

The Knights of Favonius – Genshin Impact

SI 206 FA24 Final Project by Andrew Stetz, Eva Prouty, and Nick Pisarczyk

GitHub Repo: <https://github.com/npisar/SI206-final-project>



Goals & Initial API's

As players and fans of the game Genshin Impact, we wanted to understand better how the franchise that has been running for almost five years is able to produce characters with a limited number of elements and weapons and keep gameplay fun and enjoyable for its fans. We were split, as some of us thought it had to do with the unique characters, and others believed it was the weapons. We looked at data on characters, weapons, artifacts, and banners to get a better idea of the player experience.

We also wanted to create a resource for fans to look at characters and quickly understand the data related to weapons, banners, and artifacts which are all largely variable between players.

We originally planned to work with GSImpact, GenshinDev, and Genshin API. GSImpact would give us Character data, Genshin Dev our weapon data, and Genshin API for Domain and Ascension Materials.

Conclusions and Final APIs

After this project we concluded that it wasn't one thing that contributed to the fun of playing Genshin but instead all of the different things combined. The game keeps the gameplay fresh and exciting by having so many options and allowing players to choose what kind of adventure they'd

like to have. People who prefer using bow characters have a variety of options for bow-using characters, bows in the game, and even artifacts that help power up their shots. There is something for everyone and with plenty of items it's easy to switch up your style.

As players, having the database has been extremely helpful. For example, when finishing our daily tasks and getting artifacts, we can easily use the database to check the name of my item and see if we received the maximum quality. If we didn't, then we let other players have them. This database has been genuinely helpful for managing inventory space and giving back to the Genshin community by sharing items around instead of hoarding them in our inventories.

We worked with two API's and one website;

API/Website	Data Accessed	Information Collected	# of Rows
API 1: GSHImpact (58+39+30 = 127 total rows of data)	Character Data	<ul style="list-style-type: none"> Character's ID Character's Name Characters Rarity Characters Vision Character's Weapon type 	58
	Banner Data	<ul style="list-style-type: none"> Banner ID Characters featured on the banner 	39
	Media Data	<ul style="list-style-type: none"> Character ID The sum of times they appear in different kinds of media 	30
API 2: Genshin.dev (192 total rows of data)	Weapon Data	<ul style="list-style-type: none"> Weapon Names Types of weapon (bow sword) Rarity of Weapon Base attack 	192
Website 1: Genshin Impact Wiki - Artifact Sets (251 total rows of data)	Artifact Data	<ul style="list-style-type: none"> Artifact's ID Artifacts's name Artifacts Max Set Quality 	251

Challenges and Problems

Perhaps our biggest challenge was getting the cache to limit to 25 pieces of data entered every time we ran the code. Despite attending office hours and checking countless StackOverflow posts we struggled to get it working. We eventually figured it out and got it working really seamlessly. We leaned on some of the examples from lecture to get an idea of how to best implement this into our own code for our specific use case with 2 APIs and a website.

Another challenge we faced was having one of our original API's not respond to our API calls. No matter how many times we tried, we couldn't get the Genshin API to respond. This caused us to pivot and look at Artifacts instead, which turned out to be a good thing. We believe it aligns closer to our original goals. Artifacts directly change player experience when equipped on characters and domain and ascension materials are more standardized and less interesting.

Our final challenge was getting around no duplicate string data. Initially, when we first presented we had so much string data duplicated across the database, from weapon types to character vision names. After some careful planning to reduce it, we were able to successfully convert it into IDs which could then be used to represent the data across tables without any duplicate string data.

Calculations

We had four main calculations we wanted to run on the data collected, along with an extra calculation for each of the first two main calculations.

1. Average Weapon Damage per Rarity
 - a. All differences in average damage for weapons of each rarity
2. Average Weapon Damage per Type
 - a. All differences in average damage for weapons of each rarity
3. Number of Artifacts per Quality
4. Number of Media per Character

Average Weapon Damage per Rarity:

Weapons in Genshin Impact are relatively hard to come by the higher in rarity they are so we wanted to see if it was worth it for people to spend time gathering higher quality weapons and if it would have any effect on the damage they did in battle.

For this, we looked at all the weapons in each rarity (1 - 5 Stars) , added all of the base damage stats together, and then divided by the number of weapons in that category. This gave us a category average which we could then compare to the rest of the other rarities. We further calculated the difference between each rarity's average damage and all of the other rarities' average damage to get an idea of how impactful it would be to upgrade or downgrade from one rarity to another.

```
def calculate_average_weapon_damage_per_rarity(cur):
    """
    ARGUMENTS:
    cur:
        SQLite cursor

    RETURNS:
    rarity_and_avg_attack: dict
        The keys are integers representing weapon rarity, and the values are floats representing average damage for weapons of that rarity
    """
    cur.execute(
        """
        SELECT id, rarity, base_attack FROM Weapons
        """
    )
    rows = cur.fetchall()
    weapon_data = [{'id':row[0], 'rarity':row[1], 'base_attack':row[2]} for row in rows]
    # print(f"rows is {rows}")

    rarity_dict = {}
    for rarity in range(1, 6):
        rarity_match_list = [i for i in weapon_data if i['rarity'] == rarity]
        rarity_dict[rarity] = rarity_match_list

    # debug print
    # for k, v in rarity_dict.items():
    #     print(f"rarity dict for rarity {k} is")
    #     print(f"{v}")
    #     print(f"{'-'*30}\n")

    rarity_and_avg_attack = {}
    for rarity in range(1, 6):
        rarity_attack_sum = 0
        rarity_count = len(rarity_dict[rarity])

        for entry in rarity_dict[rarity]:
            entry_attack = entry['base_attack']
            rarity_attack_sum += entry_attack
```

```
rarity_and_avg_attack[rarity] = "{:04.2f}".format(rarity_attack_sum / rarity_count)
```

```
# debug print
# for k,v in rarity_and_avg_attack.items():
#     print(f"avg attack for rarity {k} is")
#     print(f"{v}")
#     print(f"{'.'*30}\n")
# print(f"\n\n\n")
return rarity_and_avg_attack
```

```
def calculate_difference_awdpr(awdpr):
```

```
'''
```

ARGUMENTS:

awdpr: dict

Dictionary of average weapon damage per rarity returned from calculate_average_weapon_damage_per_rarity

RETURNS:

difference_awdpr: list of dicts

List of dicts where the keys are each rarity, and the values are a list of calculations comparing the difference between the average damages for each rarity (ignoring the difference between a rarity and itself)

```
'''
```

```
difference_awdpr = []
```

```
for p_i in range(1, 6):
```

```
    calcs = {p_i: []}
```

```
    for n_i in range(5, 0, -1):
```

```
        calc_name = f"{n_i}-{p_i}"
```

```
        calc = "{:04.2f}".format(float(awdpr[n_i]) - float(awdpr[p_i]))
```

```
        if not calc == '0.00':
```

```
            calcs[p_i].append({calc_name: calc})
```

```
    difference_awdpr.append(calcs)
```

```
# debug print
```

```
# for calcs_d in difference_awdpr:
```

```
#     for k,v in calcs_d.items():
```

```
#         print(f"calcs list for rarity {k} is")
```

```
#         print(f"{v}")
```

```
#         print(f"{'.'*30}\n")
```

```
# print(f"\n\n\n")
```

```
return difference_awdpr
```

Average Weapon Damage per Type:

All characters have the option of using one out of five different options for weapons. A sword, a bow, a claymore, a catalyst, or a polearm. We wanted to see if all the weapons did the same amount of damage or if one did more than another. We expected that the claymore would do the most damage as it's a larger version of the sword and was carried by characters who were typically portrayed as stronger.

We took each of the weapons and created a dictionary where the keys were the int values of weapon types and the values were the average damage. We then added the base damage they did, divided it by the number of weapons in that group and found that Polearm was the strongest weapon on average. We also calculated the difference in average damage for each weapon type versus every other weapon type—similar to how we did it with rarities—to get a better idea of the actual difference between each individual weapon type combination.

```
def calculate_average_weapon_damage_per_type(cur):
    """
    ARGUMENTS:
    cur:
        SQLite cursor

    RETURNS:
    rarity_and_avg_attack: dict
        Dict where the keys are integers representing weapon type, and the values are floats representing average damage
        for weapons of that type
    """
    cur.execute(
        """
        SELECT weapons.id, weapontypes.weapon_type, weapons.base_attack
        FROM Weapons JOIN WeaponTypes
        ON Weapons.weapon_type_id = WeaponTypes.id
        """
    )
    rows = cur.fetchall()
    weapon_data = [{'id':row[0], 'weapon_type':row[1], 'base_attack':row[2]} for row in rows]
    # print(f"rows is {rows}")
    # print(f"weapon_data is {weapon_data}")

    weapon_types = []
    for weapon in weapon_data:
        if weapon['weapon_type'] not in weapon_types:
            weapon_types.append(weapon['weapon_type'])
    if len(weapon_types) == 5:
        break
```

```

else:
    continue
# print(f"weapon types is {weapon_types}")

type_dict = {}
for w_type in weapon_types:
    type_match_list = [i for i in weapon_data if i['weapon_type'] == w_type]
    type_dict[w_type] = type_match_list
# print(f"type dict is {type_dict}")
# debug print
# for k, v in type_dict.items():
#     print(f"type dict for type {k} is")
#     print(f"{v}")
#     print(f"{'-'*30}\n")

type_and_avg_attack = {}
for w_type in weapon_types:
    type_attack_sum = 0
    type_count = len(type_dict[w_type])

    for entry in type_dict[w_type]:
        entry_attack = entry['base_attack']
        type_attack_sum += entry_attack

    type_and_avg_attack[w_type] = "{:04.2f}".format(type_attack_sum / type_count)

# debug print
# for k,v in type_and_avg_attack.items():
#     print(f"avg attack for type {k} is")
#     print(f"{v}")
#     print(f"{'-'*30}\n")
# print(f"\n\n\n")
return (type_and_avg_attack, weapon_types)

```

```
def calculate_difference_awdpt(awdpt, weapon_types):
```

```
'''
```

ARGUMENTS:

awdpt: dict

Dictionary of average weapon damage per type returned from calculate_average_weapon_damage_per_type

weapon_types: list

List of weapon types returned from calculate_average_weapon_damage_per_type

RETURNS:

difference awdpr: list of dicts

Keys are each weapon type, and the values are a list of calculations comparing the difference between the average damages for each type (ignoring the difference between a type and itself)

```
'''
difference_awdpt = []
for main_type in weapon_types:
    calcs = {main_type: []}

    for secondary_type in weapon_types:
        calc_name = f"{secondary_type}-{main_type}"
        calc = "{:04.2f}".format(float(awdpt[secondary_type]) - float(awdpt[main_type]))
        if not calc == '0.00':
            calcs[main_type].append({calc_name: calc})
        elif calc == '0.00' and not main_type == secondary_type:
            calcs[main_type].append({calc_name: calc})
    difference_awdpt.append(calcs)

# debug print
# for calcs_d in difference_awdpt:
#     for k,v in calcs_d.items():
#         print(f"calcs list for type {k} is")
#         print(f"{v}")
#         print(f"{'-'*30}\n")
# print(f"\n\n\n")
return difference_awdpt
```

Number of Artifacts per Quality:

Artifacts are perhaps the most variable thing in the game. Each time a dungeon is completed an artifact of random stats is generated within a range of quality and given to the player. Many players spend long periods doing dungeons over and over to get stronger artifacts that help their characters. But a general rule of thumb is the higher the quality of the set the better it is for the character. We wanted to see how many artifacts had maximum stat qualities to see if it was more common for lower-quality artifacts to appear or if there was an equal chance of high and low artifacts out of the total number of artifacts that can be created.

We created a dictionary that kept a record of the set quality and scoured the data to update the value counts of how many times that rarity appeared. We found that there are many more five-star quality artifacts possible but that they must be generated less often as most of my inventory space is full of lower-quality artifacts. We concluded that more data collected on artifacts would help clarify the relationship between quality and the chance of being generated for the player.


```

def calculate_num_artifacts_per_quality(cur):
    """
    ARGUMENTS:
    cur:
        SQLite cursor

    RETURNS:
    num_artifacts_per_quality: dict
        Dict where the keys are integers representing artifact max quality, and the values are ints representing the count for
        artifacts of that max rarity
    """
    cur.execute(
        """
        SELECT id, max_set_quality FROM Artifacts
        """
    )
    rows = cur.fetchall()
    artifact_data = [{ 'id':row[0], 'max_set_quality':row[1]} for row in rows]

    quality_dict = {}
    for quality in range(1, 6):
        quality_match_list = [i for i in artifact_data if i['max_set_quality'] == quality]
        quality_dict[quality] = quality_match_list

    # debug print
    # for k, v in quality_dict.items():
    #     print(f"rarity dict for rarity {k} is")
    #     print(f"{v}")
    #     print(f"{'-'*30}\n")

    quality_and_count = {}
    for quality in range(1, 6):
        quality_count = len(quality_dict[quality])

        quality_and_count[quality] = quality_count

    # debug print
    # for k,v in quality_and_count.items():
    #     print(f"count for quality {k} is")
    #     print(f"{v}")
    #     print(f"{'-'*30}\n")
    # print(f"\n\n")
    return quality_and_count

```

Number of Media per Character:

The “media” endpoint of the GSHImpact API was interesting to us, as it offers a direct insight to how popular (or unpopular) a given character is. While we believe that the data that GSHImpact had for us is at least a little outdated, it still offers a meaningful snapshot into what characters are being talked about. We took all of the pieces of data the API offered (promotion, holiday, birthday, videos, cameos, artwork) and summed them to get the total number of media appearances a character has.

```
def calculate_num_media_per_character(cur):
    cur.execute(
        """
        SELECT
            Characters.name,
            Media.promotion + Media.holiday + Media.birthday + Media.videos + Media.cameos + Media.artwork AS
sum_of_media_values
        FROM Media JOIN Characters
        ON Media.character_id = Characters.id
        """
    )
    rows = cur.fetchall()
    media_data = [{row[0]:row[1]} for row in rows]

    cur.execute(
        """
        SELECT name FROM Characters
        """
    )
    all_characters = cur.fetchall()
    all_character_names = [row[0] for row in all_characters]

    # Creating a set of character names that already exist in media_data
    existing_characters = {list(char_dict.keys())[0] for char_dict in media_data}

    # Adding characters that are not present in media_data
    for name in all_character_names:
        if name not in existing_characters:
            media_data.append({name: 0})

    # Sort the media_data list by character names
    media_data = sorted(media_data, key=lambda d: list(d.keys())[0])

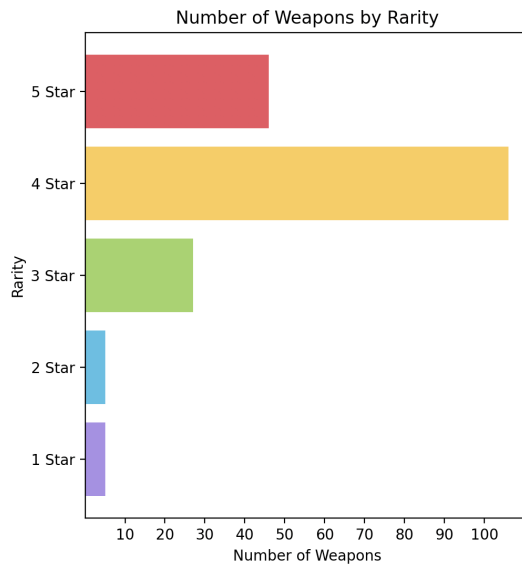
    return media_data
```

Formatted Calculations File

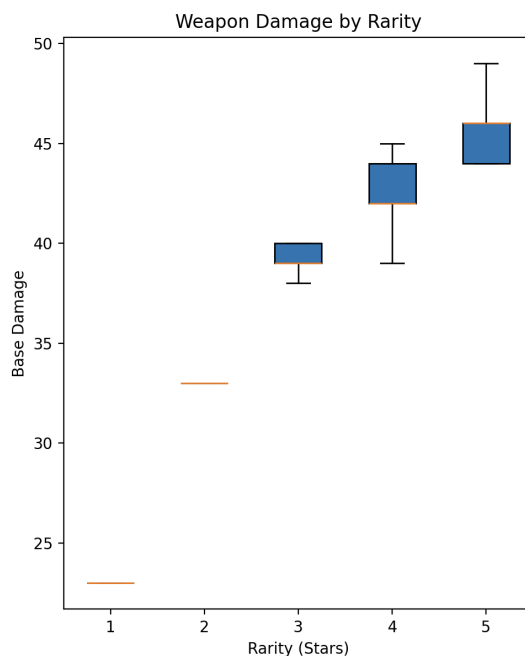
Running the calculations file will result in a new file *calculations.txt* being created/ That file looks like this:

```
1  FUNCTION / CALCULATION 1:
2  Average Weapon Damage Per Rarity:
3
4  This section shows the average base attack for weapons of each rarity (1 to 5 stars).
5  Rarity 1 Star: 23.00 Base Attack
6  Rarity 2 Star: 33.00 Base Attack
7  Rarity 3 Star: 39.15 Base Attack
8  Rarity 4 Star: 42.41 Base Attack
9  Rarity 5 Star: 45.93 Base Attack
10 -----
11
12 -----
13
14 FUNCTION / CALCULATION 2:
15 Difference in Average Weapon Damage Per Rarity:
16
17 These calculations show the difference in average base attack between weapons of different rarities.
18 Rarity 1 Star Differences:
19   Rarity 5-1 Star Difference: 22.91 Base Attack
20   Rarity 4-1 Star Difference: 19.41 Base Attack
21   Rarity 3-1 Star Difference: 16.15 Base Attack
22   Rarity 2-1 Star Difference: 10.00 Base Attack
23   Rarity 2 Star Differences:
24     Rarity 5-2 Star Difference: 12.91 Base Attack
25     Rarity 4-2 Star Difference: 9.41 Base Attack
26     Rarity 3-2 Star Difference: 6.15 Base Attack
27     Rarity 1-2 Star Difference: -10.00 Base Attack
28   Rarity 3 Star Differences:
29     Rarity 5-3 Star Difference: 6.76 Base Attack
30     Rarity 4-3 Star Difference: 3.26 Base Attack
31     Rarity 2-3 Star Difference: -6.15 Base Attack
32     Rarity 1-3 Star Difference: -16.15 Base Attack
33   Rarity 4 Star Differences:
34     Rarity 5-4 Star Difference: 3.50 Base Attack
35     Rarity 3-4 Star Difference: -3.26 Base Attack
36     Rarity 2-4 Star Difference: -9.41 Base Attack
37     Rarity 1-4 Star Difference: -19.41 Base Attack
38   Rarity 5 Star Differences:
39     Rarity 4-5 Star Difference: -3.50 Base Attack
40     Rarity 3-5 Star Difference: -6.76 Base Attack
41     Rarity 2-5 Star Difference: -12.91 Base Attack
42     Rarity 1-5 Star Difference: -22.91 Base Attack
43 -----
44
45 -----
46
47 FUNCTION / CALCULATION 3:
48 Average Weapon Damage Per Type:
49
50 This section shows the average base attack for each weapon type.
51 Weapon Type Catalyst: 42.21 Base Attack
52 Weapon Type Claymore: 41.86 Base Attack
53 Weapon Type Bow: 41.71 Base Attack
54 Weapon Type Sword: 41.85 Base Attack
55 Weapon Type Polearm: 42.61 Base Attack
56 -----
57
58 -----
59 -----
60 FUNCTION / CALCULATION 4:
61 Difference in Average Weapon Damage Per Type:
62
63 These calculations show the difference in average base attack between different weapon types.
64 Weapon Type Catalyst Differences:
65   Weapon Type Claymore-Catalyst: -0.35 Base Attack
66   Weapon Type Bow-Catalyst: -0.50 Base Attack
67   Weapon Type Sword-Catalyst: -0.36 Base Attack
68   Weapon Type Polearm-Catalyst: 0.40 Base Attack
69 Weapon Type Claymore Differences:
70   Weapon Type Catalyst-Claymore: 0.35 Base Attack
71   Weapon Type Bow-Claymore: -0.15 Base Attack
72   Weapon Type Sword-Claymore: -0.01 Base Attack
73   Weapon Type Polearm-Claymore: 0.75 Base Attack
74 Weapon Type Bow Differences:
75   Weapon Type Catalyst-Bow: 0.50 Base Attack
76   Weapon Type Claymore-Bow: 0.15 Base Attack
77   Weapon Type Sword-Bow: 0.14 Base Attack
78   Weapon Type Polearm-Bow: 0.90 Base Attack
79 Weapon Type Sword Differences:
80   Weapon Type Catalyst-Sword: 0.36 Base Attack
81   Weapon Type Claymore-Sword: 0.01 Base Attack
82   Weapon Type Bow-Sword: -0.14 Base Attack
83   Weapon Type Polearm-Sword: 0.76 Base Attack
84 Weapon Type Polearm Differences:
85   Weapon Type Catalyst-Polearm: -0.40 Base Attack
86   Weapon Type Claymore-Polearm: -0.75 Base Attack
87   Weapon Type Bow-Polearm: -0.90 Base Attack
88   Weapon Type Sword-Polearm: -0.76 Base Attack
89 -----
90
91 -----
92
93 FUNCTION / CALCULATION 5:
94 Number of Artifacts Per Max Set Quality:
95
96 This section shows the count of artifacts for each max set quality value (1 to 5 stars).
97 Max Set Quality 1 Star: 2 total artifacts
98 Max Set Quality 2 Star: 0 total artifacts
99 Max Set Quality 3 Star: 15 total artifacts
100 Max Set Quality 4 Star: 54 total artifacts
101 Max Set Quality 5 Star: 180 total artifacts
102 -----
103
104 -----
105 -----
106 FUNCTION / CALCULATION 6:
107 Number of Media Appearances Per Character:
108
109 This section shows the total media appearances (promotion, holiday, birthday, videos, cameos, artwork) for each character.
110 Character Albedo: 19 total media appearances
111 Character Aloy: 0 total media appearances
112 Character Amber: 18 total media appearances
113 Character Ayaka: 0 total media appearances
114 Character Ayato: 0 total media appearances
115 Character Barbara: 15 total media appearances
116 Character Beidou: 17 total media appearances
117 Character Bennett: 0 total media appearances
118 Character Chongyun: 16 total media appearances
119 Character Diluc: 15 total media appearances
120 Character Diona: 13 total media appearances
121 Character Eula: 0 total media appearances
122 Character Fischl: 10 total media appearances
123 Character Ganyu: 27 total media appearances
124 Character Gorou: 0 total media appearances
125 Character Hu Tao: 16 total media appearances
126 Character Itto: 0 total media appearances
127 Character Jean: 21 total media appearances
128 Character Kaeya: 14 total media appearances
129 Character Kazuha: 0 total media appearances
130 Character Keqing: 24 total media appearances
131 Character Klee: 24 total media appearances
132 Character Kokomi: 0 total media appearances
133 Character Kuki: 0 total media appearances
134 Character Lisa: 12 total media appearances
135 Character Mona: 0 total media appearances
136 Character Ningguang: 19 total media appearances
137 Character Noelle: 11 total media appearances
138 Character Qiqi: 17 total media appearances
139 Character Raiden Shogun: 0 total media appearances
140 Character Razor: 7 total media appearances
141 Character Rosaria: 0 total media appearances
142 Character Sara: 0 total media appearances
143 Character Sayu: 0 total media appearances
144 Character Shenhe: 0 total media appearances
145 Character Sucrose: 11 total media appearances
146 Character Tartaglia: 15 total media appearances
147 Character Thomas: 0 total media appearances
148 Character Traveller (female): 13 total media appearances
149 Character Traveller (male): 30 total media appearances
150 Character Venti: 22 total media appearances
151 Character Xiangling: 21 total media appearances
152 Character Xiao: 0 total media appearances
153 Character Xinglu: 17 total media appearances
154 Character Xinyan: 15 total media appearances
155 Character Yae: 0 total media appearances
156 Character Yanfei: 0 total media appearances
157 Character Yanli: 0 total media appearances
158 Character Yoimiya: 0 total media appearances
159 Character Yun Jin: 0 total media appearances
160 Character Zhongli: 30 total media appearances
161 -----
```

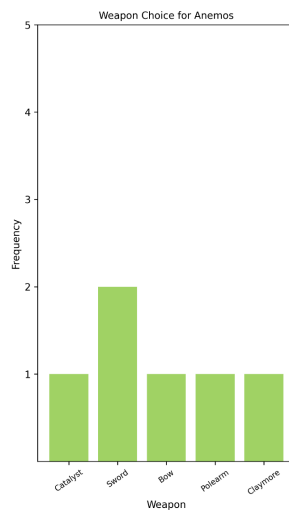
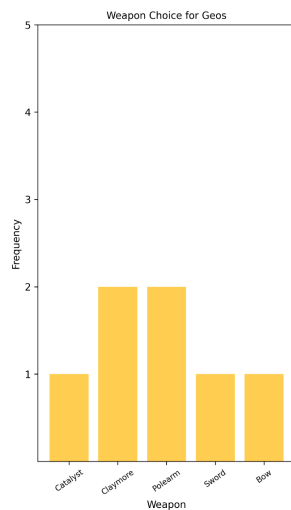
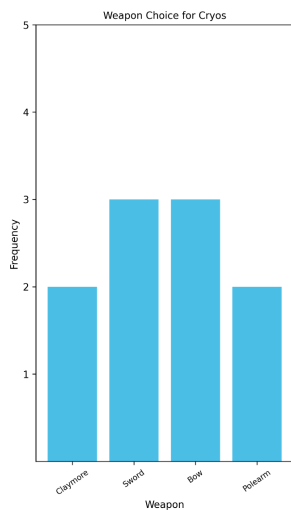
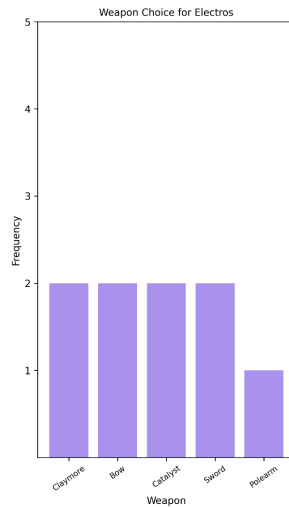
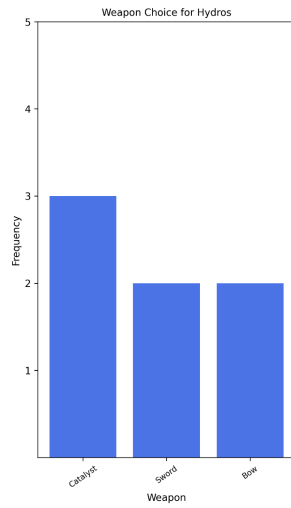
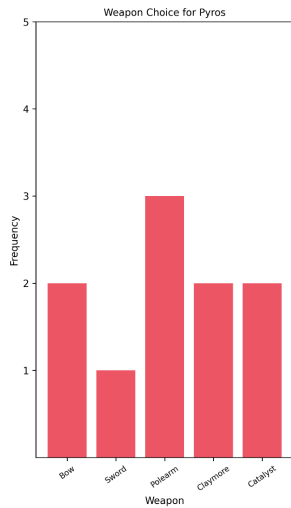
Visualizations



Our first visualization is a horizontal bar chart that has the number of weapons listed against their rarity. This shows that four-star weapons make up a majority of the weapons available to players.

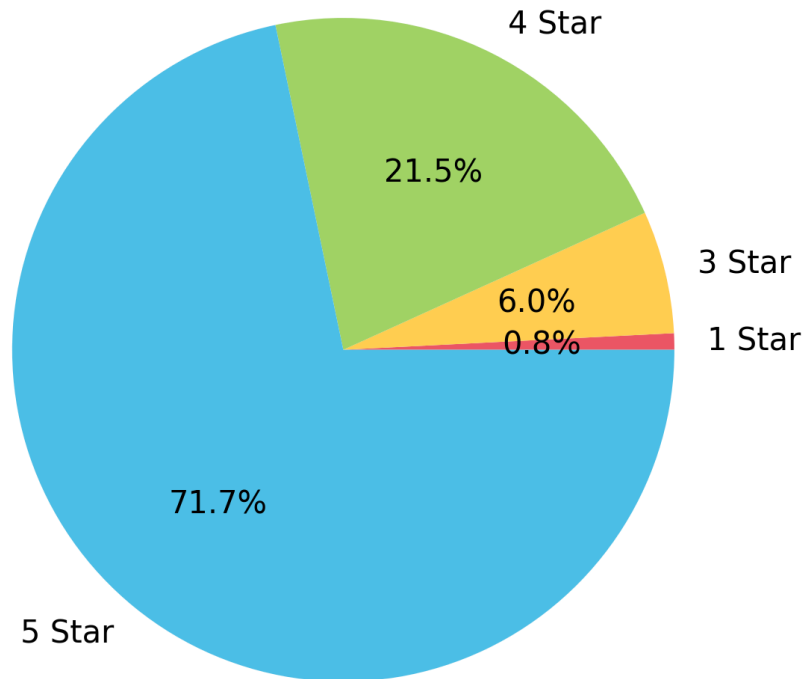


We also collected data on damage based on rarity to find that four-star weapons had the largest range of damage. Many players are only able to collect four stars weapons. However, the top whiskers of the four star weapon range show that the higher-rated four star weapons do comparable amounts of damage to lower-tier five star weapons. For players who can't afford to find the five-star weapons, four-star weapons can do just as much (if not more) than some five-star weapons. Something else interesting we found was that all one-star weapons have the same base damage, which is also true for two-star weapons.



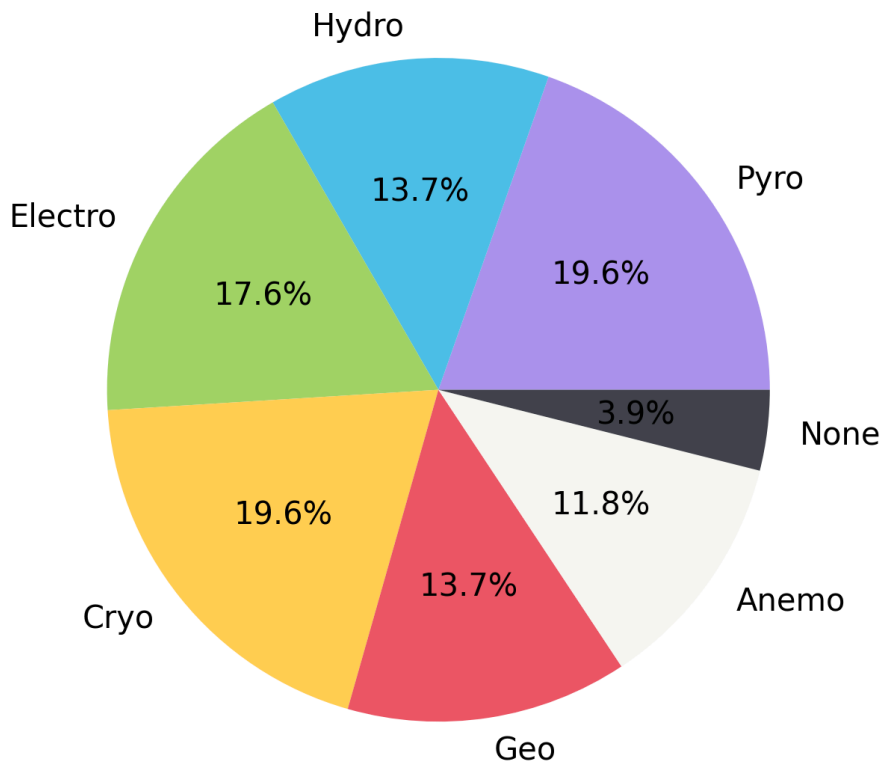
The next few visualizations are bar charts that show the number of characters that use each kind of weapon for each different character “vision” (basically their elemental type). So for example, 2 Pyro characters use the Claymore. We can compare the number of characters per weapon in each vision as well as look at the different kinds of weapons each vision has. These graphs are fairly even, with a few characters of each vision using a specific weapon. One thing to note is that not all weapon types are represented in every vision: there is no Hydro claymore or polearm, and there is no Cryo catalyst.

Maximum Artifact Set Quality Distribution



We also created a pie chart to visualize the maximum set quality distribution for artifacts. We can see that more than 75% of all artifacts have the possibility to be a five-star rarity.



Character Vision Distribution



Another pie chart we created was the character vision distribution. We were curious to see if there was one vision that had more characters, especially since there are usually more characters released that have the same vision as the region the character can visit. But after processing and visualizing the data, it ended up being pretty even among all of the visions.

Running the Code

Our code needs to be run in a very specific order, since different tables rely on being connected to data in other tables. Please read and follow these instructions carefully to ensure the database is created correctly. First, the structure of the repository:

Repository Structure Guide		
Name of Folder	Description	Guide
-old	old files not in use – ignore	<div><ul style="list-style-type: none">> -old> calculations> data-and-tables> graphs<div> README.md</div><div> TODO.txt</div></div>
calculations	contains calculations.py, and is where the calculations text file will be created	
data-and-tables	contains all 5 fill_table.py files, and is where the database file will be created	
graphs	contains graphs.py, and is where the visualization images will be created and stored	

Now that the structure of the repository is clear, here are the steps for running the code:

Run `fill_weapon_table.py` five (5) times

This file can be found in the *data-and-tables* folder.

There's a lot of data that needs to be accessed from the API on every run of this file. This is by far the data that takes the longest to fill in our entire database—it usually takes over a minute for each run.

Upon running, the terminal will print “Weapon data being gathered from the Genshin.dev API! Please wait...”, and eventually “Weapon API call done! Adding items to the database...” Make sure you wait until you see the message “25 / 94 total rows of weapon data added to the database. Run the file again!” to hit run on the script again. (It's worth noting that since we needed to add the 5 different weapon types to a separate `WeaponTypes` table to avoid any duplicate string data, the file only adds 20 rows of weapon data in the `Weapons` table on the first run.)

You'll need to run this file five (5) times total. The first four will add 25 rows of the data to the database at a time each. After 100 rows of data have been added, the remaining 94 rows will be added on the fifth run of `fill_weapon_table.py`. This means the fifth run will take a little bit longer after the API call finishes. After the fifth run of the file finishes, you'll see the message “All weapon data added to database. Please move on to `fill_character_table.py`!”

Run fill_character_table.py three (3) times

This file can be found in the *data-and-tables* folder.

Each run will take far less time than fill_weapon_table.py.

Upon running, the terminal will print "Character data being gathered from the GSHImpact API! Please wait...", and eventually "Character API call done! Adding items to the database..." Make sure you wait until you see the message "25 / 58 total rows of character data added to the database. Run the file again!" to hit run on the script again. (It's again worth noting that since we needed to add the 7 different character visions to a separate CharacterVisions table to avoid any duplicate string data, the file only adds 18 rows of character data in the Characters table on the first run.)

You'll need to run this file three (3) times total. The first two will add 25 rows of the data to the database at a time each. The third will add the remaining 8 rows. After the fifth run of the file finishes, you'll see the message "All character data added to the database. Please move on to fill_banner_table.py!"

Run fill_banner_table.py two (2) times

This file can be found in the *data-and-tables* folder.

Upon running, the terminal will print "Banner data being gathered from the GSHImpact API! Please wait...", and eventually "Banner API call done! Adding items to the database..." Make sure you wait until you see the message "25 / 39 total rows of banner data added to the database. Run the file again!" to hit run on the script again.

You'll need to run this file three (2) times total. The first will add 25 rows of the data to the database. The second will add the remaining 14 items. After the second run of the file finishes, you'll see the message "All banner data added to database. Please move on to fill_artifact_table.py!"

Run fill_artifact_table.py five (5) times

This file can be found in the *data-and-tables* folder.

Upon running, the terminal will print "Banner data being gathered from the GSHImpact API! Please wait...", and eventually "Artifact web scraping done! Adding items to the database..." Make

sure you wait until you see the message “25 / 251 total rows of artifact data added to the database. Run the file again!” to hit run on the script again.

You’ll need to run this file five (5) times total. The first four will add 25 rows of the data to the database at a time each. After 100 rows of data have been added, the remaining 151 rows will be added on the fifth run of `fill_artifact_table.py`. This means the fifth run will take much longer after the API call finishes. After the fifth run of the file finishes, you’ll see the message “All artifact data added to database. Please move on to `fill_media_table.py`!”

Run `fill_media_table.py` two (2) times

This file can be found in the *data-and-tables* folder.

Upon running, the terminal will print “Media data being gathered from the GSHImpact API! Please wait...”, and eventually “Media API call done! Adding items to the database...” Make sure you wait until you see the message “25 / 30 total rows of media data added to the database. Run the file again!” to hit run on the script again.

You’ll need to run this file three (2) times total. The first will add 25 rows of the data to the database. The second will add the remaining 5 items. After the second run of the file finishes, you’ll see the message "All media data added to database. Please move on to `calculations.py` in the calculations folder!" Now, it’s time to move on to running the visualizations and calculations.

Run `calculations.py` one (1) times

This file can be found in the *calculations* folder.

Upon running, the `calculations.txt` file will be created in the calculations folder. This is our formatted text file containing all of the calculations and detailed information about each. After the file finishes running, the terminal will print “Calculations text file created. Please move onto `graphs.py` in the graphs folder!”

Run `graphs.py` one (1) time

This file can be found in the *graphs* folder.

Upon running, the first visualizations of our data will open (as long as Matplotlib is installed or Anaconda is being used) and the user should see a set of box plots next to a horizontal bar chart.

Closing out this figure will result in the next figure opening, and so on until the user has seen all five figures. At this point, the terminal will print “Visualization files created. You're all done running scripts!”, and all of the generated .png files can be found in the *graphs* folder.

Documentation

fill_artifact_table.py		
Function Name & Purpose	Inputs	Outputs
set_up_database → Sets up a SQLite database connection and cursor	db_name → The name of the SQLite database	Tuple (Cursor, Connection) → A tuple containing the database cursor and connection objects
get_artifact_data → Scrapes artifact data from the Genshin Impact Wiki Artifacts/Sets page and inserts it into the Artifacts table	url → The URL of the Genshin Impact Wiki Artifacts/Sets page	all_media_data → List of data for each artifact on the Genshin Impact Wiki Artifacts/Sets page
setup_artifacts_table → Sets up the Artifacts table	cur → SQLite cursor object conn → SQLite connection object	None
insert_artifact_data → Inserts the media data for each character into the Media table	artifact_data → List of data for all artifacts, returned from get_artifact_data start → Integer starting point which to iterate from (remembers where it left off) end → Integer upper bound of where to iterate through (maximum number of items the API supplies) limit → Integer limit of how many items can be returned cur → SQLite cursor object conn → SQLite connection object	None
main → Sets up database Sets up artifact table Calls functions Inserts information into database		

fill_banner_table.py		
Function Name & Purpose	Inputs	Outputs
set_up_database → Sets up a SQLite database connection and cursor	db_name → The name of the SQLite database	Tuple (Cursor, Connection) → A tuple containing the database cursor and connection objects
get_banner_data → Gets banner data from the GSHIMPACT API	banner_url → The base URL for the GSHImpact API	all_media_data → List of GSHIMPACT json responses for each banner ID
setup_banners_table → Sets up the banner table	cur → SQLite cursor object conn → SQLite connection object	None
insert_banner_data → Inserts the media data for each character into the Media table	banner_data → List of data for all banners, returned from get_banner_data start → Integer starting point which to iterate through (remembers where it left off) end → Integer upper bound of where to iterate through (maximum number of items the API supplies) limit → Integer limit of how many items can be returned cur → SQLite cursor object conn → SQLite connection object	None
main → Sets up database Sets up banner table Calls functions Inserts information into database		

fill_character_table.py		
Function Name & Purpose	Inputs	Outputs
set_up_database → Sets up a SQLite database connection and cursor	db_name → The name of the SQLite database	Tuple (Cursor, Connection) → A tuple containing the database cursor and connection objects
get_character_data → Gets character data from the GSHIMPACT API	character_url → The base URL for the GSHImpact API	all_media_data → List of GSHIMPACT json responses for each character ID
setup_character_tables → Sets up the Characters and CharacterVisions table	cur → SQLite cursor object conn → SQLite connection object	None
insert_character_vision_table → Inserts the character visions data into the CharacterVisions table.	character_data → List of data for all characters, returned from get_character_data cur → SQLite cursor object conn → SQLite connection object	None
insert_character_data → Inserts the character data into the Characters table	character_data → List of data for all banners, returned from get_banner_data start → Integer starting point which to iterate from (remembers where it left off) end → Integer upper bound of where to iterate through (maximum number of items the API supplies) limit → Integer limit of how many items can be returned cur → SQLite cursor object conn → SQLite connection object	None
main → Sets up database Sets up character tables Calls functions Inserts information into database		

fill_media_table.py		
Function Name & Purpose	Inputs	Outputs
set_up_database → Sets up a SQLite database connection and cursor.	db_name → The name of the SQLite database	Tuple (Cursor, Connection) → A tuple containing the database cursor and connection objects
get_character_ids → Gets all of the character ID's from the Characters table	cur → SQLite cursor	character_ids → A list containing the integer character ID's from the Characters table
get_media_data → Fetches media data from the GSHIMPACT API for all character ids in the database	character_ids → Every character ID from the dataset, returned from get_character_ids character_url → The base URL for the GSHImpact API	all_media_data → List of GSHIMPACT json responses for each character ID
setup_media_table → Sets up the media table	cur → SQLite cursor object conn → SQLite connection object	None
insert_media_data → Inserts the media data for each character into the Media table	media_data → List of data for all banners, returned from get_banner_data start → Integer starting point which to iterate from (remembers where it left off) end → Integer upper bound of where to iterate through (maximum number of items the API supplies) limit → Integer limit of how many items can be returned cur → SQLite cursor object conn → SQLite connection object	None
main → Sets up database Sets up media table Calls functions Inserts information into database		

fill_weapon_table.py		
Function Name & Purpose	Inputs	Outputs
set_up_database → Sets up a SQLite database connection and cursor.	db_name → The name of the SQLite database	Tuple (Cursor, Connection) → A tuple containing the database cursor and connection objects
get_weapon_data → Gets weapon data from the Genshin.dev for all character ids in the database	weapon_url → The base URL for the Genshin.dev API	all_media_data → List of Genshin.dev json responses for each weapon
setup_weapons_tables → Sets up the Weapons and WeaponTypes table	cur → SQLite cursor object conn → SQLite connection object	None
insert_weapon_type_data → Inserts the weapon type data into the WeaponTypes table	weapon_data → List of data for all weapons, returned from get_weapon_data cur → SQLite cursor object conn → SQLite connection object	None
insert_weapon_data → Inserts the weapon data into the Weapons table	weapon_data → List of data for all banners, returned from get_banner_data start → Integer starting point which to iterate from (remembers where it left off) end → Integer upper bound of where to iterate through (maximum number of items the API supplies) limit → Integer limit of how many items can be returned cur → SQLite cursor conn → SQLite connection	None
main → Sets up database Sets up weapons tables Calls functions Inserts information into database		

calculations.py		
Function Name & Purpose	Inputs	Outputs
calculate_average_weapon_damage_per_rarity Calculates the average weapon damage per rarity (1 through 5 stars)	cur → SQLite cursor object	rarity_and_avg_attack → Dictionary where the keys are integers representing weapon rarity, and the values are floats representing average damage for weapons of that rarity
calculate_difference_awdpr Calculates the differences between each pair of rarities	awdpr → Dictionary of average weapon damage per rarity returned from calculate_average_weapon_damage_per_rarity	difference_awdpr → List of dictionaries where keys are each rarity and the values are the difference between the average damages for each rarity
calculate_average_weapon_damage_per_type Calculates the average weapon damage per weapon type (sword, bow, etc.)	cur → SQLite cursor object	rarity_and_avg_attack → Dictionary where the keys are integers representing weapon type, and the values are floats representing average damage for weapons of that type
calculate_difference_awdpt Calculates the differences between each pair of weapon types	awdpt → Dictionary of average weapon damage per type returned from calculate_average_weapon_damage_per_type weapon_types → List of weapon types returned from calculate_average_weapon_damage_per_type	difference_awdpt → List of dictionaries where the keys are each weapon type, and the values are a list of calculations comparing the difference between the average damages for each type
calculate_num_artifacts_per_quality Calculates the number artifacts for each quality (1 through 5 stars)	cur → SQLite cursor	num_artifacts_per_quality → Dictionary where the keys are integers representing artifact max quality, and the values are ints representing the count for artifacts of that max quality
calculate_num_media_per_character Calculates the number artifacts	cur → SQLite cursor	num_media_per_character → Dictionary where the keys are strs representing character

for each quality (1 through 5 stars)		names, and the values are ints representing the count for that character's total number of media appearances
main Creates the SQLite connection and cursor Calls each function and performs the calculations Creates the text file they are written into (calculations.txt)		

graphs.py		
Function Name & Purpose	Inputs	Outputs
get_weapon_stats Gathers all of the weapon information from the database	cur → SQLite cursor	weapons_d → Nested dictionary containing all of the weapon information used in visualizations
get_char_stats Gathers all of the character information from the database	cur → SQLite cursor	characters_d → Nested dictionary containing all of the character information used in visualizations
get_artifact_stats Gathers all of the artifact information from the database	cur → SQLite cursor	artifact_d → Dictionary containing all of the artifact information used in visualizations
box_rarity_versus_dmg Creates the box plot used to track rarity against damage and the horizontal bar graph of number of weapons by rarity	weapons_d → Dictionary with each weapon and its statistics	None
char_vision_vs_weapon Creates the bar charts used to compare weapon frequency by character vision	chars_d → Dictionary with each character and their information	None

chars_by_vision Plots the distribution of characters of each type of vision they have using a pie chart	chars_d → Dictionary with each character and their information	None
artifact_qual_dist Plots the distribution of artifacts' maximum set quality using a pie chart	artifact_d → Dictionary with each artifact and its maximum set quality	None
main Creates the cursor connection Calls the functions to create the charts		

Resources

Date	Issue Description	Location of Resource	Result
Nov 17	Trouble understanding how to actually use APIs	GSHImpact	We used the endpoint under the playground to understand how the API calls worked
Dec 2	Couldn't get the database to be created	Week 11 Slides	Referenced the slides to get the language for the SQLite database.
Dec 4	Issues getting data to insert into the SQLite database	Office Hours	Learned how to use the table methods and was able to get data inserted and organized into the database successfully
Dec 4	Not sure how to format data in a way that can be used to make the plots we needed	Discussion 13	Learned that we can use lists or loop through dictionaries/tuples to populate the plots with the right information
Dec 4	General Matplotlib formatting issues	Matplotlib Cheat Sheet (Week 13 Module)	Learned how to make the different types of graphs we needed, format the plots/subplots, label, and colors
Dec 5	Linking tables and removing string data	Forum Post	Helped us understand how we can write the same data to different tables to organize them
Dec 6	General debugging issues – errors recurring during simplification of code	ChatGPT	ChatGPT pointed out issues like referencing the wrong variable and using the wrong url to call the wrong API and those issues were amended.
Dec 9	Trouble getting 25 items at a time into the database	11/25/2024 Lecture recording	Ended up using some of the logic from both Prof. Ericson's and past student's examples to figure out how to best implement the 25 limit into our code