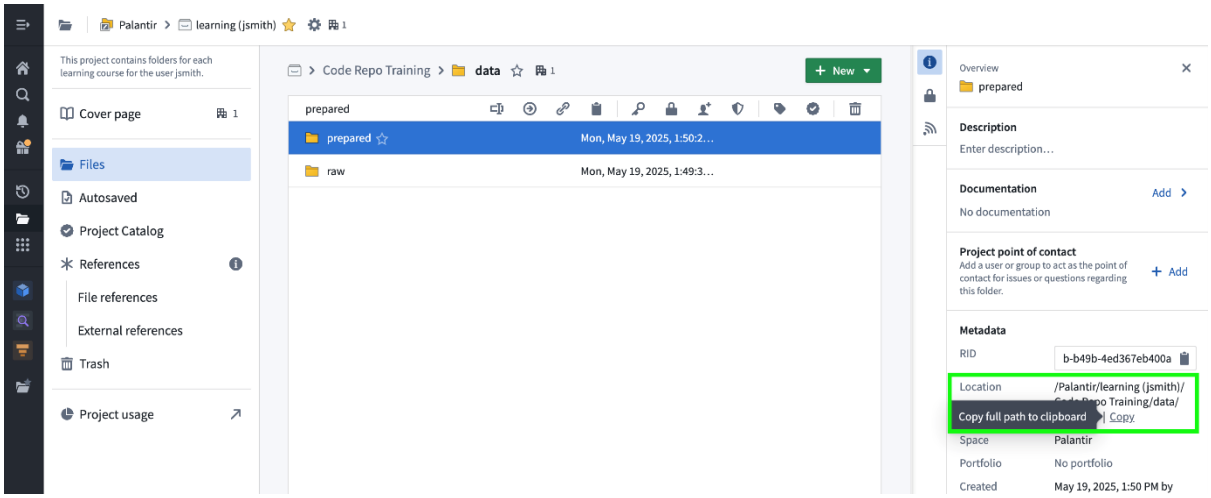


3. paste the path in your code as INPUT

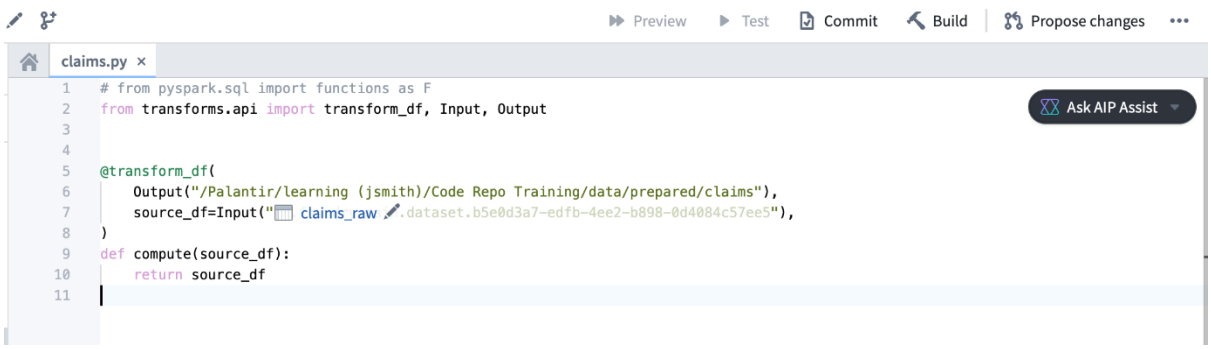


4. Foundry suggests to you the correct file to select
5. Click **Replace paths with RIDs** if this shows.

2. Second, we define our output dataset.
    - Copy the path to your **prepared** folder which you created above.

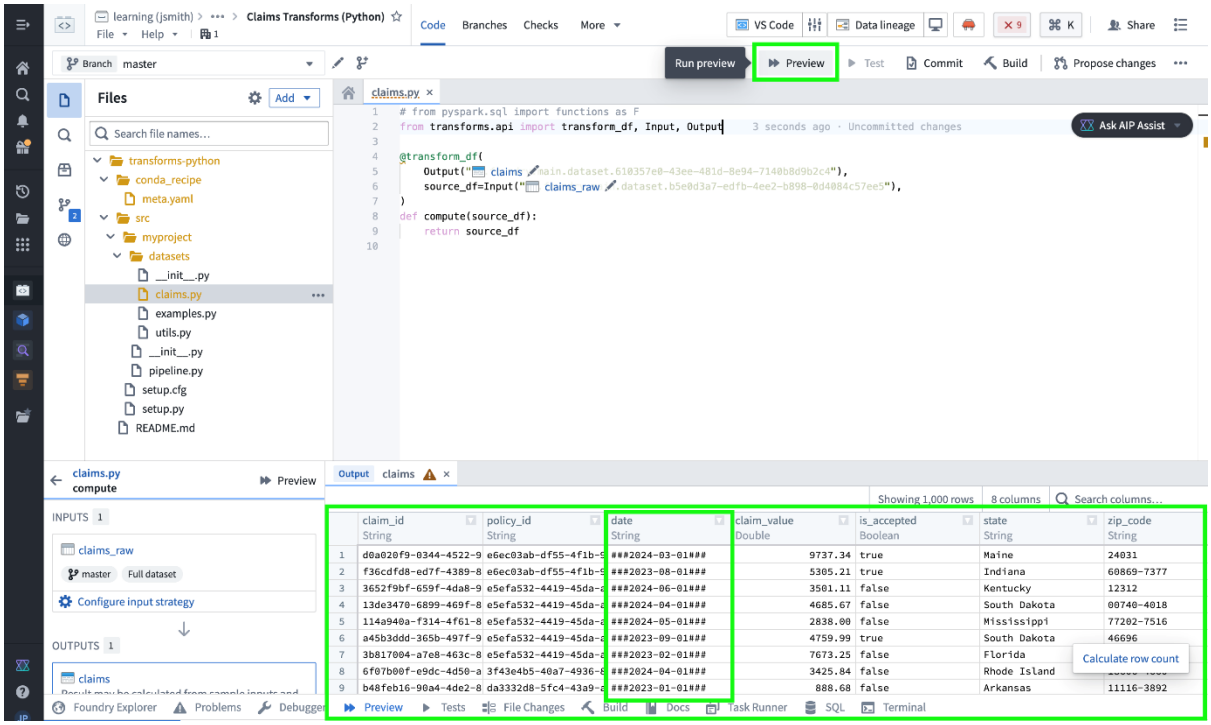


- In line 6 of your claims.py file, paste the copied path in the Output(“”) brackets inbetween the quotes “”
- Extend the path by a /claims. This will be your newly created dataset originating from this transform.
- Your code file should currently look as follows:



Step 3: Preview your dataset

1. Click the **>> Preview** button in the top bar to view your output dataset.
2. Preview produces a sample output without committing changes, running checks, or materializing any datasets in Foundry. Computing preview of output may take a few moments.



3. Your output dataset is currently simply a copy of your input dataset, because - as you can see in the code - the compute function currently returns the source as a result. We will change this to a more meaningful transformation in the next steps

4. Please note that we currently have a data quality issue in our input dataset: the values in the '**date**' column are surrounded by '###' and hence cannot be stored as a date type. We'll clean this in the next lesson.

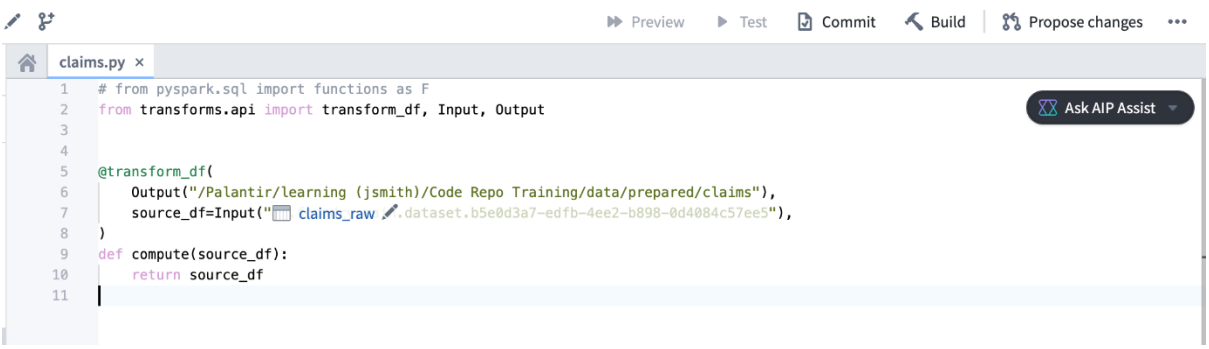
Clean Your Dataset

In this lesson, you will clean your input dataset by applying a simple cast and filter transformation and output a transformed dataset.

As a first transformation, we want to cast the date column from **String** to **Date**. In addition, we will filter by the Boolean column **is_accepted**.

Here is a comparison of what your code looks like right now and once you have run through these instructions.

<u>Before applying these instructions:</u>



from pyspark.sql import functions as F

from transforms.api import transform_df, Input, Output

@transform_df(

Output("ri.foundry.main.dataset.610357e0-43ee-481d-8e94-7140b8d9b2c4"),

source_df=Input("ri.foundry.main.dataset.b5e0d3a7-edfb-4ee2-b898-0d4084c57ee5"),

)

def compute(source_df):

return source_df

<u>After applying the following instructions:</u>

from pyspark.sql import functions as F

from pyspark.sql import types as T

from transforms.api import transform_df, Input, Output

```
@transform_df(
    Output("ri.foundry.main.dataset.610357e0-43ee-481d-8e94-7140b8d9b2c4"),
    claims_raw=Input("ri.foundry.main.dataset.b5e0d3a7-edfb-4ee2-b898-0d4084c57ee5"),
)
def compute(claims_raw):
    # Cast the "date" column from string to date
    claims = claims_raw.withColumn("date", F.regexp_replace("date", "###", "").cast(T.DateType()))

    # Filter out declined claims
    claims = claims.filter(F.col("is_accepted") == True)
    return claims
```

Step 1: Import functions and types

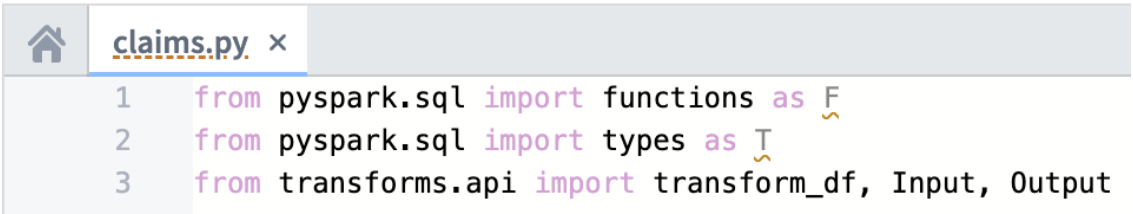
1. Uncomment the first line of your code repository, i.e. remove the “#” in line 1. This makes the Pyspark SQL library available for us to use in our transformation code.

```
# from pyspark.sql import functions as F
```

2. Include the following additional line of code right underneath the code above:

```
from pyspark.sql import types as T
```

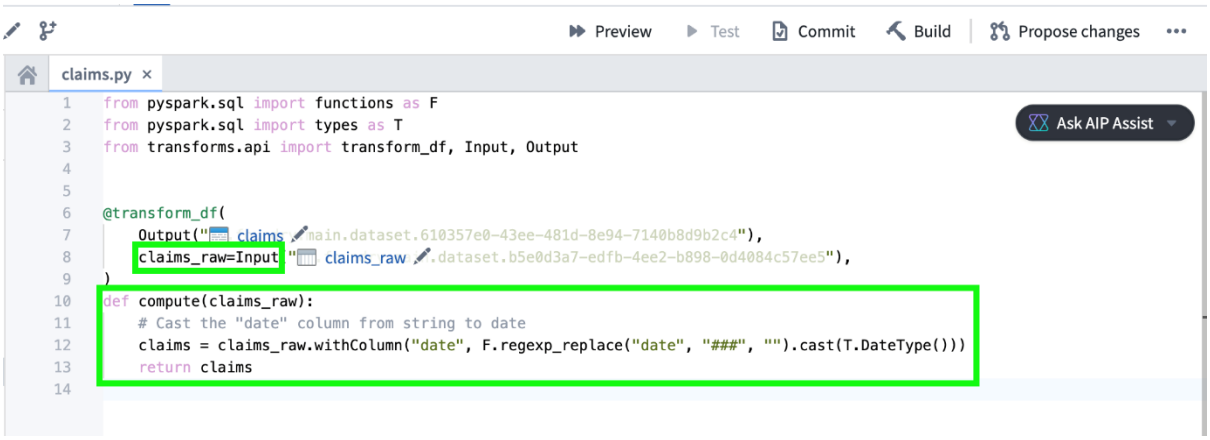
3. Your first three lines of code should now look like this:



Step 2: Cast the previous string column to a date column

We want to clean up the date column by casting the string to a Date. We will use pyspark functions such as withColumn or cast. See the official Spark documentation for context and an overview of which functions are available.

Modify the following sections in the code file. You can copy and paste the code given in the steps below:



1. In line 8, replace source_df with claims_raw.
2. The previous lines 10 and 11 become:

def compute(claims_raw):

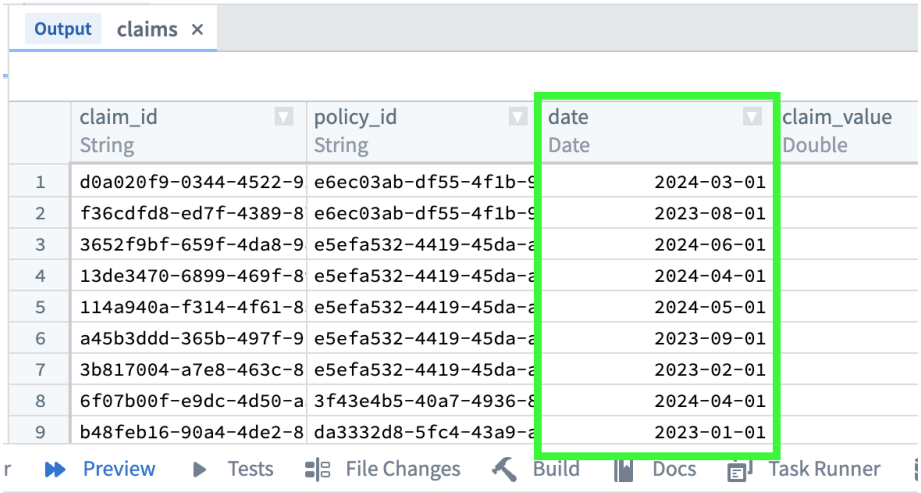
Cast the "date" column from string to date

claims = claims_raw.withColumn("date", F.regexp_replace("date", "###", "").cast(T.DateType()))

return claims

3. Preview your results to check whether this has worked as expected and that your date column is now of type **Date** as opposed to type **String**.

    - Click **Preview** at the top of your Code Repository, as done before.
    - Once the table shows, verify that the **date** column is indeed of type **Date** and that the trailing ###s have been removed.



Step 3: Filter your dataset

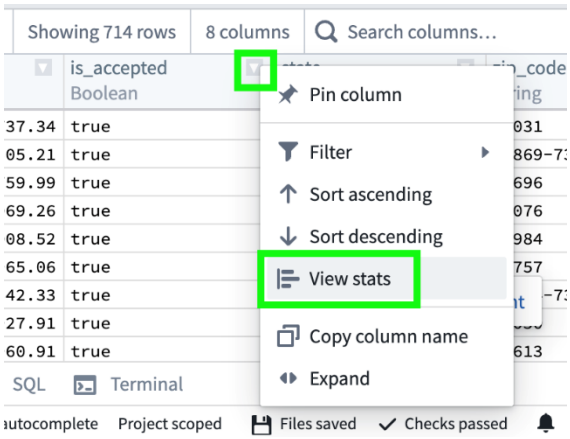
Next, we want to filter the claims dataset to only those claims that have been accepted for payout.

1. Modify the Python function in the **claims.py** file as follows:
    - Extend the previous code in lines 10-13 to this code:

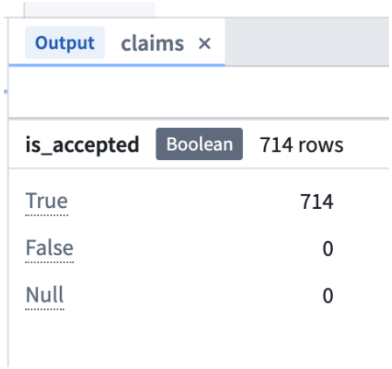
```
def compute(claims_raw):
    # Cast the "date" column from string to date
    claims = claims_raw.withColumn("date", F.regexp_replace("date", "###", "").cast(T.DateType()))

    # Filter out declined claims
    claims = claims.filter(F.col("is_accepted") == True)
    return claims
```

2. Verify your results using Preview again
    - Click the **Preview** button in the top, just like previously
    - Confirm that only rows with *is_accepted* being *true* remain
        - Navigate to the *is_accepted* column in the Preview table at the bottom
        - Click the little triangle in the column header
        - Choose **View stats**



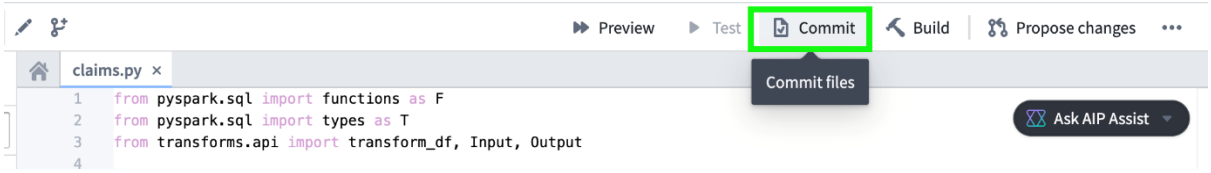
- Confirm there is only **True** values appearing



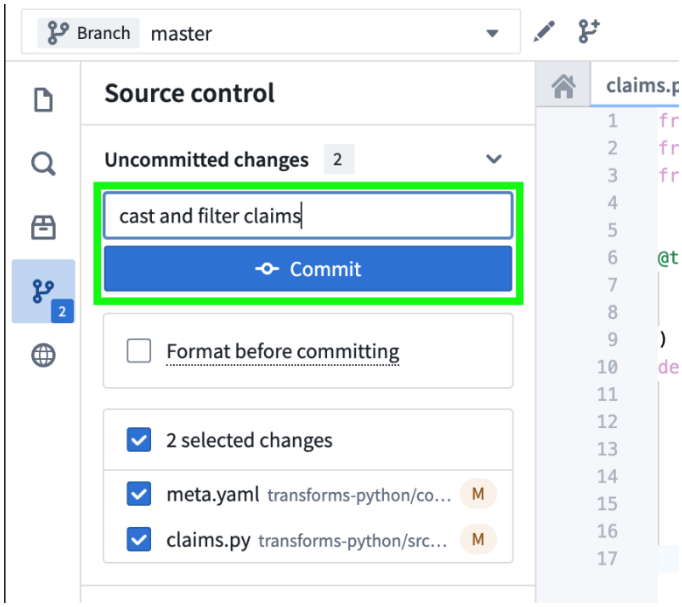
Step 4: Commit your work to save it

Commits will trigger checks which help us keep your code correct and performant.

1. Click **Commit** in the top right corner



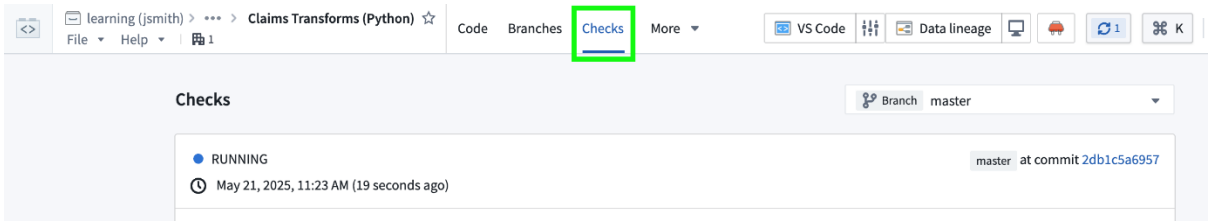
2. Input a commit message, e.g. “cast and filter claims”



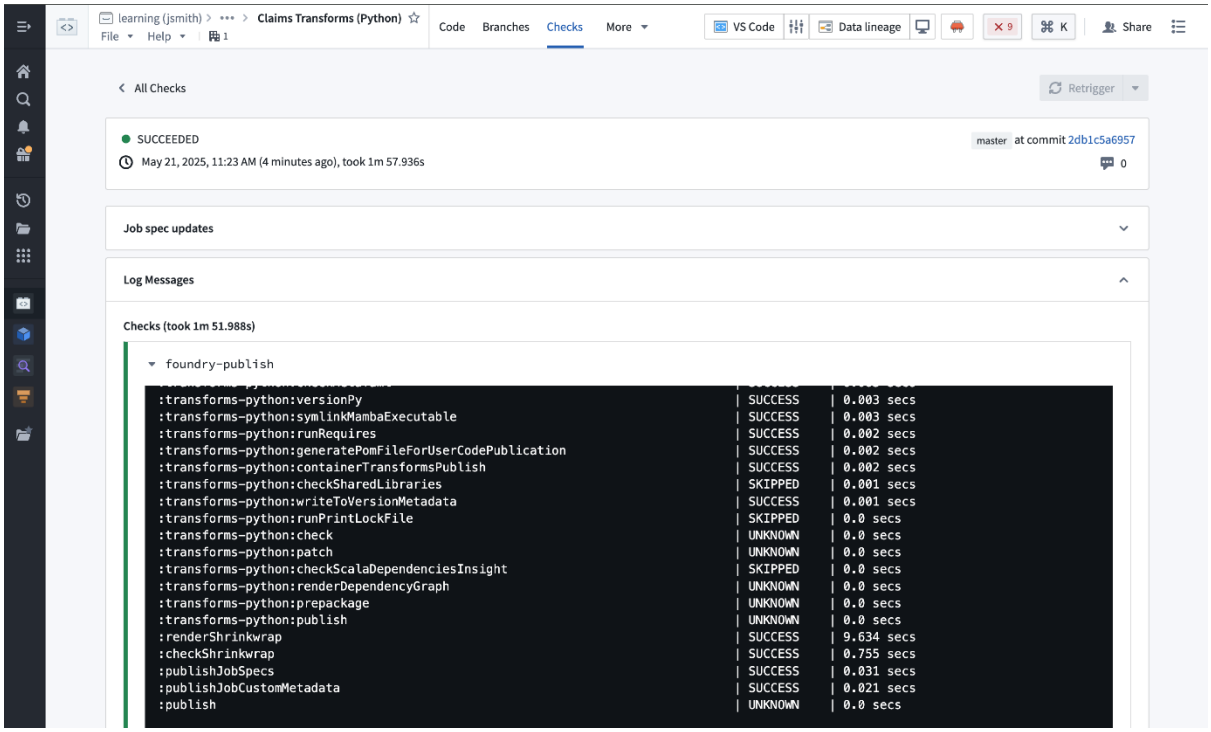
3. click **Commit**

This will trigger checks, which you can see as follows:

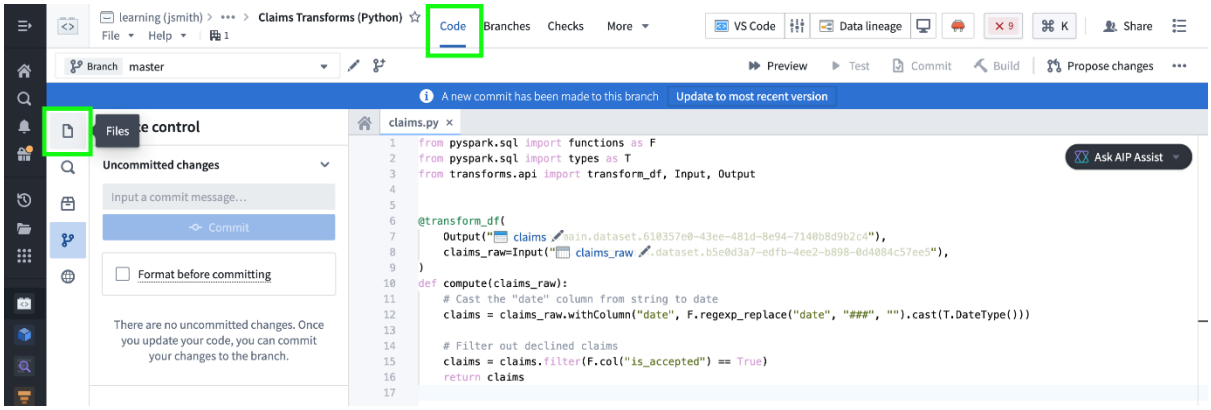
1. Click the **Checks** tab in the top of your screen



2. Open the first Check to see its log messages



3. Click the **Code** tab above and then the files icon in the left sidebar to return to your files.



Optional: if you see the blue bar with the message "A new commit has been made to this branch", click **Update to most recent version.**

Join in Another Dataset

In preparing the financial overview for the CFO, you need to add the lines of business for each claim to your dataset. This information is contained in the second dataset we uploaded previously, **policies_raw**.

We will use the JOIN operation to include the new column(s) to our existing claims dataset. We perform this transformation in the same code file which we used for our cast and filter operation.

Below you see the difference between the code file as is and post applying these changes.

Before applying these instructions:

from pyspark.sql import functions as F

from pyspark.sql import types as T

from transforms.api import transform_df, Input, Output

@transform_df(

Output("ri.foundry.main.dataset.610357e0-43ee-481d-8e94-7140b8d9b2c4"),

claims_raw=Input("ri.foundry.main.dataset.b5e0d3a7-edfb-4ee2-b898-0d4084c57ee5"),

)

def compute(claims_raw):

Cast the "date" column from string to date

claims = claims_raw.withColumn("date", F.regexp_replace("date", "###", "").cast(T.DateType()))

Filter out declined claims

claims = claims.filter(F.col("is_accepted") == True)

return claims

After applying these instructions:

from pyspark.sql import functions as F

from pyspark.sql import types as T

from transforms.api import transform_df, Input, Output

@transform_df(

Output("ri.foundry.main.dataset.610357e0-43ee-481d-8e94-7140b8d9b2c4"),

claims_raw=Input("ri.foundry.main.dataset.b5e0d3a7-edfb-4ee2-b898-0d4084c57ee5"),

policies_raw=Input("ri.foundry.main.dataset.08535268-5766-41ef-b708-c092faf9f598"),

)

def compute(claims_raw, policies_raw):

Cast the "date" column from string to date

claims = claims_raw.withColumn("date", F.regexp_replace("date", "###", "").cast(T.DateType()))

Filter out declined claims

claims = claims.filter(F.col("is_accepted") == True)

Join in policies to get LOB

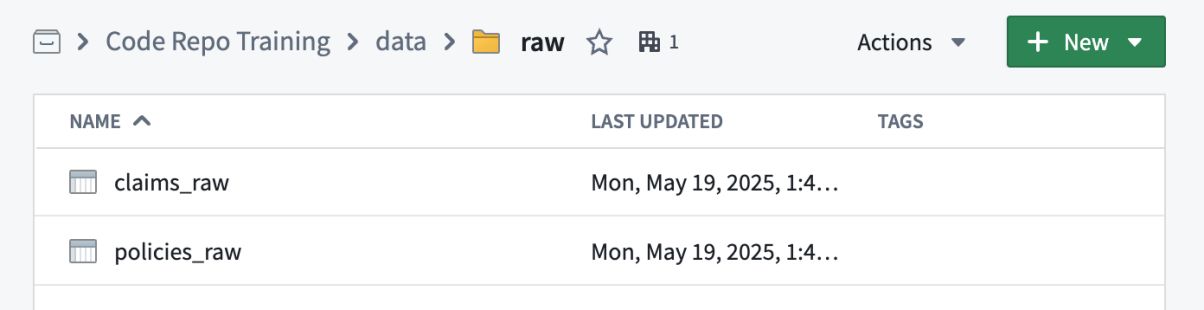
```
    claims = claims.join(
        policies_raw,
        on="policy_id",
        how="left"
    )
    return claims
```

Step 1: Add your second input dataset

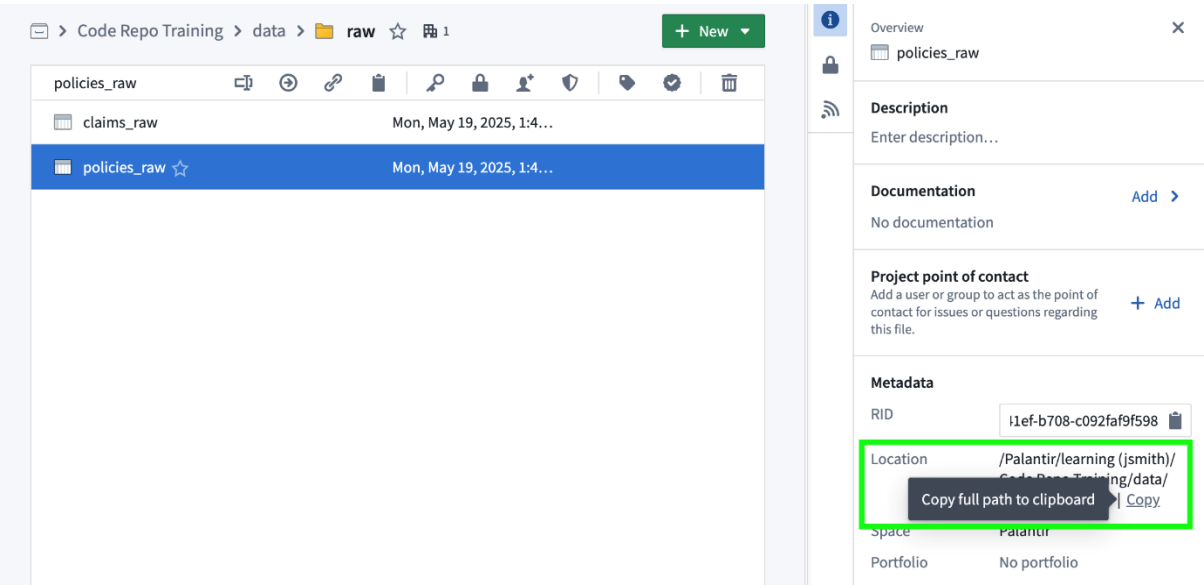
1. In between line 8 and 9, add the following second Input line following the previous one:

```
policies_raw=Input("<<path will go here>>"),
```

2. Navigate to your **policies_raw** dataset located in your **data > raw** folder.



3. Select **policies_raw** and copy its full location path (remember you can also copy the rid instead.



4. Go back to your code repository
5. Paste the path within the quotes of your new line

policies_raw=Input("path goes here"),

6. Your code file now looks like this:

Step 2: Perform a basic join operation

The two datasets to join share a column, namely **policy_id**. We will use this as the foreign key when left joining the policies to claims. Our intent is to add the line of business to the original dataset.

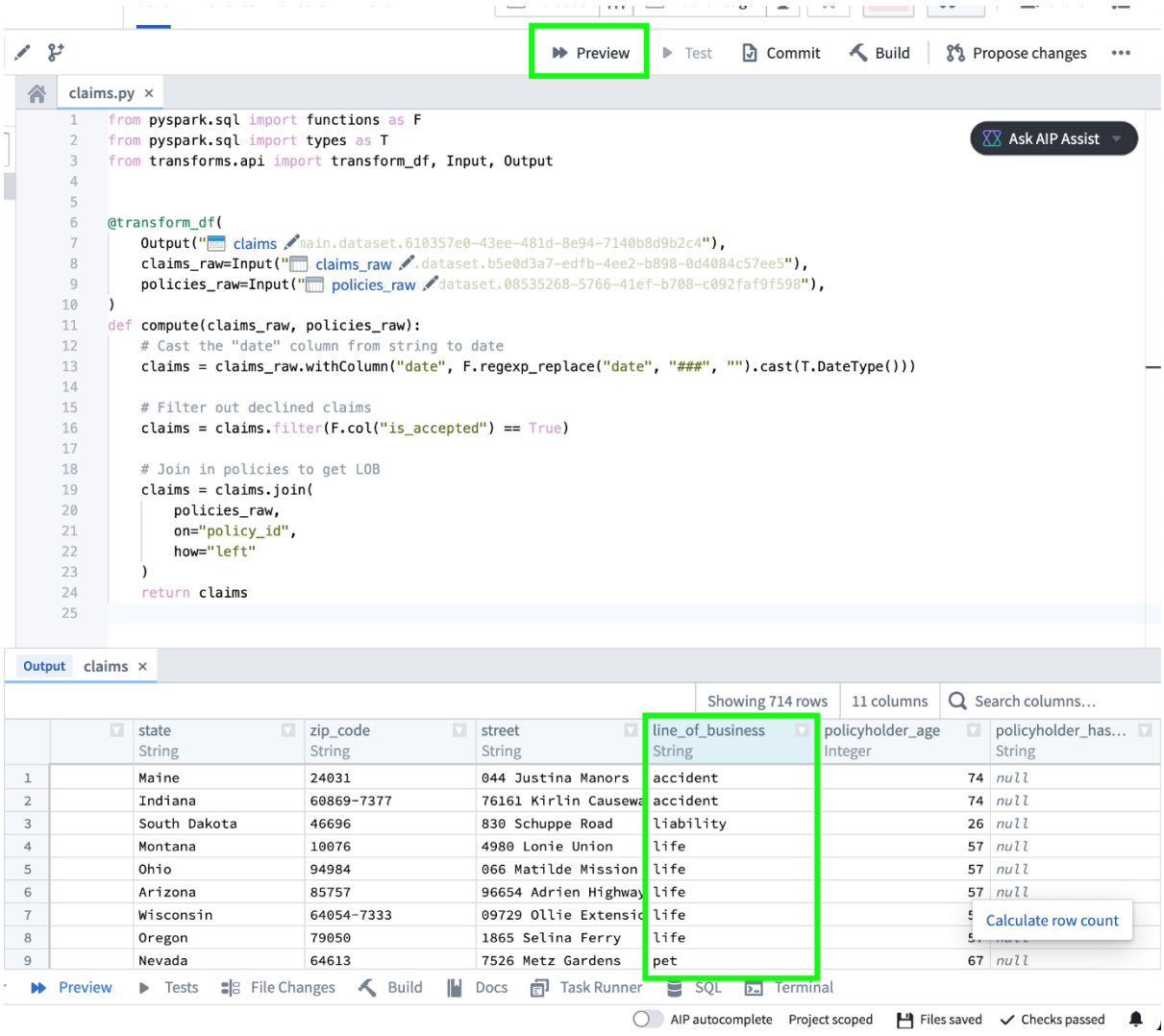
1. In line 11, first add the policies_raw as a second input to your function:

def compute(claims_raw, policies_raw):

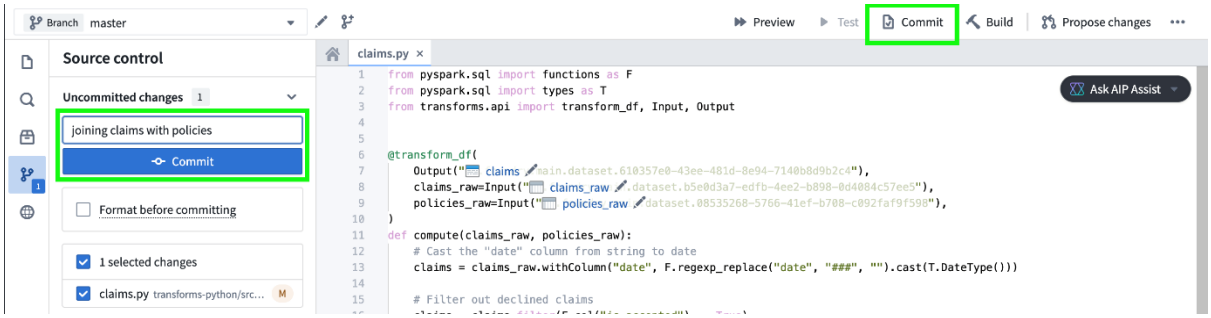
2. Below your filter for only accepted claims, i.e. below line 16 in the screenshot above, add the following code:

```
# Join in policies to get LOB
claims = claims.join(
    policies_raw,
    on="policy_id",
    how="left"
)
```

3. Your code file will now look as follows:



4. Click **Preview** to confirm that the new dataset contains the **line_of_business** column
    - Note: When using preview on larger datasets, it will typically sample a subset of the rows. If desired, the sampling strategy can be specified to ensure particular rows are included in the preview. This is of critical importance when using joins to ensure both samples have matching rows based on the join criteria.
    - Close the Preview by clicking the **Preview** button in the bottom navigation bar
5. **Commit** your work to save it, like you did previously.



6. Review your checks for any potential failures.

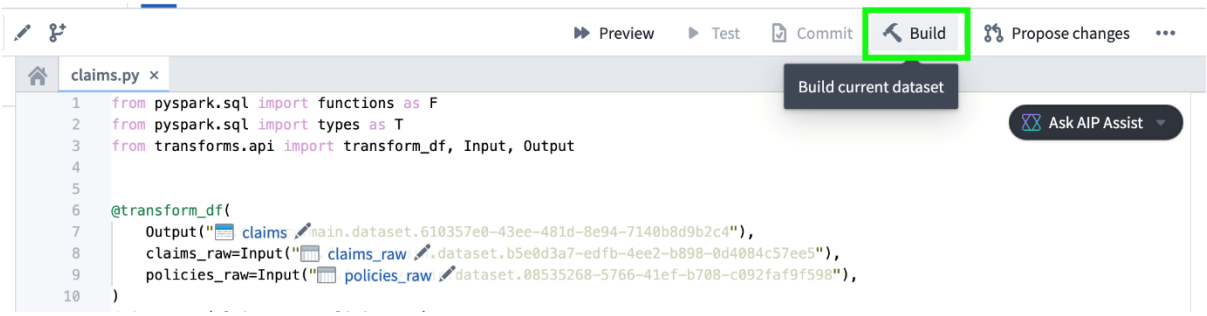
Build Your Dataset

Previously, we only previewed our results and committed them to save our code.

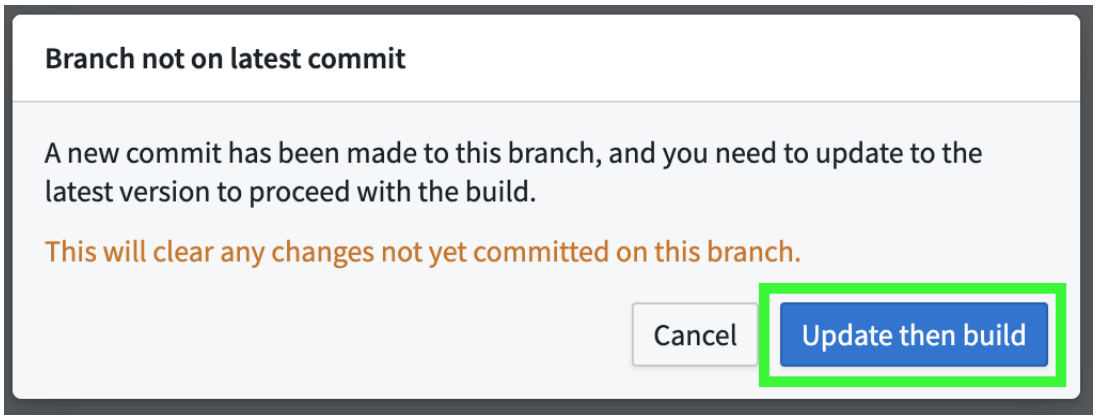
- **Preview** produces a sample output without committing changes, running checks, or materializing any datasets in Foundry.
- **Committing** saves our work and triggers Checks, but it doesn't build the defined outputs as datasets.
- **Build** actually builds the dataset(s) which you define as your output in the code file(s). It also commits the latest code changes and runs checks as part of this.

---

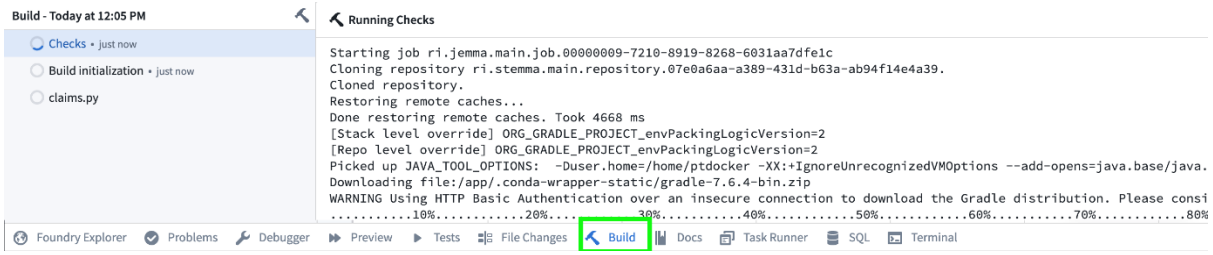
1. Click the **Build** button



2. If you get the following message, click **Update then build.**



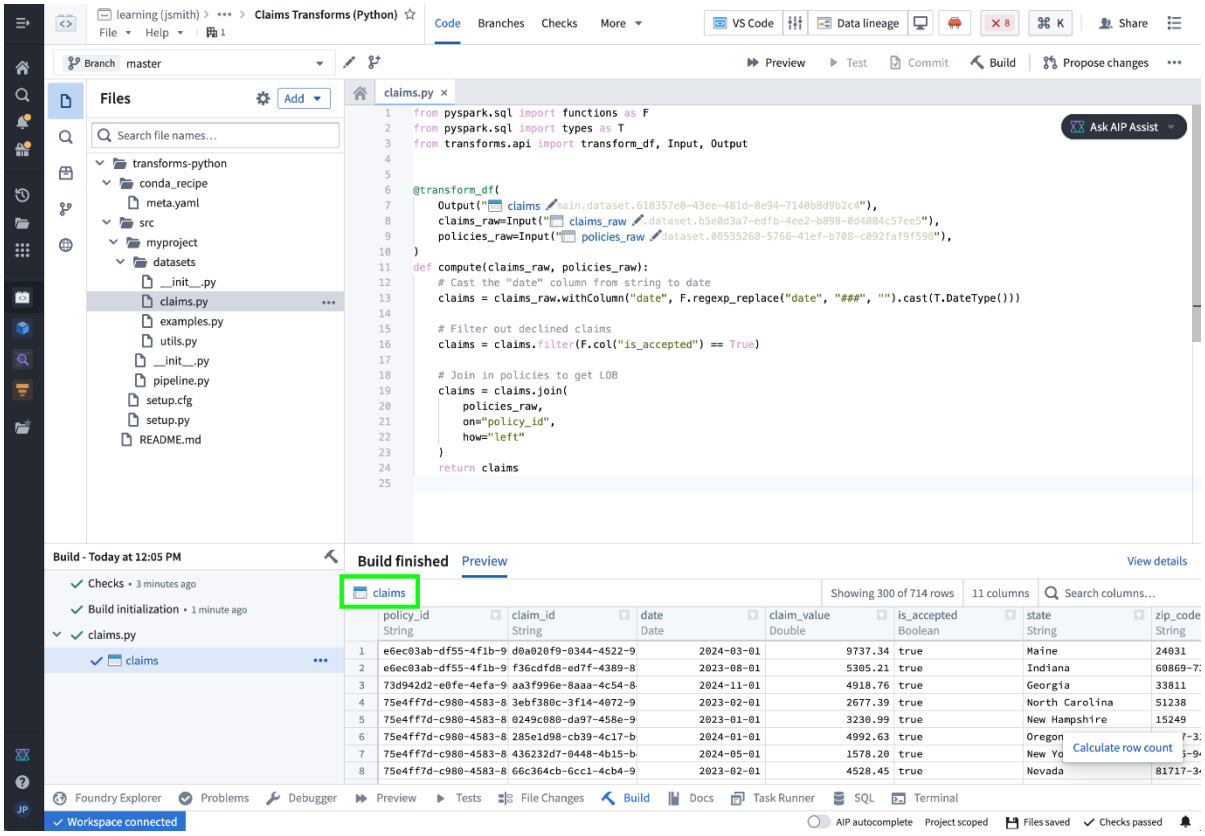
3. Building your dataset will kick off Checks automatically which you see in the bottom part of your screen.



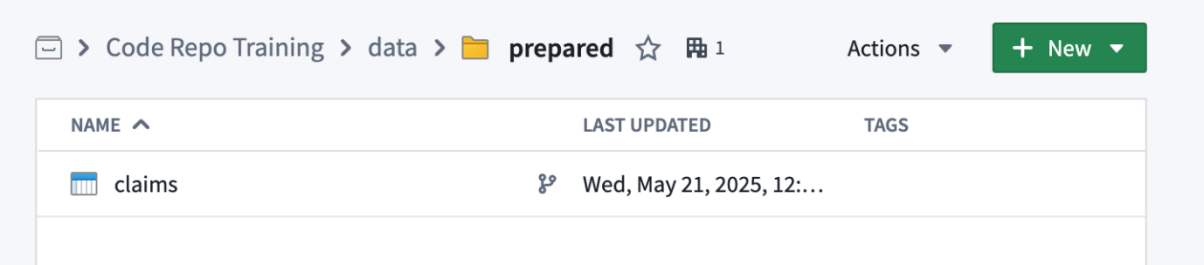
-
    - You can toggle the Build details on and off by clicking the **Build** tab in the bottom navigation bar.
    - The build might take a couple of minutes to complete.

4. Once the build is finished, it shows the preview of the dataset in the bottom of your code repository.

5. Open the dataset by clicking on "**claims**" just above the preview table, ideally holding ctrl/cmd to open it in a new tab.



6. Verify that the dataset now exists in the **prepared** folder.



Congratulations. You have built your first dataset using Code Repositories.

Collaboration via Branching

Introduction

Branching in Foundry Code Repositories is a powerful feature that allows you to create independent lines of development within your project, enabling multiple team members to work on different tasks simultaneously without interfering with the main codebase. By creating branches, you can experiment with new features, perform bug fixes, or implement enhancements in isolation, ensuring that the master branch remains stable and production-ready. This approach not only facilitates collaboration and code review processes but also supports efficient version control and project management within Foundry. Foundry branching implements an industry-standard Git-like version control paradigm. For the scope of this training, we'll be working with Code Repositories branches, as opposed to Foundry branches.

In this module, you will explore the following concepts:

- protecting your master branch,
- creating a new branch,
- perform a transform on your new branch,
- merge the changes to your master branch.

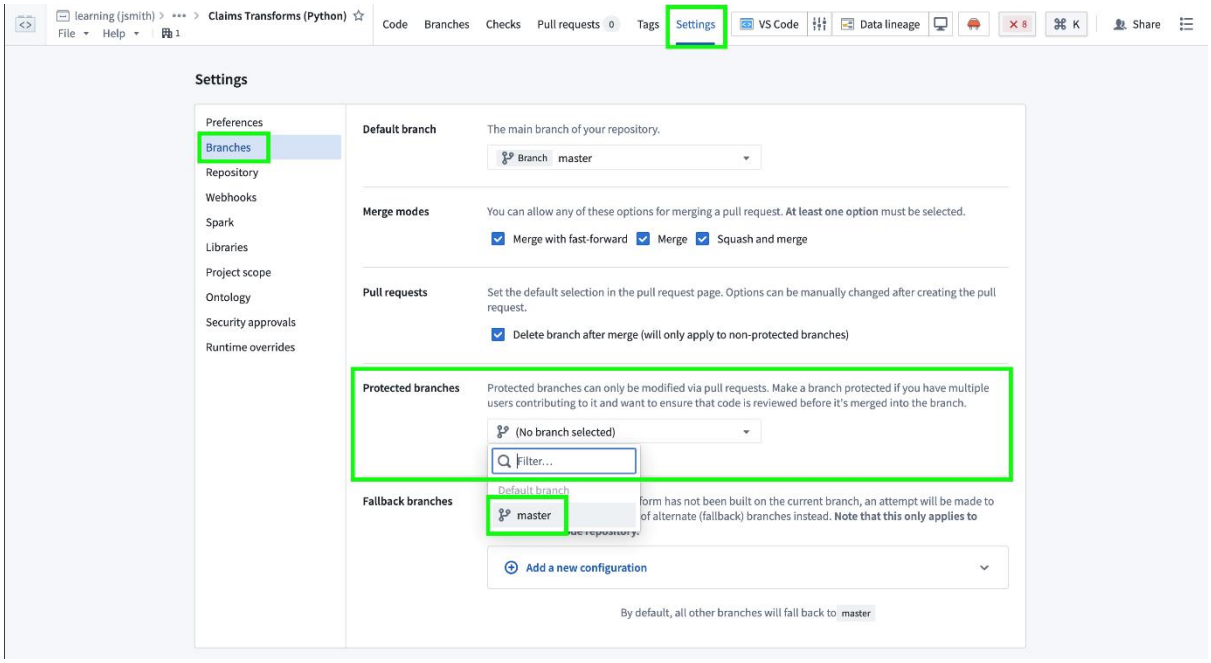
Protecting Master Branch

Protected branches can only be modified via pull requests. Make a branch protected if you have multiple users contributing to it and want to ensure that code is reviewed before it's merged into this branch.

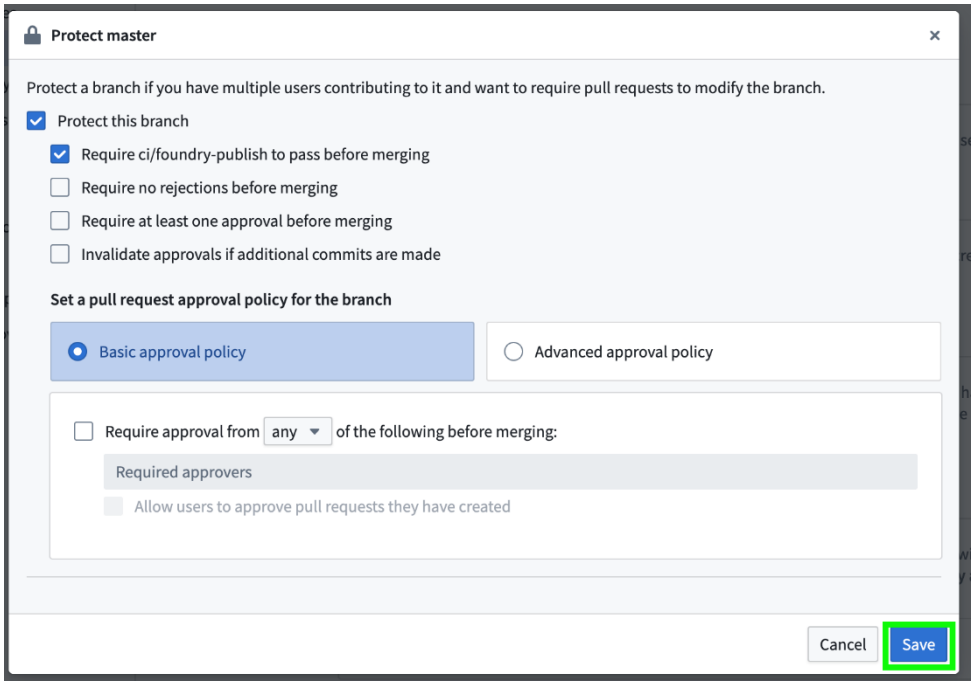
Step 1: Protect master branch

1. In your code repository, open **Settings** from the top navigation bar

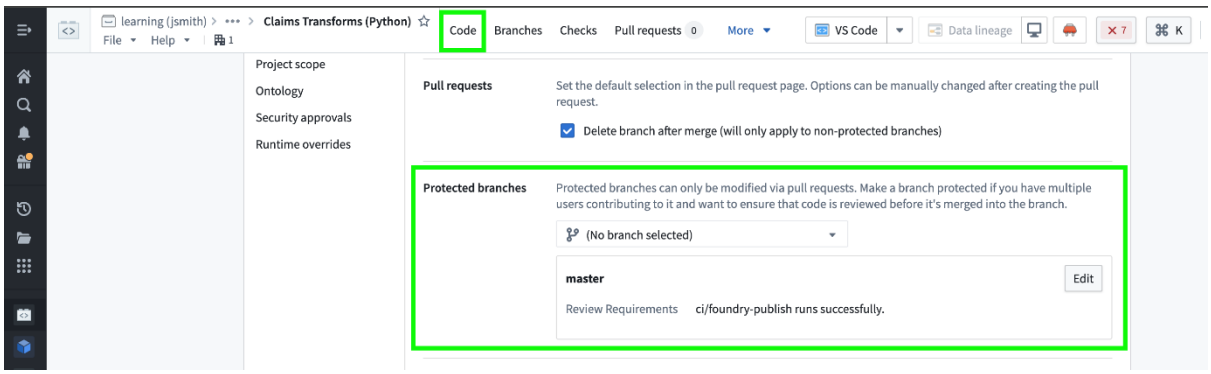
2. Select **Branches**
3. Choose the **master** branch as one of the **Protected branches**
    - Important note: if you don't see the **Protected branches** option, it's likely due to missing permissions. To overcome this, you can either:
        1. try to change your Code Repositoriy permissions to Owner (this should be possible if it is located in your own project),
        2. ask an admin to give you privileged owner access to this repository, or
        3. just skip this part. You will still be able to proceed with the tutorial even if your master branch is not protected.



4. In the popup that shows, leave everything as is and click **Save**.



5. Your settings will update and show as selected:



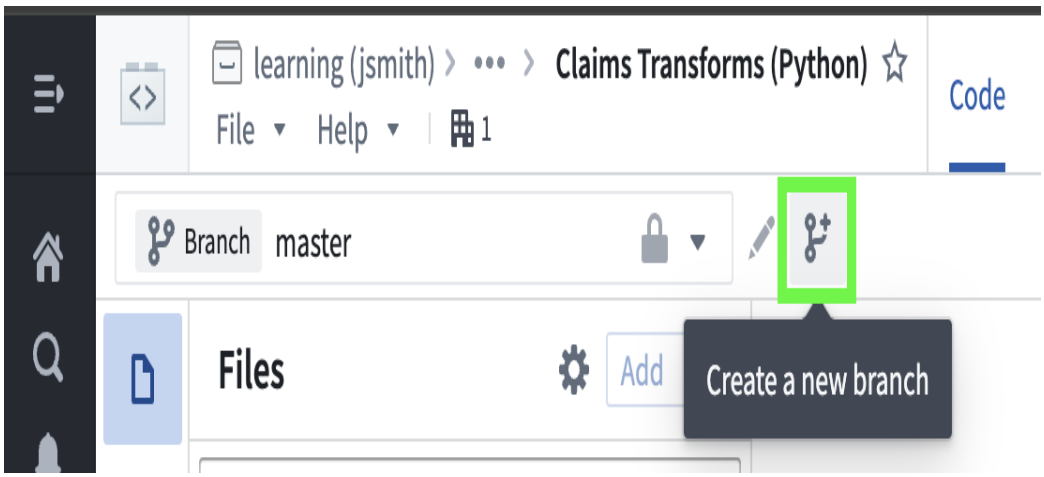
6. Return to your **Code** via the top navigation bar.

Performing a Transform on Your Branch

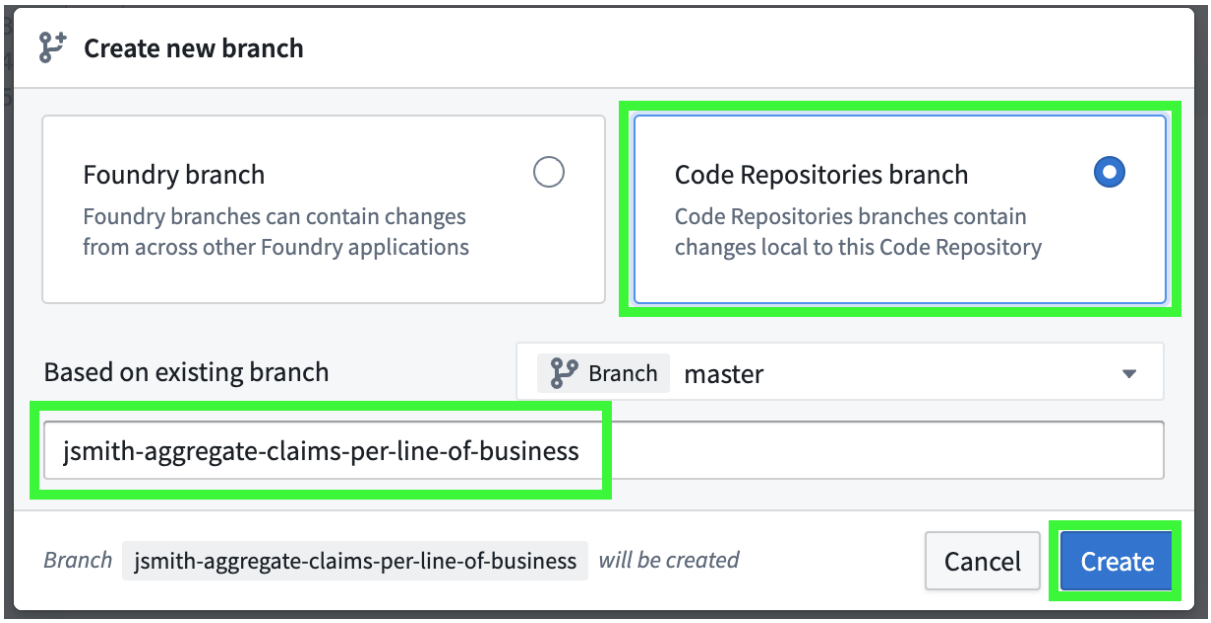
In this lesson, we will create another transform in your existing code file on a new branch.

Step 1: Create new branch

1. Click the little icon with the + to the right of your **Branch master** dropdown, by which you **Create a new branch**.



2. Select **Code Repositories branch**, if presented with the choice
3. **Name** your branch following a descriptive pattern, e.g. <your_username>-aggregate-claims-per-line-of-business
4. Click **Create**

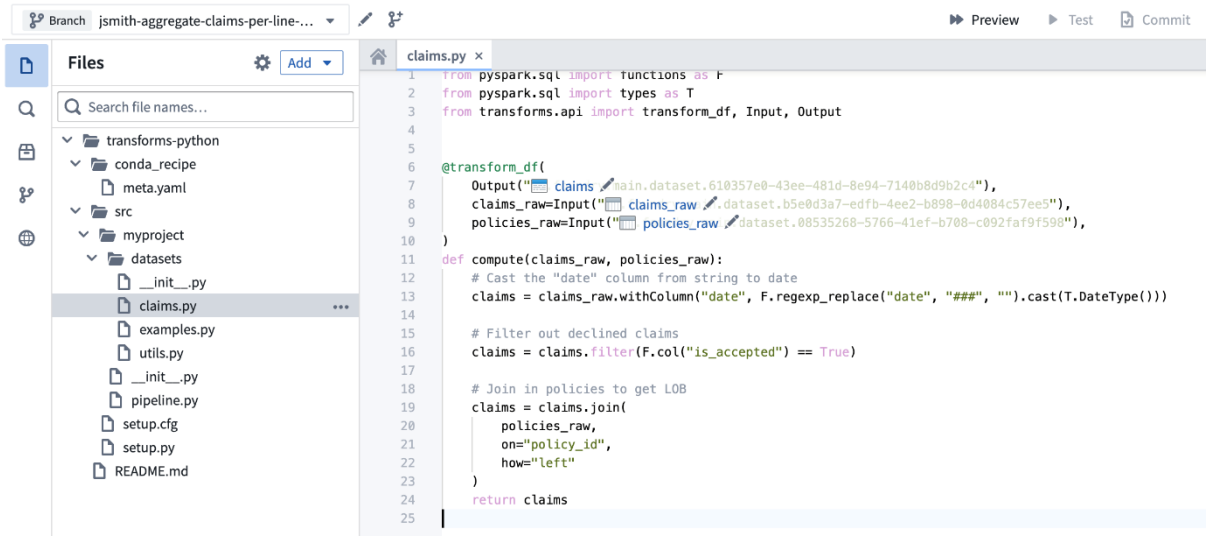


5. You will now land on your code view showing the newly created branch.

Step 2: Create your transform

Remember that we wanted to find the average of all paid out claims by line of business. We will use the same code file as in the previous section, **claims.py**.

Your file should look as follows at this point in time:



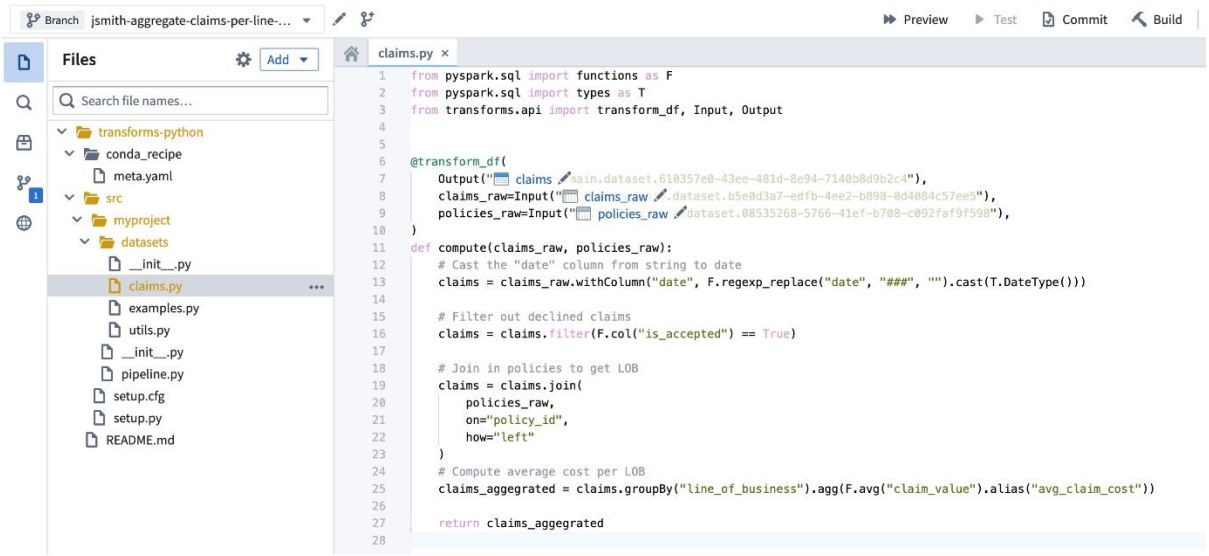
1. Replace line 24 with the following code to aggregate claims by line of business.

Compute average cost per LOB

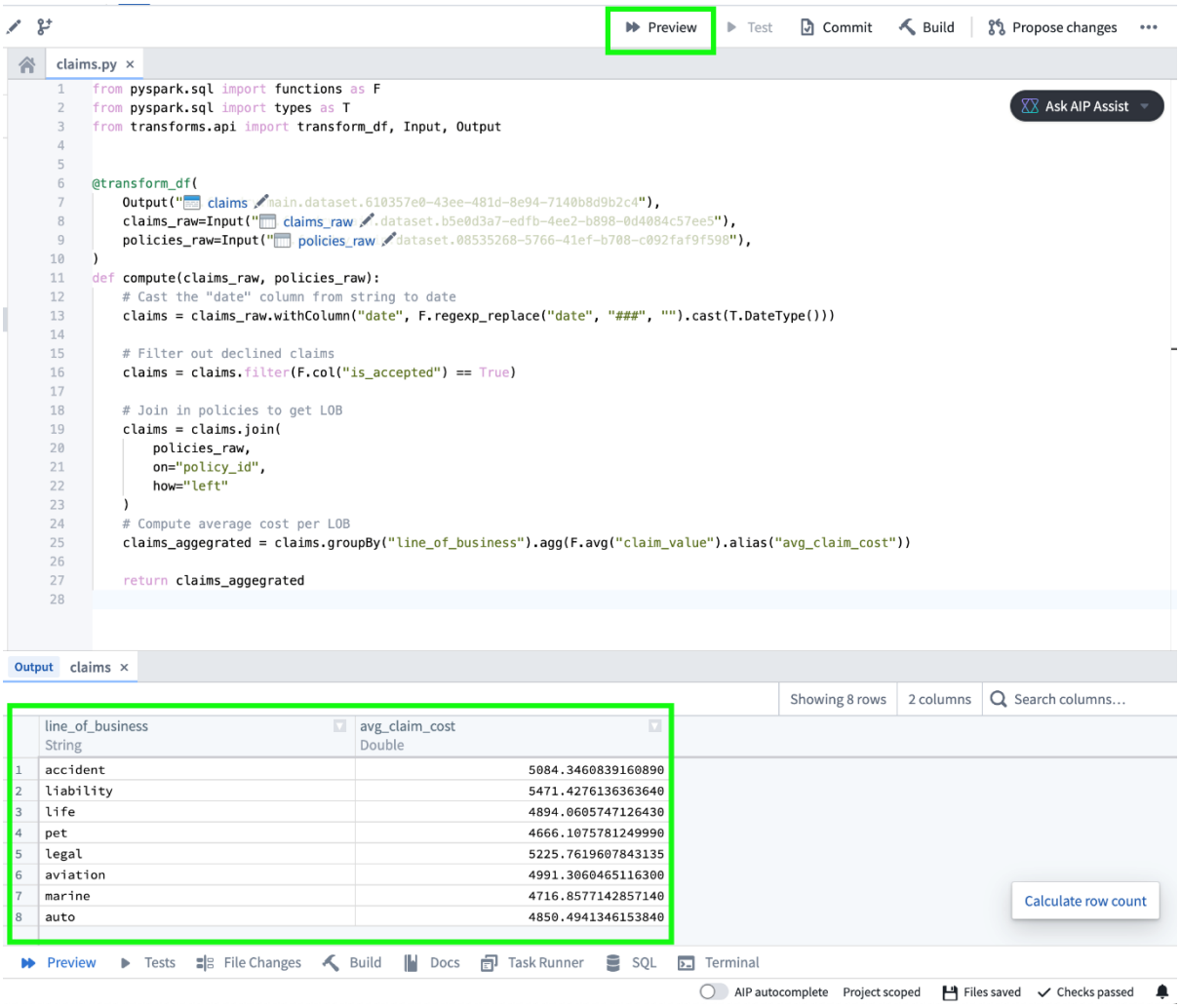
claims_aggegrated = claims.groupBy("line_of_business").agg(F.avg("claim_value").alias("avg_claim_cost"))

return claims_aggegrated

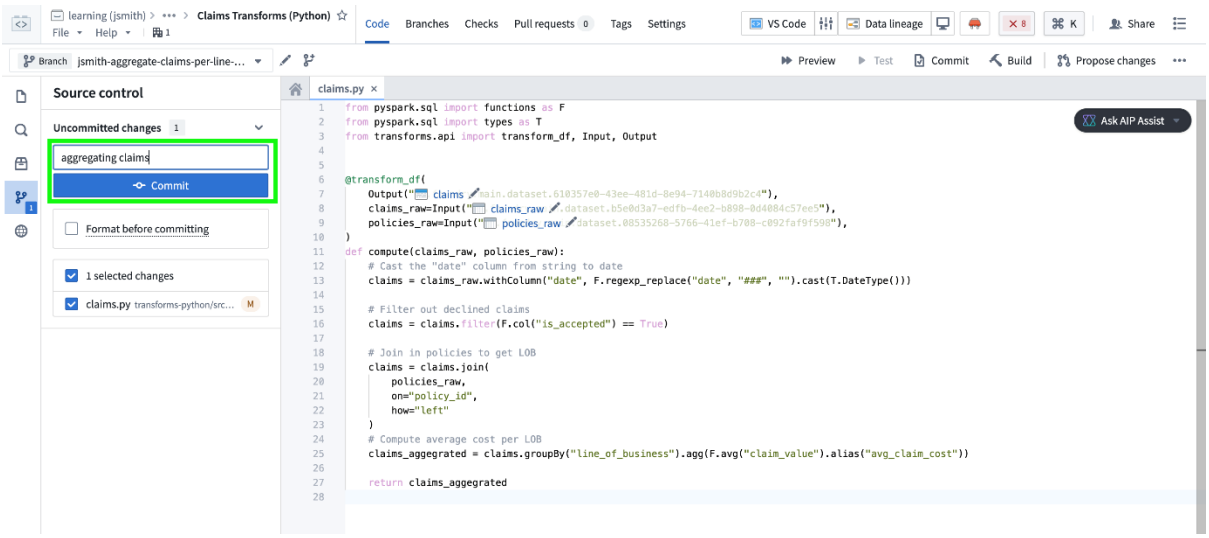
- This will make your file look like this:



2. Hit **Preview** in the top to run a preview of your results.
3. Confirm that you see a new table with two columns: **line_of_business** and **avg_claim_cost**

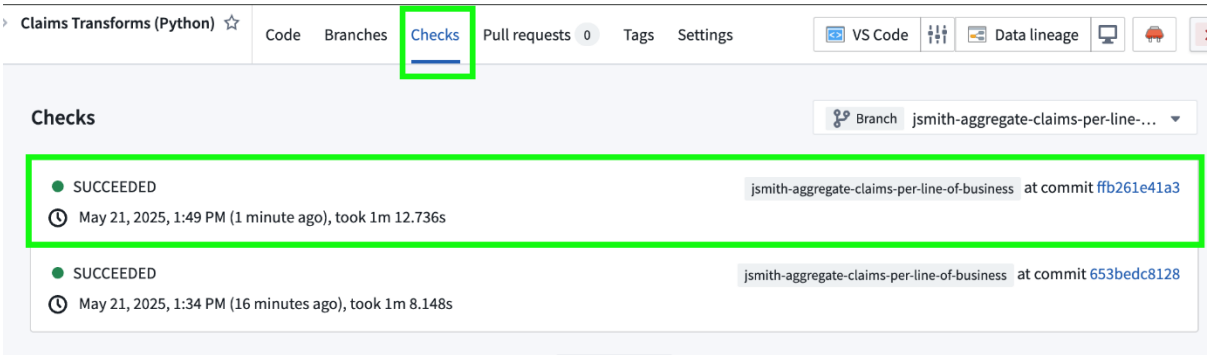


4. Commit your work to save it, on your feature branch
    - Click **Commit** in the top right corner
    - Name your commit, e.g. aggregating claims



5. Open Checks to see that your commit has run successfully

- Click **Checks** in the top navigation bar
- Confirm that the first check in line succeeds

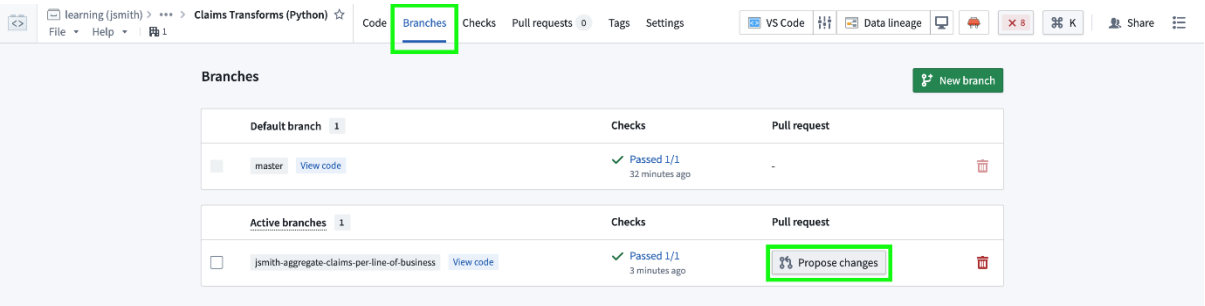


Merge Your Branch into Master

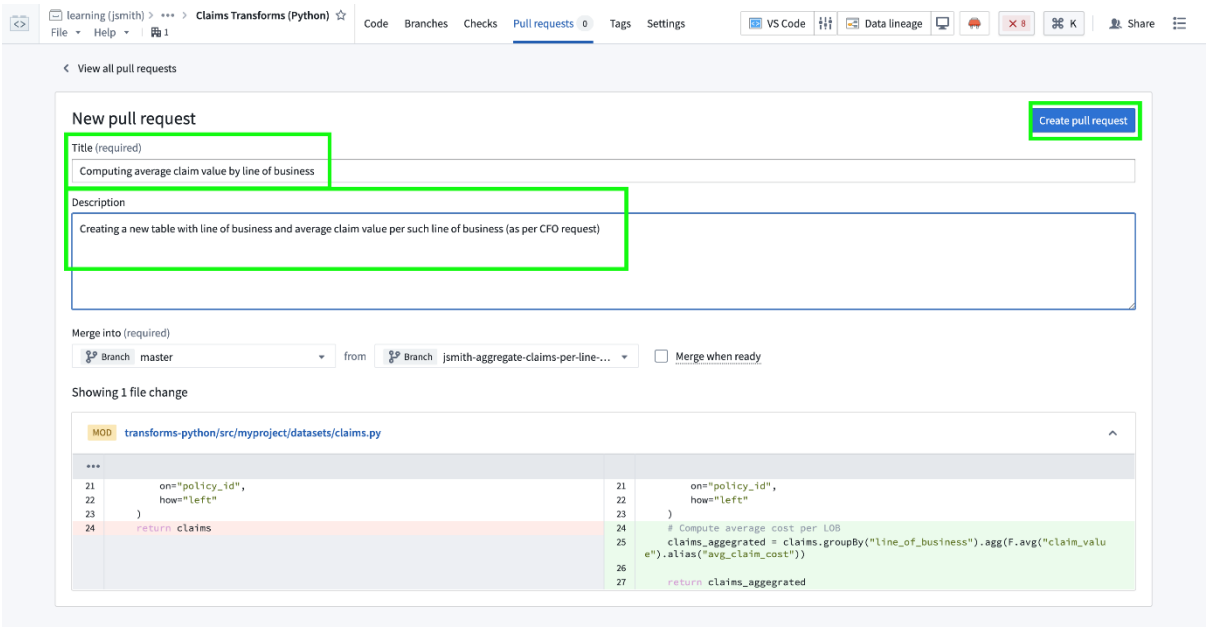
Now that we have verified that our change creates the desired result and checks run successfully, we will merge the code change into the master branch via a Pull Request.

Step 1: Create a new pull request

1. Open **Branches** from the top navigation bar
2. Click on **Propose changes** in the panel belonging to your feature branch

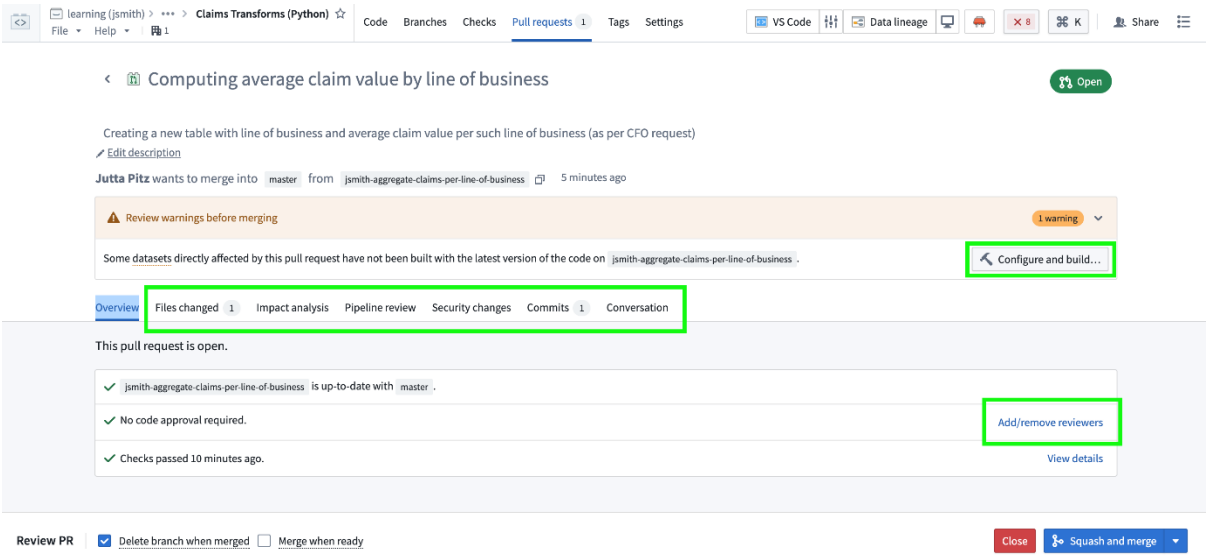


3. A new screen called **New pull request** will open
4. The title is autofilled. Modify if desired, e.g. "Computing average claim value by line of business"
5. Add a meaningful description which helps your colleagues understand the need and the specifics of your change, e.g. “Creating a new table with line of business and average claim value per such lob (as per CFO request).”
6. Leave the rest as is.
7. Click **Create pull request**



Step 2: Merge your changes

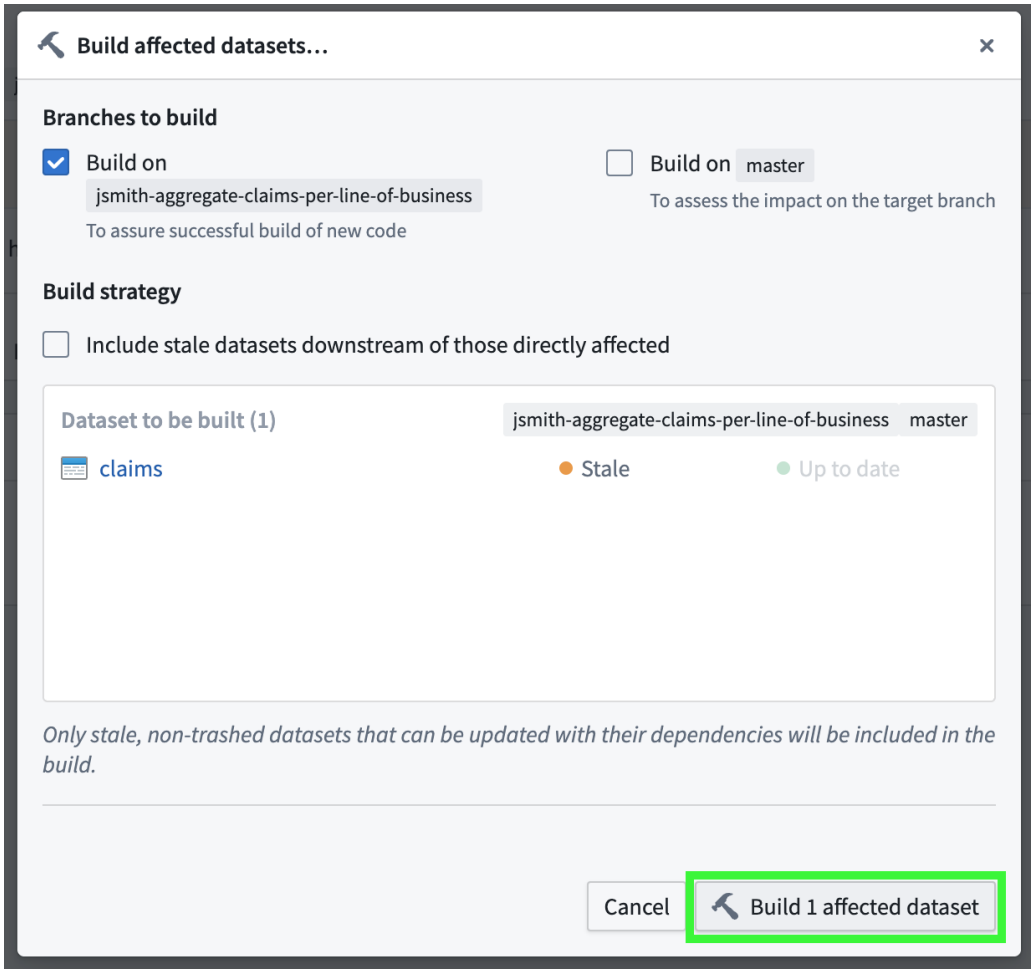
After clicking **Create pull request** in the previous step, you will land on the following screen. This is a summary of your pull request.



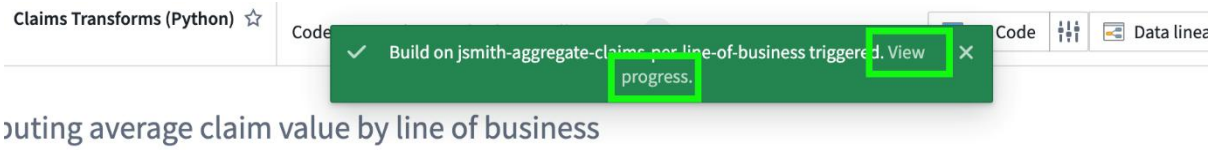
In our example, a warnings panel shows which indicates that some of the datasets have not yet been built with the latest code changes. Normally, it's a good practice to build the dataset on a branch first (see optional steps below) but in our case we can also go ahead and merge without these.

[Optional steps]

- *[optional]* Click on **Configure and build**
- *[optional]* In the new panel that opens, click on **Build 1 affected dataset**



- *[optional]* Open the running build in a new tab by clicking on **View progress** (with ctrl/cmd) in the green notification that shows at the top.

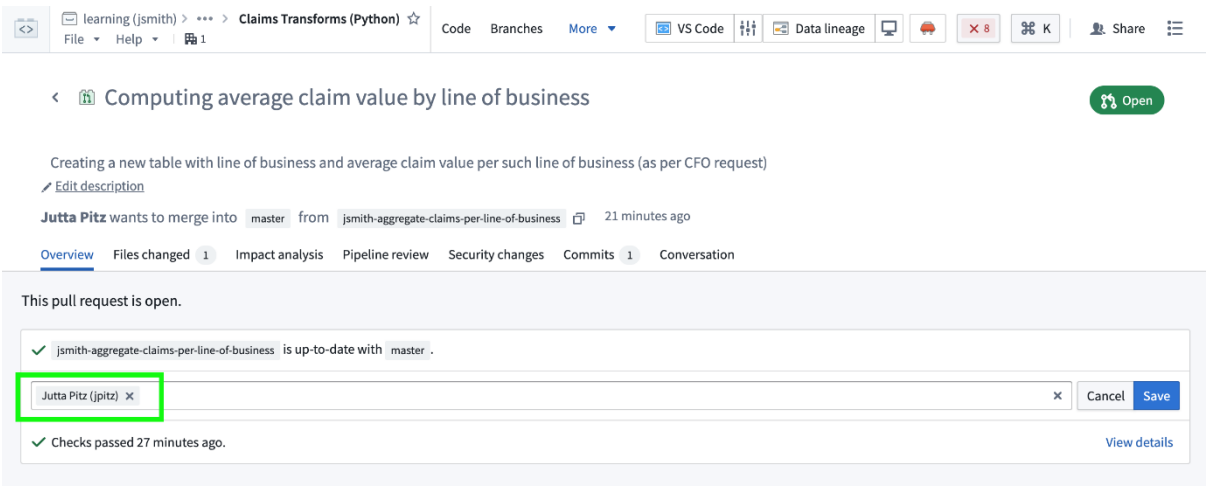


- *[optional]* Refresh your screen to see the screen without the warnings panel.

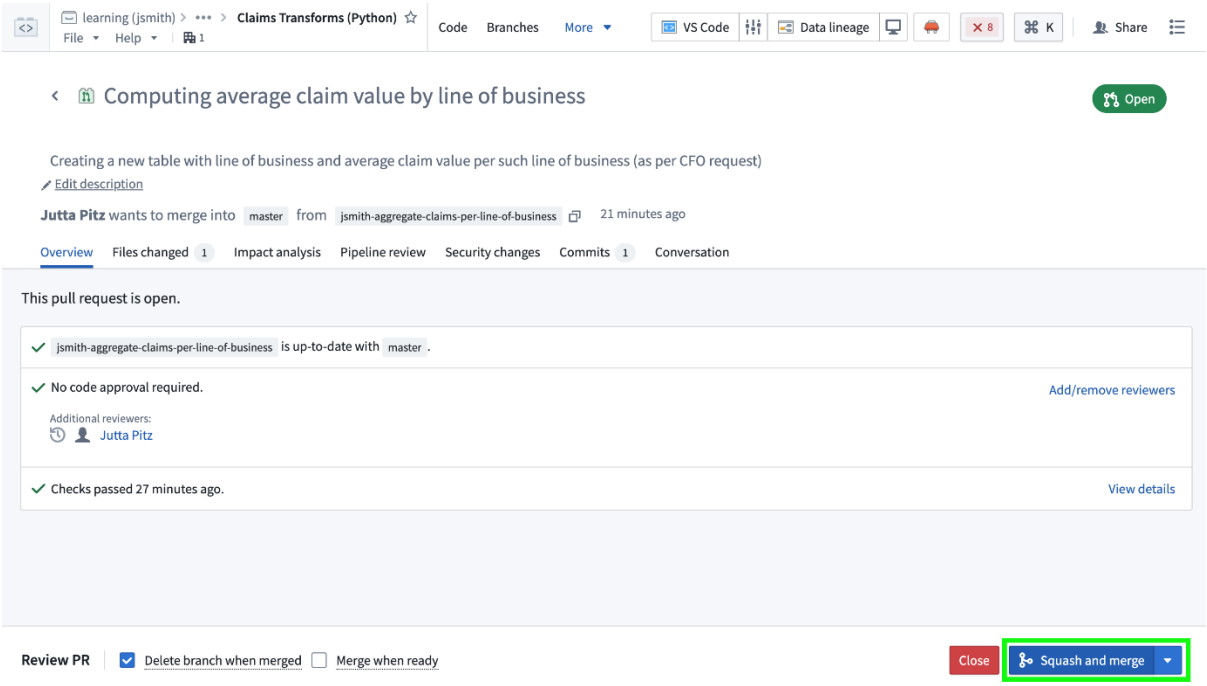
[End of optional steps]

1. Explore the bottom part of this screen further:

- Review the updated files by clicking in the **Files changed 1** tab.
- View a graph of your raw and prepared datasets under **Pipeline review.**
- Review the Commits you made by clicking in the **Commits 1** tab.
- Add reviewers where it currently says **"No code approval required."** Find a teammate's name from the list to add them.

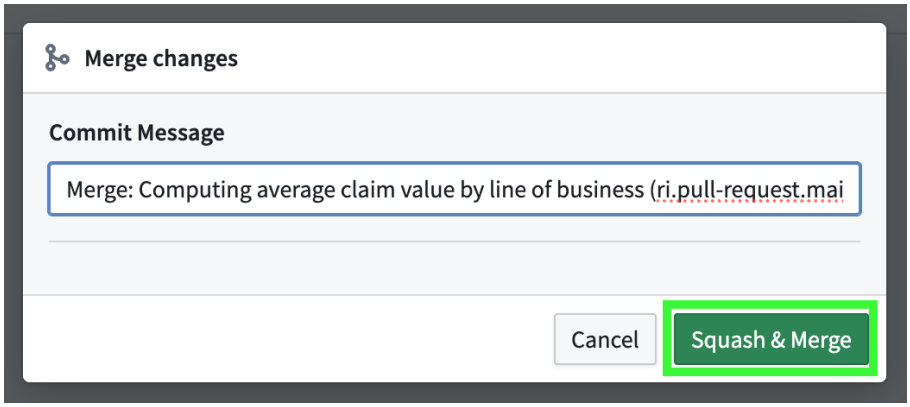


2. When you are done exploring, hit **Squash and merge.**

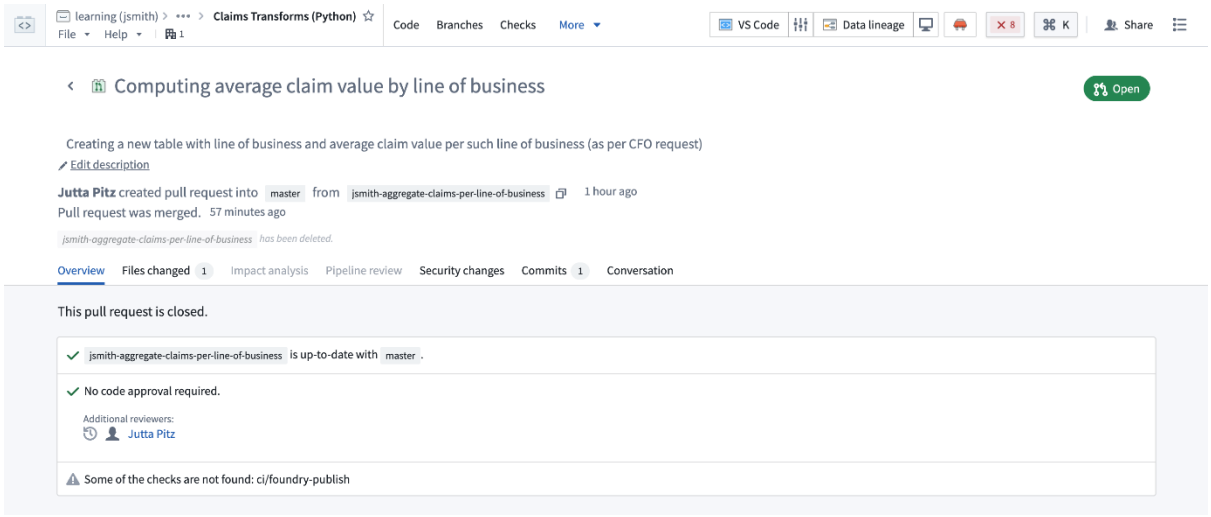


- General note: "Squash and merge" combines all the commits from a pull request into a single commit before merging it into the master, resulting in a cleaner, more concise commit history. On the other hand, "Merge" integrates all individual commits from the pull request as they are, preserving the detailed commit history but potentially making the commit log more cluttered.

3. Leave the Commit Message as is and hit **Squash & Merge**

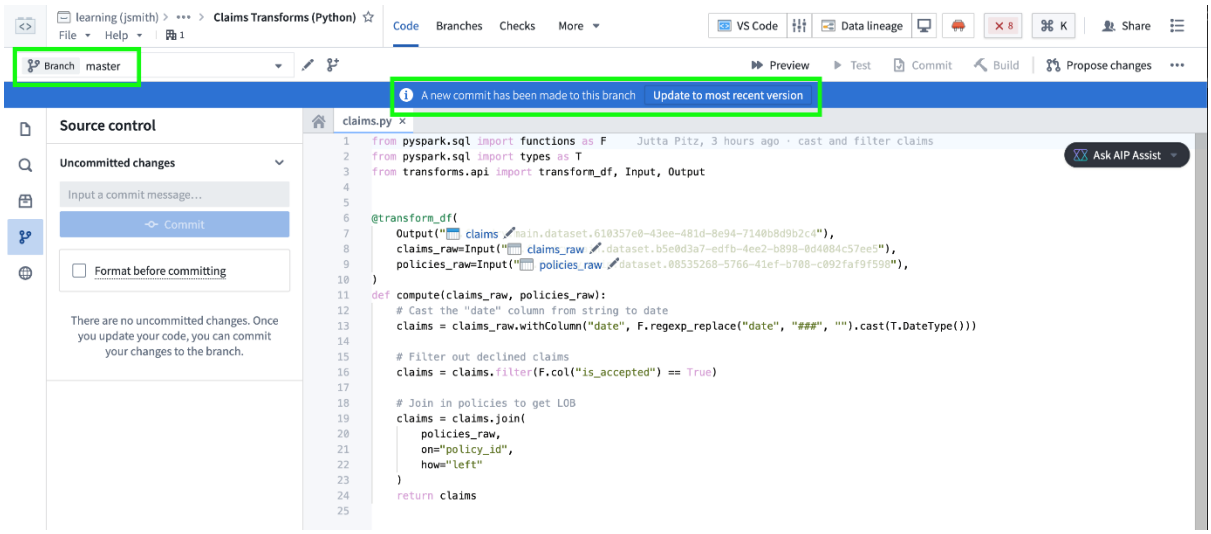


4. Your Pull request will show as merged afterwards and the count next to Pull requests in the top navigation bar will show 0.



5. Click back on **Code.** Code Repositories will notify you that “A new commit has been made to this branch”, which in this case means the master branch.

6. Click “Update to most recent version” to see the changes made.



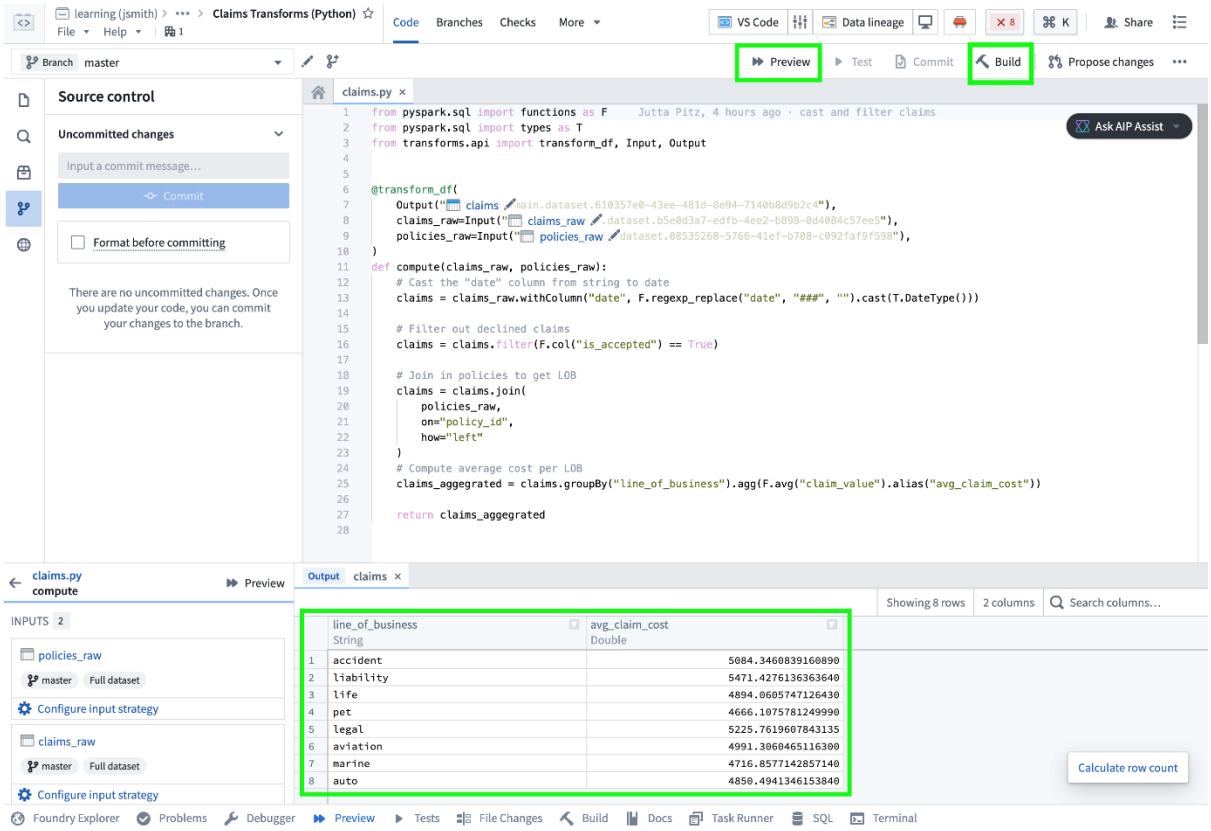
Your code file will now entail the latest changes merged in from your Pull request.

Build Your Dataset

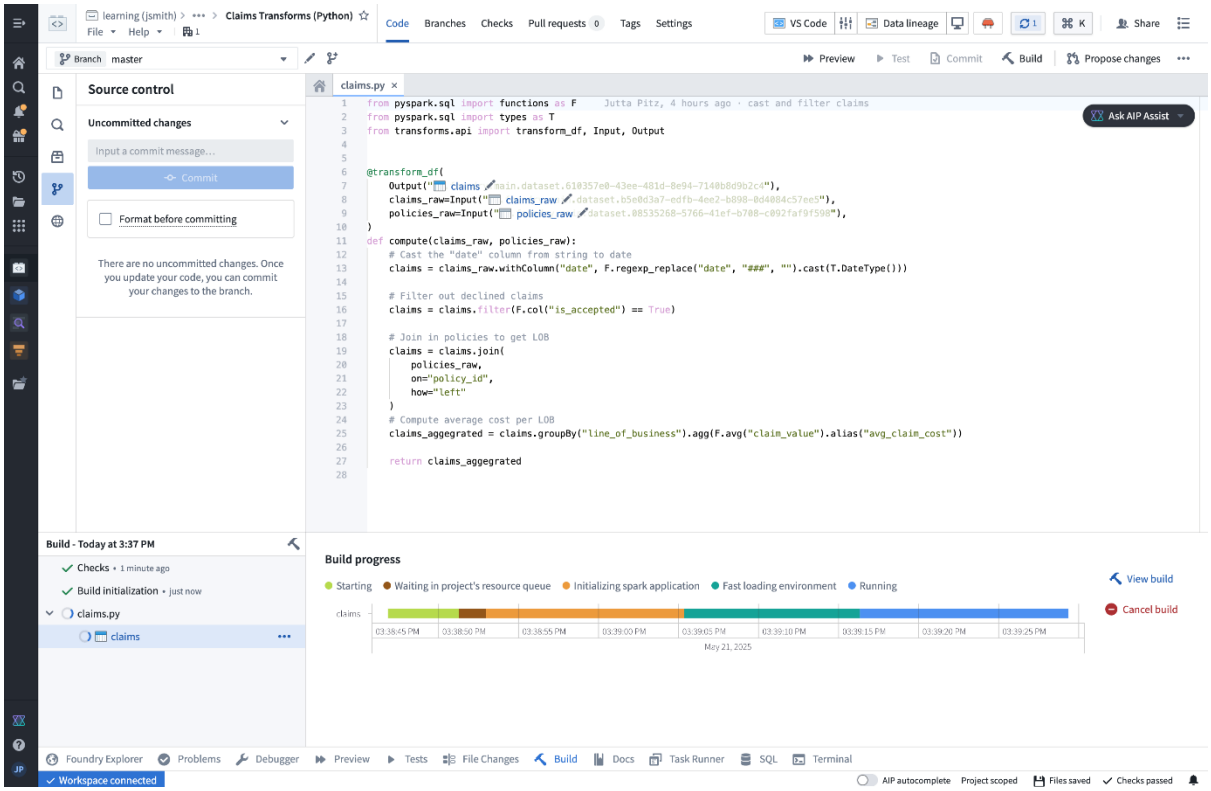
As a final step, you will build your desired dataset on master. You will also investigate the data lineage that we saw at the very beginning of this course.

Step 1: Build the final dataset on master

1. Click **Preview** to ensure your merged changes create the desired 2 column table.



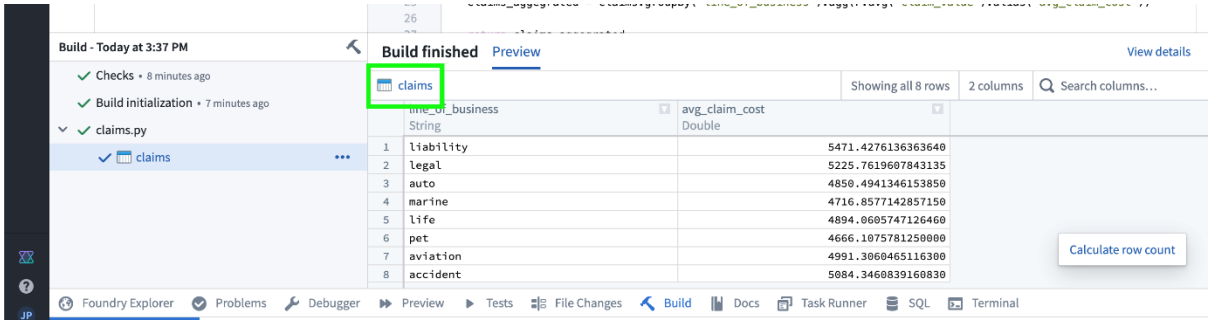
2. Click **Build.**
    - Just like we've seen before, Checks will run automatically.
    - The build then gets initialized.
    - Then the build runs and the build panel shows its progress with this view from job tracker.



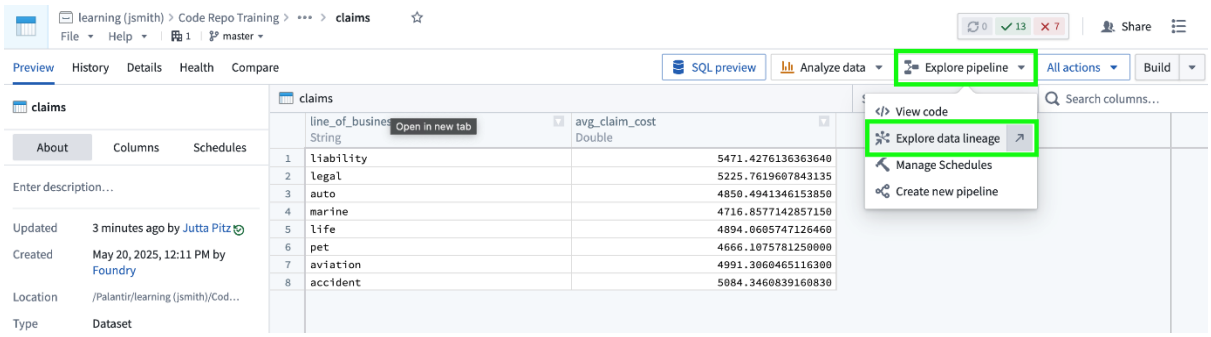
Your dataset will show at the bottom of the page as soon as the build has completed.

Step 2: View the data lineage

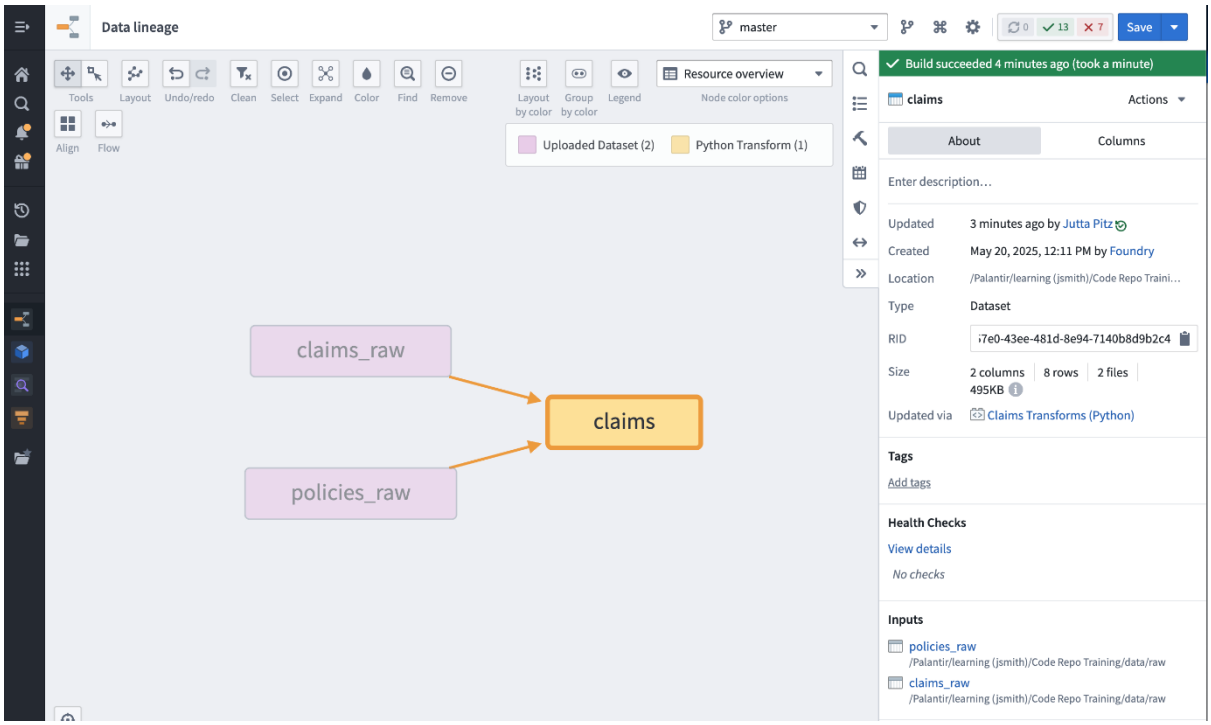
1. Click **claims** with cmd/ctrl to open the newly built dataset in a new tab.



2. Click on **Explore Pipeline** and then **Explore data lineage**.



3. The graph shows you your pipeline created with Code Repository where your two input datasets get joined to your output dataset.



Note if you only see the claims box and not its ancestors, you can add them to the graph by clicking the little arrow to the left on the claims box.

Summary

Congratulations, you've completed the Deep Dive: Transformations in Code Repositories! You have created a PySpark Transform including casting and filtering, used joins and aggregations, worked collaboratively with potential colleagues using Branching and seen how Code Repositories interact with other Foundry tools like Data Lineage.