1. **Describe your solution. What tradeoffs did you make when designing your solution, and why did you make those decisions?**

Punk API requests, Database setup

To import data from the PunkAPI I used the async HTTPX module to send a GET request to the API endpoint and receive the data in JSON format. Due to pagination limits, 5 requests were sent to receive all 325 beer recipes. Then, I created a database schema with two tables: “beers” and “hops”. The “beers” table contains information about each beer, and the “hops” table contains information about the type and amount of hops used in each beer. I then looped over each beer and inserted its data into the “beers” table, and for each beer, I looped over the hops and inserted their data into the “hops” table. I used ORM provided by SQLAlchemy to interact with database.

The average (mean) fermentation temperature for each type of hops

To show the average (mean) fermentation temperature for each type of hops, I created a SQL query that joins the beers and hops tables, groups by the hop name, and calculates the average fermentation temperature.

The average (mean) fermentation temperature for the primary hops

To show the average (mean) fermentation temperature for the primary hops I created two subqueries.
First, that shows the primary hop for each beer along with its maximum amount. Some beers had the same hops within a single recipe, I made sure to calculate the sum first, then showcase only those with the highest value per beer. Some beers have several hops with the same amount as showcased in the example data in README.md.
Second subquery calculates average fermentation temperature for each hop. I was able to join the above queries to showcase results.
For beers with multiple primary hops, average fermentation temperature varies for individual hops.

Tradeoffs

I extracted only relevant data that are required for making necessary queries from JSON response due to its nested structure and nested arrays. I checked for the missing data in the extracted data.
Dropped beer data 3 of 325 (0.92%):

- Beer ID 169 was missing fermentation temp and had no hops.
- Beer ID 72 was missing fermentation temp.
- Beer ID 178 had a fermentation temp 99 C which based on the research is way above the max fermentation temperature (up to 32 C). It would affect average fermentation significantly.

Giving more time, I would try to make the process more robust to handle missing or corrupted data and extract all available data for future analysis. If more values were missing, dropping data would greatly impact the results. One solution would be to use

the average fermentation temperature from all beers to fill missing or corrupted fermentation values in those 3 beers. Some data cleaning logic would be required to mitigate this.

2. **Take a look at the views you created, and make any other queries you think might be useful. From this data (not from your knowledge about brewing beer, if you have any), do you think there is a correlation between hop types chosen for a recipe and fermentation temperature? You don't need to demonstrate a correlation formally (or lack thereof). Just tell us what you see.**

   I created the following additional queries and corresponding API endpoints:
   - Show the top 10 most used hops in the recipes
   - Get all beers that have a fermentation temperature greater than X
   - Get all hops that have an amount greater than or equal to X
   - Get all beers that have a hop with the name X and order them by fermentation temperature

   Correlation
   From the data, I couldn't notice any correlation between hop types and fermentation temperature. Some hop types, such as "Ahtanum" and "Styrian Goldings", have a wide range of fermentation temperatures, while others, such as "Bramling Cross" and "Chinook", have a narrower range.

3. **We use the data from our data warehouse to make important product and business decisions, and some reports are visible to important stakeholders (investors), so it's important that the data is always as accurate and as up-to-date as possible. What kinds of problems could arise with the data import as you have implemented it? How would you mitigate these issues and monitor and/or alert when there are problems?**

   One problem that could arise with the data import is if the PunkAPI changes its schema or returns data in a different format. To mitigate this, we could implement error handling to detect any changes in the API response and modify the import process accordingly. Another problem could be if the data in the PunkAPI is inaccurate or incomplete. To mitigate this, we could implement data validation and cleaning routines to ensure that only valid and complete data is inserted into the database.

4. **Suppose that now we would like to know how the beer recipes are updated over time, so we would like to store a history of the data. How would you approach this? (No need to implement anything, but describe your solution).**

   To store a history of the beer recipe data, we could create a new table "beer_recipe_history" with additional timestamp column and relationship to "beers" and "hops". Whenever a beer recipe is updated, a new row can be inserted into the

"beer_recipe_history" table with the updated information and a timestamp. We could then query this table to retrieve the history of beer recipe changes over time. We could also implement versioning with data migration tool like Alembic.

5. SUMMARY. What have I learnt from this technical task?

- Working with such a big dataset that's nested with arrays was a challenge by itself. Extracting only the relevant data into the tables was a necessary simplification but giving more time, I would make sure to extract all available data and clean it for the following data analysis.
- Making more advanced SQL queries to showcase the expected results was challenging and required lots of trial and error. It allowed me to get a deeper understanding of SQL queries with both SQL language and ORM SQLAlchemy models.
- Testing the application locally by running docker container and directly from uvicorn server taught me that giving different environments, I might encounter some additional bugs with the same application. One solution could be to pin dependencies versions in requirements.txt to make sure that any updates to dependencies do not affect how application is running.
- Migrating database and SQL query logic from sqlite to postgres turned out to be more challenging than expected. Coming across compatibility issues between those two, when running sql queries with sqlalchemy ORM, and dedicating more than 2 hours to fixing bugs on postgres, instead of delivering something not working, I decided to provide my working solution with sqlite. I will allocate additional time to research how to make my databases and queries more robust and compatible with postgres.