Technology	Choice
Platform	Databricks (Community Edition)
IDE	Databricks Notebooks
Processing Engine	Apache Spark(pyspark)
Visualization	Seaborn python library
Languages	SQL and Python

Why the above tools/technology were chosen?

I have chosen **Databricks** as the platform to perform the required Data Analysis due to the following reasons -

- (1) Provides ability to upload CSV files and store data in Databricks File System(DBFS).The data can be accessed from DBFS as though you would query a typical RDBMS Table.
- (2) Provides Notebooks using which Python and SQL code can be written and executed on the same storage layer.
- (3) Can handle Structured and Semi-structured data(columns having data in JSON format in this take home test) equally well
- (4) Prints the cell(python or SQL) execution time automatically
- (5) Provides the ability to generate output of an SQL query in a tabular format or render charts.
- (6) Built to handle Big Data.The below presented code should work with a reasonable performance for actual production volumes.

seaborn was chosen for visualization as it provides the ability to embed the charts in the same platform(Databricks Notebooks) thereby eliminating the need to -

- (a) perform data export onto another BI tool (like Tableau,PowerBI) or

- (b) switch applications to view the analysis and charts respectively

seaborn is a good fit for serving this specific purpose of generating graphs for exploratory analysis.However , I acknowledge the need to use core BI tools for generating dashboards and sharing them with a large number of people in the organization.

Start Analysis

Please Note - The default cell type is Python. However, There are many cells that use SQL which are denoted by starting line "%sql"

Ingest the data from CSV files provided into Databricks lakehouse

```
#########################
# File location and type
#########################
file_type = "csv"

# Remove previous associations if any
dbutils.fs.rm("dbfs:/user/hive/warehouse/bugs", True)
dbutils.fs.rm("dbfs:/user/hive/warehouse/bugtracking", True)
dbutils.fs.rm("dbfs:/user/hive/warehouse/progress_status_lookup", True)
dbutils.fs.rm("dbfs:/user/hive/warehouse/single_player_pond", True)

#dbutils.fs.ls("dbfs:/user/hive/warehouse/")

bugs_file_location = "/FileStore/tables/Bugs.csv"
bugtracking_file_location = "/FileStore/tables/BugTracking.csv"
progress_status_file_location = "/FileStore/tables/Progress_Status_Lookup.csv"
singleplayer_pond_file_location =
"/FileStore/tables/Single_Player_Pond_Dataset.csv"

####################
# CSV options
####################
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

####################
# Build a Pyspark dataframe by loading data from CSV file
####################
bugs_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(bugs_file_location)

bugtracking_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(bugtracking_file_location)

progress_status_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(progress_status_file_location)
```

```
singleplayer_pond_df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("escape", "\"") \
  .option("sep", delimiter) \
  .load(singleplayer_pond_file_location)

########################
# create tables from the respective DataFrames.
# Once saved, these tables will persist across cluster restarts as well as
allow various users across different notebooks to query this data.
########################

bugs_permanent_table_name = "bugs"
bugtracking_permanent_table_name = "bugtracking"
progress_status_permanent_table_name = "progress_status_lookup"
singleplayer_pond_permanent_table_name = "Single_player_pond"

# Create a permanent table from the above dataframe
bugs_df.write.format("parquet").saveAsTable(bugs_permanent_table_name)
bugtracking_df.write.format("parquet").saveAsTable(bugtracking_permanent_table_
name)
progress_status_df.write.format("parquet").saveAsTable(progress_status_permanen
t_table_name)
singleplayer_pond_df.write.format("parquet").saveAsTable(singleplayer_pond_perm
anent_table_name)
```

Verify the tables are created

```
%sql show tables
```

	database	tableName	isTemporary
1	default	bugs	false
2	default	bugtracking	false
3	default	progress_status_lookup	false
4	default	single_player_pond	false

Showing all 4 rows.

Question One:

a) Please provide the SQL query that would accurately join the provided tables: ‘Bugs’, ‘BugTracking’, and ‘ProgressStatusLookup’

Answer : Query below

```
%sql

CREATE OR REPLACE TEMPORARY VIEW bugs_complete_view AS
SELECT
   bugs.*
  ,bugtracking.* EXCEPT(bugtracking.projectid, bugtracking.bugid,
bugtracking.`game area`)
                    -- Do not print projectid, bugid from bugtracking(as these
are already printed from bugs)
  ,progress_status_lookup.progressstatusname
FROM
  bugs
LEFT OUTER JOIN bugtracking
    ON bugs.projectid = bugtracking.projectid AND bugs.bugid =
bugtracking.bugid
    -- Used LEFT OUTER JOIN to be on the safer side(Could have used JOIN
instead as bugtracking has a corresponding record for every bugid from bugs)
LEFT OUTER JOIN progress_status_lookup
    -- Definitely have to use LEFT OUTER JOIN here as progressstatusname is not
defined for few progress status IDs
    ON bugs.projectid = progress_status_lookup.projectid AND
bugtracking.`progress status id`=progress_status_lookup.progressstatusid
```

OK

(b) How many distinct bugs with an ‘In Progress’ progress status exist in this joined table?

Answer : 45 (SQL below)

```
%sql

SELECT COUNT(DISTINCT projectid,bugid) FROM bugs_complete_view WHERE
progressstatusname='In Progress'
```

	count(DISTINCT projectid, bugid)
1	45

Showing all 1 rows.

(c) How many distinct bugs have a null value in either the ProgressStatusName or Priority field in the joined table?

Answer : 100 (SQL below)

```
%sql

SELECT COUNT(DISTINCT projectid,bugid) FROM bugs_complete_view WHERE
progressstatusname IS NULL OR priority = 'NULL'
```

	count(DISTINCT projectid, bugid)
1	100

Showing all 1 rows.

What could be the possible causes of this?

Answer :

(a) ProgressStatusName - It appears that whenever a bug is opened , There is a record inserted in bugtracking with progresstatusid as Blank or NULL.

Considering the timestamp of this NULL record is same as that of the next sequential record with status as 'Confirmed' , It may be OK to ignore the NULL record for any analytics.

(b) Priority - The reason this field has NULL values is probably due to the tester not assigning a priority intentionally(due to him/her not being aware of the right priority to assign) or unintentionally(due to priority not being a mandatory field in the bug tracking tool)

d) Please answer the following observational questions about the data:

i. What is the average Open Bug Life for each Severity Level?

Methodology

I followed 2 alternate approaches -

- Approach1 : Open Bug Life = {[Date Assigned when progress status LIKE Closed] - [Date Assigned when progress status = Confirmed]} from "BugTracking"

- Approach2 : Open Bug Life = {[AssignedTime] - [CreationTime]} from "bugs"

Both the approaches yielded the same result. However , My solution did not match with any of the 4 answer options provided.

Answer : A = 6.41, B = 18.51, C = 24.08, D = 34 (SQLs below)

```
%sql

-- Approach1
SELECT
  severity,
  COUNT(1) AS No_of_bugs,
  ROUND(AVG(bug_open_days),2) AS Average_Open_Bug_Life
FROM
  (
    SELECT
      projectid,
      bugid,
      severity,
      MIN(to_timestamp(dateassigned, 'h:m a M-d-yyyy')) AS bug_opened_time, --
MIN is used to pick the earliest record when there is more than one record
having progressstatus='Confirmed'
      MAX(to_timestamp(dateassigned, 'h:m a M-d-yyyy')) AS bug_closed_time,--
MAX is used to pick the latest record when there is more than one record having
progressstatus Like Closed
      datediff (MAX(to_timestamp(dateassigned, 'h:m a M-d-yyyy'))
,MIN(to_timestamp(dateassigned, 'h:m a M-d-yyyy')) ) AS bug_open_days
    FROM bugs_complete_view
    WHERE (progressstatusname='Confirmed' OR progressstatusname LIKE
'%Closed%')
    AND severity IN ('A','B','C','D')
    GROUP BY
      projectid,bugid,severity
  )
GROUP BY
  severity
ORDER BY
  severity
```

	severity	No_of_bugs	Average_Open_Bug_Life
1	A	17	6.41
2	B	39	18.51
3	C	36	24.08
4	D	2	34

Showing all 4 rows.

```
%sql

-- Approach2
SELECT
  severity,
  ROUND(AVG(bug_open_days),2) AS Average_Open_Bug_Life
FROM
(
  SELECT *,
    datediff (to_timestamp(assignedtime_local, 'h:m a M-d-yyyy')
,to_timestamp(createdtime_local, 'h:m a M-d-yyyy')) AS bug_open_days
  FROM
    bugs
  WHERE
    severity IN ('A','B','C','D')
)
GROUP BY
  severity
ORDER BY
  severity
```

	severity	Average_Open_Bug_Life
1	A	6.41
2	B	18.51
3	C	24.08
4	D	34

Showing all 4 rows.

ii. Within the "PR - Problem Report" Issue Type in the table, how many null Priority values are there?

Answer : 40 - Option C (Query below)

```
%sql

SELECT COUNT(1) FROM bugs WHERE issuetype='PR - Problem Report' AND priority =
'NULL'
```

	count(1)

1	40

iii. Which Test Team found the most Severity = "B" bugs, in the Bugs table?

Answer : TT5 - Option B (Query below)

```
%sql

SELECT * FROM
(
    SELECT
      testteam,
      COUNT(1) severity_b_bugs_found
    FROM
      bugs
    WHERE
      severity='B'
    GROUP BY
      testteam
    ORDER BY
      2 DESC            -- Order by bugs found in descending order
)
LIMIT 1     -- Pick only the first record which has found the most bugs with
severity=B
```

	testteam	severity_b_bugs_found
1	TT5	18

• Question Two:

a) Visualize and explain (in 150 of your own words or less) one of the following KPIs (choose whichever you find most interesting).

My choice of KPI is KPI 3: Bug Find – The number of bugs found over time

• What business value do you think this KPI could provide?

Answer :

This KPI can provide valuable insights when overlayed with different dimensions as below :

- Overlay with software release data - Are there significant spikes in bugs found during certain months when new versions of the game software was released?
- Overlay with Testteam data - Are the spikes due to a new test team onboarded who recorded a higher number of bugs(false positives) due to gaps in training ?
- Overlay with public holiday or End of year shutdown periods - Were lesser bugs recorded due to optimally performing game software or was it due to reduced testing activity(due to testing team on holiday)

• Please create a running total visualization depicting number of bugs found over time.

```
%sql

SELECT
  SUM(Bugs_Found) OVER (ORDER BY bug_created_time) as Running_Total_Bugs_Found,
  bug_created_time
FROM
(
  SELECT
    COUNT(1) as Bugs_Found,
    TO_DATE(to_timestamp(createdtime_local, 'h:m a M-d-yyyy'),'M-d-yyyy') AS
bug_created_time
  FROM
    bugs
  GROUP BY
    TO_DATE(to_timestamp(createdtime_local, 'h:m a M-d-yyyy'),'M-d-yyyy')
)
ORDER BY
  bug_created_time
```

	Running_Total_Bugs_Found	bug_created_time
1	10	2015-04-03
2	40	2015-04-23
3	50	2015-05-29
4	64	2015-06-15
5	80	2015-06-16
6	100	2015-08-08

Showing all 6 rows.

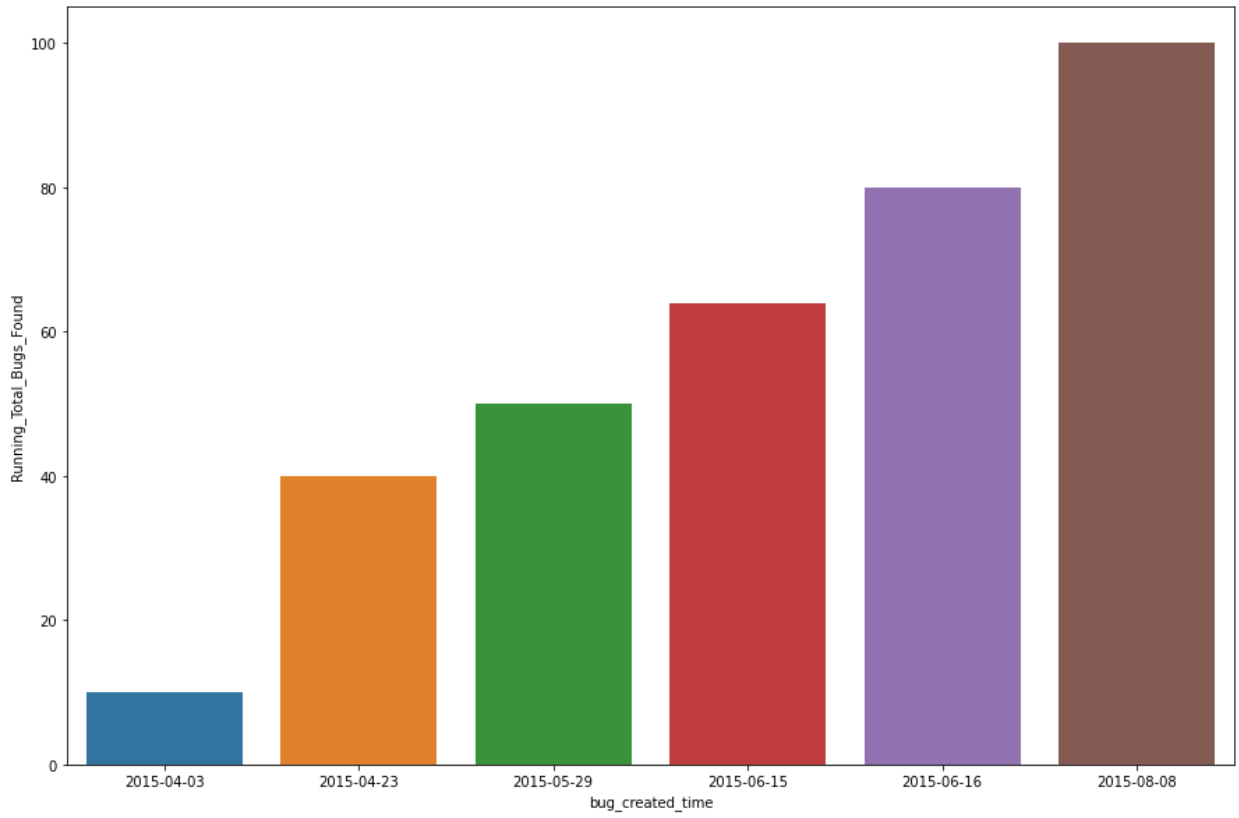
Answer : Visualization below

```
import seaborn as sns
from matplotlib import rcParams

# The query shown in previous cell is used
bugs_found_df=spark.sql(f"SELECT \
  SUM(Bugs_Found) OVER (ORDER BY bug_created_time) as Running_Total_Bugs_Found, \
  bug_created_time \
  FROM (SELECT COUNT(1) as Bugs_Found, TO_DATE(to_timestamp(createdtime_local, 'h:m a M-d-yyyy'),'M-d-yyyy') AS bug_created_time FROM bugs GROUP BY TO_DATE(to_timestamp(createdtime_local, 'h:m a M-d-yyyy'),'M-d-yyyy') ) ORDER BY bug_created_time")

bugs_found_df_pandas = bugs_found_df.toPandas()      # Converting from spark dataframe to pandas dataframe for plotting using seaborn

rcParams['figure.figsize'] = 15,10
ax = sns.barplot(x="bug_created_time", y="Running_Total_Bugs_Found", data=bugs_found_df_pandas)
```



- **Question Three:**

The dataset in the "Single Player Pond Dataset" file is a log of in-game actions performed by a single given player. Given this data, please answer the following questions:

(a) The event_name field indicates which action the player is performing. What is this particular player's overall shot to score ratio? Please explain your methodology.

Methodology:

Shot to score ratio (STSR) is assumed to be a measure of how successful the player is able to convert his goal shoot attempts to actual goals.

STSR is expressed as a percentage below -

```
STSR = Total goals scored *100 / Total shots attempted
```

Answer : Shot to score ratio = 6.25% (i.e. The player roughly scores around 6 goals for every 100 shots he attempts)

Query below

```
%sql

SELECT ROUND(COUNT(1)*100 / (SELECT COUNT(1) FROM single_player_pond WHERE event_name='shot'),2) as STSR FROM single_player_pond WHERE event_name='score'
```

	STSR
1	6.25

Showing all 1 rows.

(b) There is additional information about the pass event in the event_params field. Create a visualization showing the overall breakdown of the different pass types that the player performed. Please explain your methodology and reasoning behind your choice of visualization.

Methodology:

- Step 1 : Understand the JSON data in event_params as to what key and values(and their data type) it has

- Step 2 : Create a view by transforming the data from 'String' datatype to 'Struct' data type so as to be able to query the

embedded JSON elements as though they were a column by themselves

- Step 3 : Query the individual keys from transformed view and ensure the data looks right

- Step 4 : Write the SQL Query to get a breakdown of different pass types

- Step 5 : Load the output of above query into a dataframe needed to generate the visualization

- Step 6 : Generate the visualization using seaborn python library

Step 1 : Understand the JSON data in event_params as to what key and values(and their data type) it has 👇

```
%sql

-- Pick a sample record from the table where event_name='pass'
-- Pass that sample data from event_params field into "schema_of_json" function
to understand the data type of "value" belonging to each "key"
SELECT
schema_of_json('{"mid":"H2HS_5049060562","clock_time":16778156,"type":"GROUND",
"passer_char":168542,"passer_attr":92,"receiver_char":222492,"start_loc":
[112.679397583,2.083006859,23.990615845],"target_loc":
[111.396255493,0.100000001,57.968891144],"receive_loc":
[106.728500366,0.756773651,52.239437103],"pass_details":
["FIRST_TOUCH"],"pass_result_details":
["GOOD","COMPLETE"],"touch_part":"RFOOT","pass_diff":2.4,"power_meter":0.28,"pa
ss_rel_angle":0.51}
') as event_param_schema;
```

	event_param_schema
1	STRUCT<clock_time: BIGINT, mid: STRING, pass_details: ARRAY<STRING>, pass_diff: DOUBL pass_result_details: ARRAY<STRING>, passer_attr: BIGINT, passer_char: BIGINT, power_meter ARRAY<DOUBLE>, receiver_char: BIGINT, start_loc: ARRAY<DOUBLE>, target_loc: ARRAY<DC type: STRING>

Showing all 1 rows.

Step 2 : Create a view by transforming the data from 'String' datatype to 'Struct' data type so as to be able to query the embedded JSON elements as though they were a column by themselves 👇

```
%sql

-- Use the above information to create a Temporary view so as to easily query
the embedded keys and values within event_params field
-- As an alternative , a Table could have been created here instead of a view.
-- Considering this is a one-off analysis and to save storage space ,I went
with the temporary view option.

CREATE OR REPLACE TEMP VIEW single_player_pond_parsed AS
SELECT player_id,event_name,event_step,
from_json(event_params,'STRUCT<clock_time: BIGINT, mid: STRING, pass_details:
ARRAY<STRING>, pass_diff: DOUBLE, pass_rel_angle: DOUBLE, pass_result_details:
ARRAY<STRING>, passer_attr: BIGINT, passer_char: BIGINT, power_meter: DOUBLE,
receive_loc: ARRAY<DOUBLE>, receiver_char: BIGINT, start_loc: ARRAY<DOUBLE>,
target_loc: ARRAY<DOUBLE>, touch_part: STRING, type: STRING>') as
event_params
from single_player_pond WHERE event_name='pass'
```

OK

Step 3 : Query the individual keys from transformed view and ensure the data looks right 👇

```
%sql

DESCRIBE single_player_pond_parsed
-- As can be seen below , event_params is now a STRUCT data type instead of a
STRING
-- Thus helps to query the individual elements within it
```

	col_name	data_type
1	player_id	double

2	event_name	string
3	event_step	int
4	event_params	struct<clock_time:bigint,mid:string,pass_details:array<string>,pass_diff:dc array<string>,passer_attr:bigint,passer_char:bigint,power_meter:double,re art_loc:array<double>,target_loc:array<double>,touch_part:string,type:stri

Showing all 4 rows.

```
%sql

-- Lets have a look at the transformed data(though it is just a data type
change) in our temporary view
-- We are now able to query on shot type as though it was a seperate column by
itself
SELECT *,event_params.type FROM single_player_pond_parsed
```

	player_id	event_name	event_step	event_params
1	1000000000000	pass	1321	▸ {"clock_time": 16778069, "mid": "pass_diff": 0.46, "pass_rel_angle": "passer_char": 176580, "power_met "receiver_char": 158023, "start_loc' 0.100000001, -2.792277336], "tou
2	1000000000000	pass	1326	▸ {"clock_time": 16778080, "mid": "pass_diff": 0.31, "pass_rel_angle": "SHOOTING_CHANCE"], "passer_at 0.446851403, 26.954238892], "re "target_loc": [125.642990112, 0.10(
3	1000000000000	pass	1330	▸ {"clock_time": 16778084, "mid": "INTO_CENTRE"], "pass_diff": 0.1, " 76, "passer_char": 231443, "power_ "receiver_char": 168651, "start_loc' 0.100000001, 20.310703278], "tou
4	1000000000000	pass	1336	▸ {"clock_time": 16778100, "mid": ' "pass_rel_angle": 0.03, "pass_result "WING_OPPORTUNITY", "COMPLE [148.833145142, 0.365867645, 51.

Showing all 286 rows.

Step 4 : Write the SQL Query to get a breakdown of different pass types 👇

```
%sql
-- Below query gets the no. of occurences/records for each "shot_type"
-- It is evident that "GROUND" pass type is the clear winner

SELECT COUNT(1) occurences ,event_params.type shot_type FROM
single_player_pond_parsed GROUP BY event_params.type ORDER BY 1 DESC
```

	occurences	shot_type
1	205	GROUND
2	30	THROUGH_GROUND
3	13	CLEARANCE
4	8	GROUND_HARD
5	5	CROSS_GROUND
6	4	LOB
7	3	THROUGH_GROUND_HARD
8	3	GK_SIDE_KICK
9	3	CROSS_HIGH
10	3	LOB_CORNERKICK
11	2	KICKOFF_ONEPLAYER
12	1	GK_KICK
13	1	LOB_GOALKICK_HIGH
14	1	THROUGH_LOB
15	1	THROWIN
16	1	CROSS
17	1	LOB_PLACEKICK
18	1	LOB_GOALKICK

Step 5 : Load the output of above query into a dataframe needed to generate the visualization 👇

```
# Load the data from temporary view into a dataframe for generating
visualization
shot_type_pysparkDF=spark.sql('SELECT event_params.type AS
shot_type,event_params.touch_part  FROM single_player_pond_parsed ')
shot_type_pandasDF = shot_type_pysparkDF.toPandas() # Converting from spark
dataframe to pandas dataframe for plotting using seaborn
```

Step 6 : Generate the visualization using seaborn python library 👇

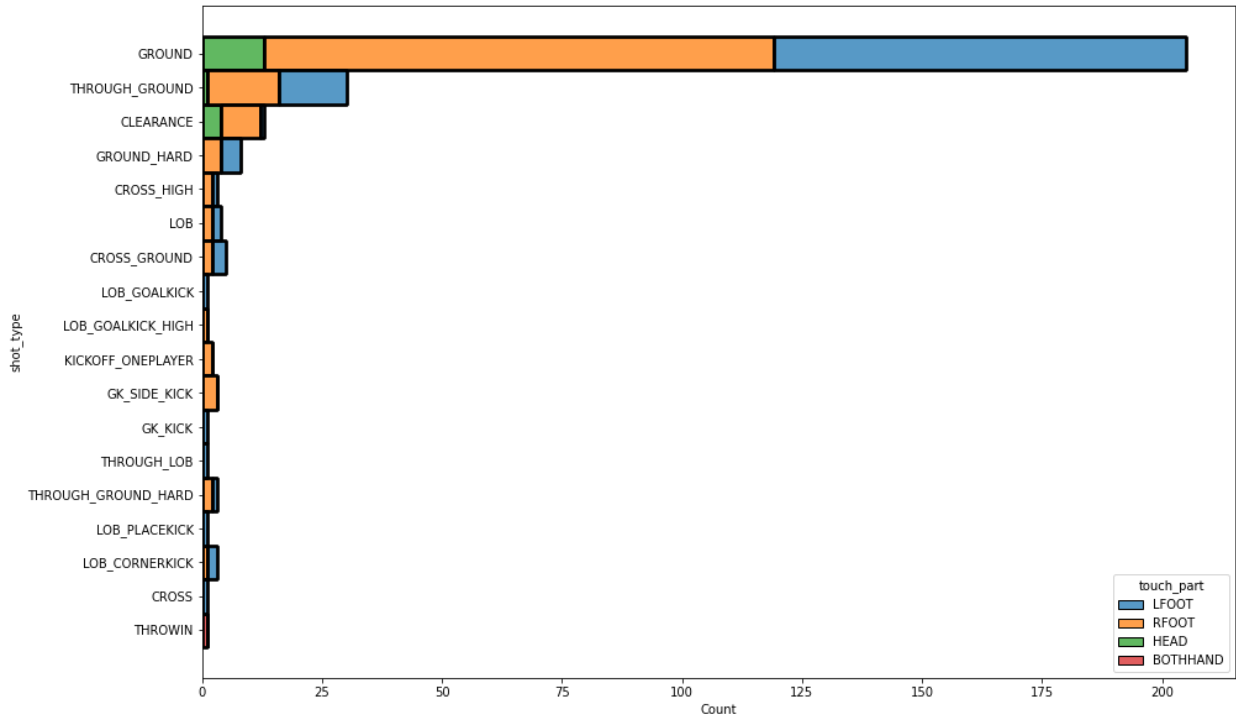
```
# seaborn python library is used for visualization
import seaborn as sns
from matplotlib import rcParams

# "Histogram" is the chart type used to understand the distribution/frequency
of different shot_types by the player
# The hue/color indicates the body part used to play that shot as well

# The below Viz. shows that -
# "GROUND" is the highest shot_type used by the player
# "BOTHHAND" was used only when THROWING" (which seem to make sense and provide
confidence that the data is clean)

rcParams['figure.figsize'] = 15,10
sns.histplot(data=shot_type_pandasDF,y="shot_type",hue="touch_part",multiple="s
tack",legend=True)

Out[6]: <AxesSubplot:xlabel='Count', ylabel='shot_type'>
```



Reasoning behind choice of visualization

I have chosen Histogram as it best provides the distribution/frequency of different shot_types by the player.

End Analysis.

Feedback

The data dictionary document states that "Progress Status ID" is a column in "Bugs" table which doesnt seem to be the case.

Can you please check and correct the data dictionary if needed?

Self Reflection

I personally enjoyed working on this take home test which helped me get a good idea of how game test data might look like.

I can also imagine the daily volume of this test data might be huge(truly Big data!) considering the variety of games and platforms available and number of test teams performing the testing.

It also helped me to reaffirm my Analytic skills in SQL,Python and Visualization.Thank you!