# Visualization of data from League of Legends esport matches

Zdeněk David

May 12, 2022

# 1 Introduction

All the data is polled using API endpoints from https://vickz84259.github.io/lolesports-api-docs/. For each league we can retrieve its match schedule. For this project I have chosen a single league to make things easier: LEC - League of Legends European Championship. The match schedule contains all the finished and scheduled games in the split. I will retrieve all the data for the finished games and will be focusing on ways to create easy to understand visualizations of this data in this analysis.

First and foremost, it will be important to determine what to look for in the provided data and what may be visualized. After deciding on the issue of this investigation, the next step will be to determine how the data may be displayed, compare various ways, and choose at least one that best fits the situation.

## 2 Implementation of polling data

Because of my custom made assignment topic, I had to download the data first, in case the API endpoints stopped working or have been modified - which as stated on the website could happen any time.

I started with mapping the API myself in Postman, because as you can see upon visiting the API page, there is almost no documentation as it is supposed be used internally only. First, we have to poll data about all the Leagues, so we can get our specific league ID - in our case we want LEC. My implementation can be seen in the code below:

```
# Declaring variables
headers = {
    'x-api-key': '0TvQnueqKa5mxJntVWt0w4LpLfEkrV1Ta8rQBb9Z'
}
parameters = {
    'hl': 'en-US'
}

# API endpoint
getLeagues = 'https://esports-api.lolesports.com/persisted/gw/getLeagues'

# Storing the request into 'data' variable
data = rq.get(getLeagues, params=parameters, headers=headers)
data = data.json()['data']['leagues']

lecID = 0
for i in data:
    for key, value in i.items():
        if value == 'LEC':
            lecID = i['id']

print(lecID)
```

Now, having the LEC league ID, we can poll its schedule. So, we can use the /getSchedule endpoint and add the league ID from before as a parameter. Very similarly to before, on the code snippet below can be seen the important part - filtering all the match IDs from the schedule.

```
# Storing the request into 'data' variable
data = rq.get(getSchedule, params=parameters, headers=headers)
data = data.json()['data']['schedule']['events']

ids = []
for event in data:
    ids.append(event['match']['id'])

print(ids)
```

Due to the fact that we did not need any additional information at this time, we got the match details from the /getWindow and /getDetails endpoints in Postman. However, we are likely going to poll all the data using a Python script later. The code is stored in a GitHub repository at https://github.com/zdenduk/lolesports-data. All the polled JSON files can be found in a Google Drive folder here https://drive.google.com/drive/folders/1__qHpINEqFdlVtZY6lcbbnJY6deF7vYR?usp=sharing.

# 3 Data structure

We can either get match details for a finished game or we can poll data in a specific moment in a game. In the first case, we can use techniques to visually encode player statistics. In the latter, we are going to need to use techniques to visualize time oriented data.

Using the /getSchedule endpoint, we can poll all games scheduled to be played in the League. There are a total of 80 finished games. Each game is stored in a JSON as an event object, containing important data like starting time of the game, in which week of the season the game was played, match ID, both opposing teams with its names, logos, record and most importantly the result of the game. You can see an example below - data from SK Gaming vs Fnatic in week 1.


```
{
    "startTime":"2022-01-16T19:00:00Z",
    "state":"completed",
    "type":"match",
    "blockName":"Week 1",
    "league":{
        "name":"LEC",
        "slug":"lec"
    },
    "match":{
        "id":"107417059263300159",
        "flags":[
            "hasVod"
        ],
        "teams":[
            {
                "name":"SK Gaming",
                "code":"SK",
                "image":"http://static.lolesports.com/teams/1643979272144_SK_Monochrome.png",
                "result":{
                    "outcome":"loss",
                    "gameWins":0
                },
                "record":{
                    "wins":7,
                    "losses":11
                }
            },
            {
                "name":"Fnatic",
                "code":"FNC",
                "image":"http://static.lolesports.com/teams/1631819669150_fnc-2021-worlds.png",
                "result":{
                    "outcome":"win",
                    "gameWins":1
                },
            "record":{
                "wins":13,
                "losses":5
            }
        }
    ],
    "strategy":{
        "type":"bestOf",
        "count":1
    }
}
```


Starting with the finished game data - we have participant metadata for each team - for each of the 5 players, we can find multiple IDs for later purposes, player name, the champion he was playing and his role within the team. You can see an example in the JSON snippet below:

```
{
    "participantId": 1,
    "esportsPlayerId": "101389749294612370",
    "summonerName": "SK Jenax",
    "championId": "Gragas",
    "role": "top"
}
```


Then, we have an object with the important information about the game that took place. First, there is total gold, inhibitors and towers destroyed, sum of kills and barons and dragons slain. After that follows all participant info. Participants are labeled using their participant ID. You can see below in a JSON snippet, that participant with ID of 1 - we can retrieve from the previous info we have that it is SK Jenax, has aquired a total of 13061 gold, leveled up to level 17, ended the game with a KDA of 3/3/3 and has farmed 287 minions.


```
{
    "participantId": 1,
    "totalGold": 13061,
    "level": 17,
    "kills": 3,
    "deaths": 3,
    "assists": 3,
    "creepScore": 287,
    "currentHealth": 1212,
    "maxHealth": 2343
}
```

# 4 Data content

As described in the previous chapter, we will be mainly using data from the finished games. Most attributes contain numerical values, always being positive integers representing occurrences of some action. The Figure 1 below illustrates how all the available attributes can be visualized in the form of a table chart. While it shows a lot of data, it does not show correlation between any of them.

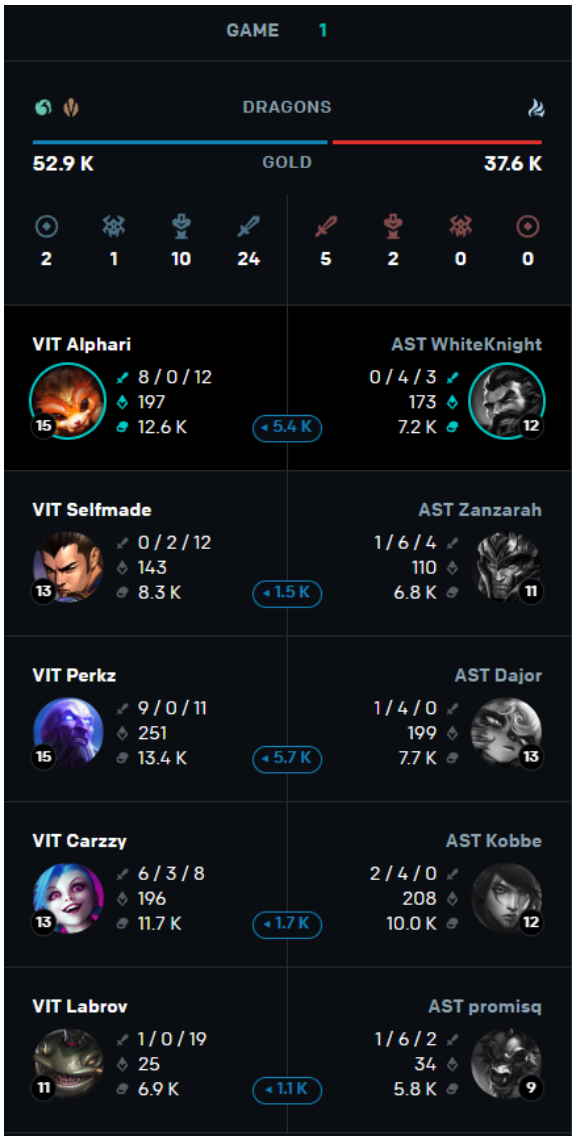



Figure 1: An example of data visualization of a finished game [1]

The usual statistical data calculations can therefore be performed on these data calculations such as mean, median, standard deviation, correlation, minimum, maximum and much more. We can also create filters to select samples - for example calculate winrate for a specific champion only for players that played X or more games on him.

It would also be possible to analyze some correlation, between individual data components. E.g. if in general, the more kills a player has, the more winrate his team has (meaning he has a positive impact towards his team winning), and the like. However, we immediately encounter several problems.

Firstly, some roles and specifically champions require more resources from the map. Hence, we would be handicapping weak side players ("Weak side" is a phrase used by casters to refer to the side of the map that the jungler does not gank or spend much time on. Although this can technically refer to bottom or top lane, it is most often applied to top. Champions or players that are good at in this situation are also called weak side champions or weak side players. Its opposite is the strong side.[2]) or low resource players in general.

Secondly, the data surely has multivariate dependencies. For example, the number of kills

influences also the gold acquired, which also contributes to additional advantage for the player. We could also include all variables, but that would be difficult to create.

# 5 Visualization techniques

The purpose of this visualization should be to determine best teams or individual players based on statistical data. Additionaly, we could want to compare multiple targets (e.g. teams, players).

## 5.1 Visual encoding of attributes of time-oriented data

If we were to only visualize data from finished games - we have quite a lot of attributes to show for each game, not to mention there is 80 games in total. We are dealing with tabular data with a lot of attributes, to be exact. Most of the attributes are of the L type.

What I think is the best way to visualize this data is a set of simple graphs that will interactively visualize user-selected statistics. This means that one attribute (sometimes combined multiple attributes - like KDA) from the dataset will be displayed on each graph. This reduced the problem of multidimensional data visualization to a simple problem visualization of time-oriented data. One example could be to select an attribute, then we could use a simple line chart, where on the x-axis there will be a game value (e.g. Week 1 game 1) and on the y-axis the corresponding value of the examined attribute. An example can be seen below in Figure 2.

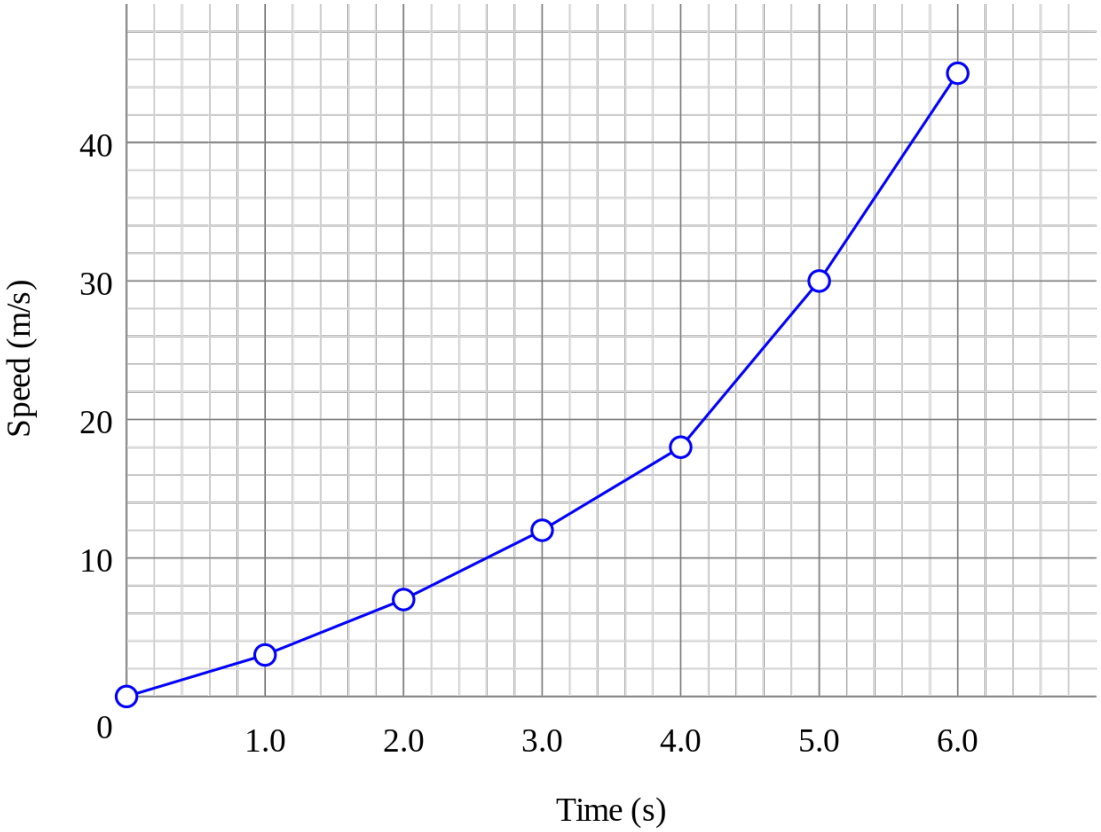



Figure 2: Line chart example [3]

If it is necessary to manage to compare several players at the same time, it will plot multiple lines for multiple players, but all these values will be from the same attribute, as it would not make sense to plot more different attributes.

When the values of several user-selected players are in one graph, they must somehow differ in order to know which player belongs to which line. An easy approach to that is mapping the players to colors.

Another option would be to create a heatmap of selected attribute per game for selected player-s/teams. For example KDA per game could be done similarly to how it is done in Figure 3, where better KDA corresponds to bolder color, on the x-axis there will be a game value (e.g. Week 1 game 1) and on the y-axis selected players/teams.

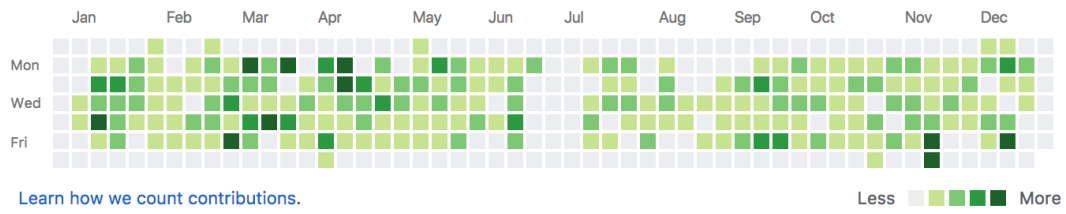



Figure 3: Heatmap example [4]

A different possibility would be a theme river layout. Its big advantage lies in easily identifying trends and being easy to understand. For example KDA or any other attribute like gold could be plotted similarly to how it is done in Figure 4 below, where each stream represents a player/team, on the x-axis there will be a game value (e.g. Week 1 game 1) and on the y-axis selected attribute.

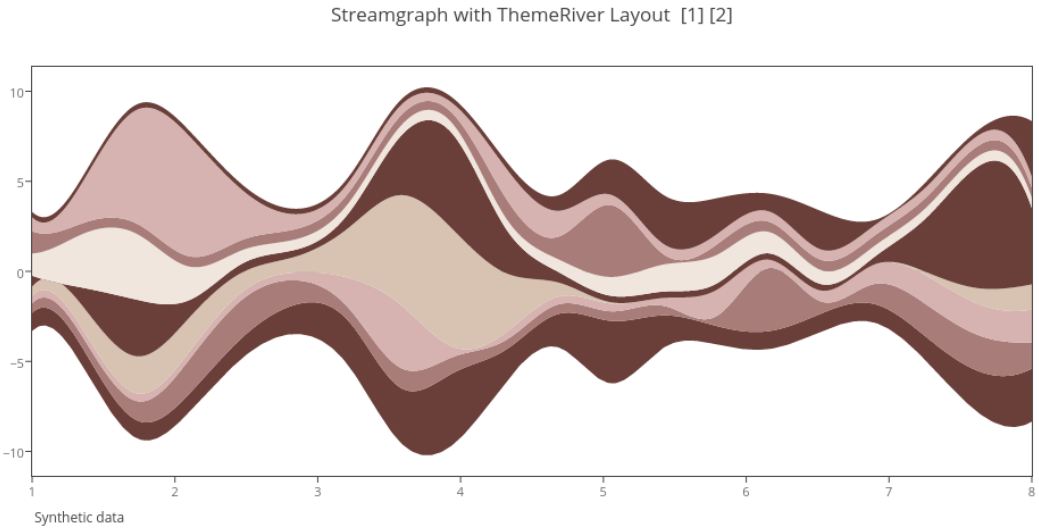



Figure 4: Theme river layout example [5]

An additional approach could be a bar graph. It is similar to a theme river layout, but offers easier reading of attribute values at the cost of not being able to see trends that fast. Again, for example KDA or any other attribute like gold could be plotted similarly to how it is done in Figure 5 below, where part of bar represents a player/team, on the x-axis there will be a game value (e.g. Week 1 game 1) and on the y-axis the selected attribute.

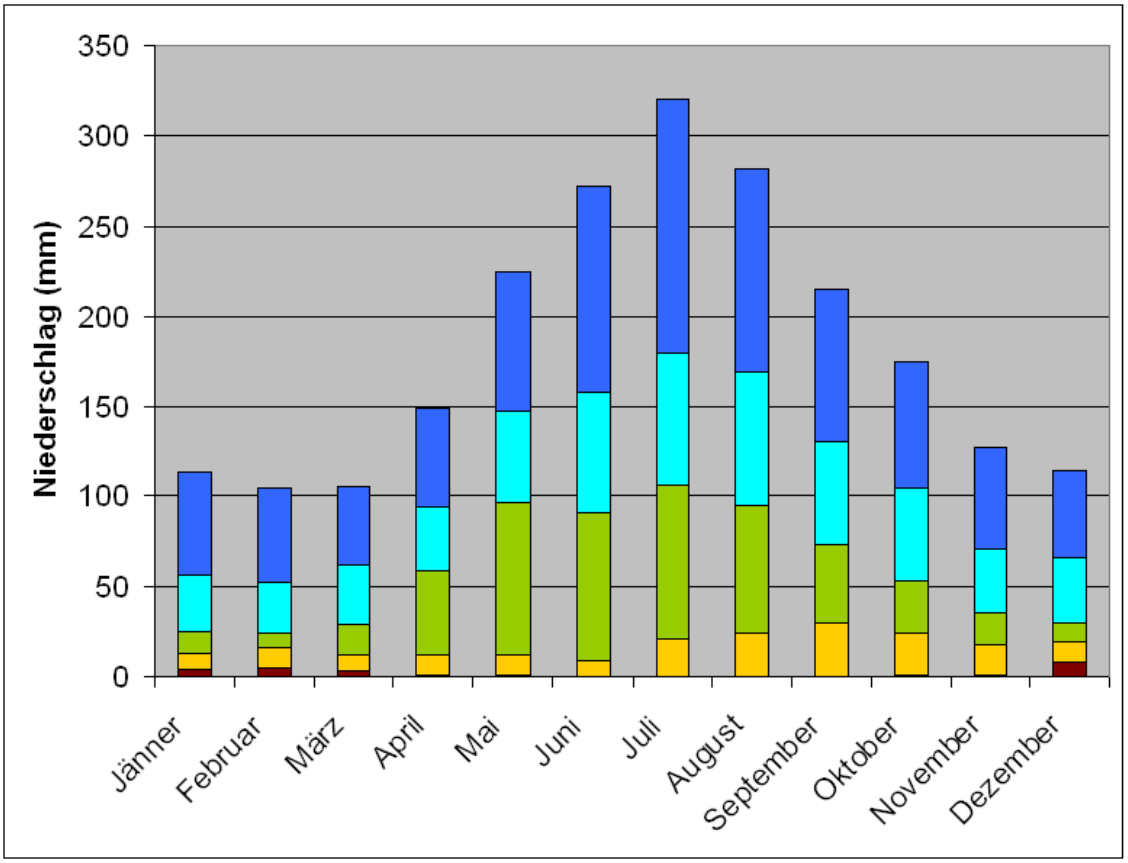



Figure 5: Stacked bar graph example [6]

A completely alternative approach would be to visualize the data by animation. Time is no longer represented as an axis of visualization in animations. However, an application that allows users to travel interactively to specific points in time gives them a strong understanding of the facts at that time.

Deviations and the lack of a characteristic at a certain moment in time may be plainly noticed in the playing animation. However, the whole picture of the patterns is lost. In this instance, the ThemeRiver may be preferred.

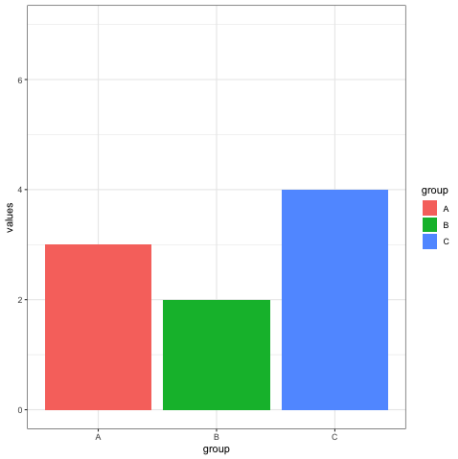



Figure 6: First frame of the animation

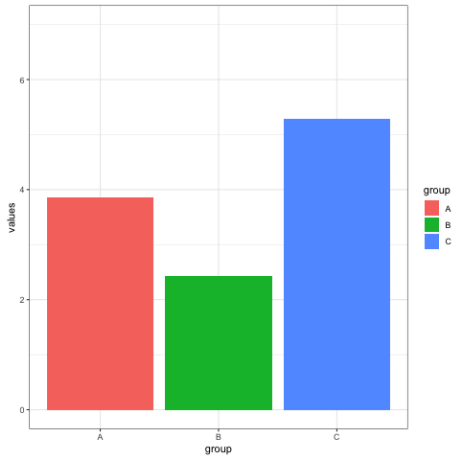



Figure 7: Second frame of the animation

# 6 Conclusion of this introductory analysis

In the first part of this report, we have explained the data structure and the meaning of the individual attributes.

Then, we analyze what can be done with our specific data and what information can be gathered.

Next, I have explained how the data was polled from the API endpoints and showed parts of my python code.

Lastly, based on the knowledge from previous chapters, we have proposed a way to visualize the data in a intelligible and intuitive way.

It would be beneficial to get the most out of interaction, which would entail:

- Selection of players/teams whose data is to be visualized
- Attributes to be visualized
- Assign a user-defined color to each player or automatically add colors that are easily distinguishable from others
- Zoom in on a specific section of data

I will be attempting to implement some of the strategies outlined above in JavaFX in the future part of this semestral work. While I think all of the strategies mentioned above would be somehow useful, I think line chart with the option to add multiple lines will be the most beneficial, while being the easiest to implement.

# 7 Implementation

As stated before, we still had to poll the specific games from the API. This was done using another python script, which simply took the data for each of the specified game IDs and downloaded them as .json files. Below can be seen the important part of the script used to generate them.

```
for x in gameIDs:
    print('processing ' + str(x))
    getMatchDetailsSpecificGame = getMatchDetails + str(x)

    data = rq.get(getMatchDetailsSpecificGame, params=parameters, headers=headers)
    data = data.json()

    metadata = data['gameMetadata']
    frame = data['frames'][-1]

    output_data = metadata
    output_data.update(frame)

    name = str(x) + '.json'
    with open(name, 'w') as outfile:
        json.dump(output_data, outfile)
```

Then, we created a new JavaFX project and specified Maven as the build system. This allowed us to add Gson[7] library, which was used to deserialize the JSON files into equivalent Java objects. We used jsonschema2pojo[8] library to automatically generate the Java files needed for the deserialization. The code used to deserialize the data can be seen below.

```
for (String gameID : gameIDs) {
    // create Gson instance
    Gson gson = new Gson();

    // create a reader
    String path = "data/" + gameID + ".json";
    InputStream inputStream = getClass().getResourceAsStream(path);
    Reader reader = new BufferedReader(new InputStreamReader(inputStream));

    // convert JSON file to GameData class
    GameData gameData = gson.fromJson(reader, GameData.class);

    gamesData.add(gameData);

    // close reader
    reader.close();
}
```

After that, I created a simple graph showing the total gold acquired for each team per game as a way to check the data. We found two inconsistencies - firstly, the game between Fnatic and Astralis was played on 18. February seems to have only data before the start of the game - the total gold for each team is 2500 and upon inspecting the file manually, every other stat is either 0 or starting value as well. This can be seen in Figure 8 below as the massive slump in the middle of the graph.

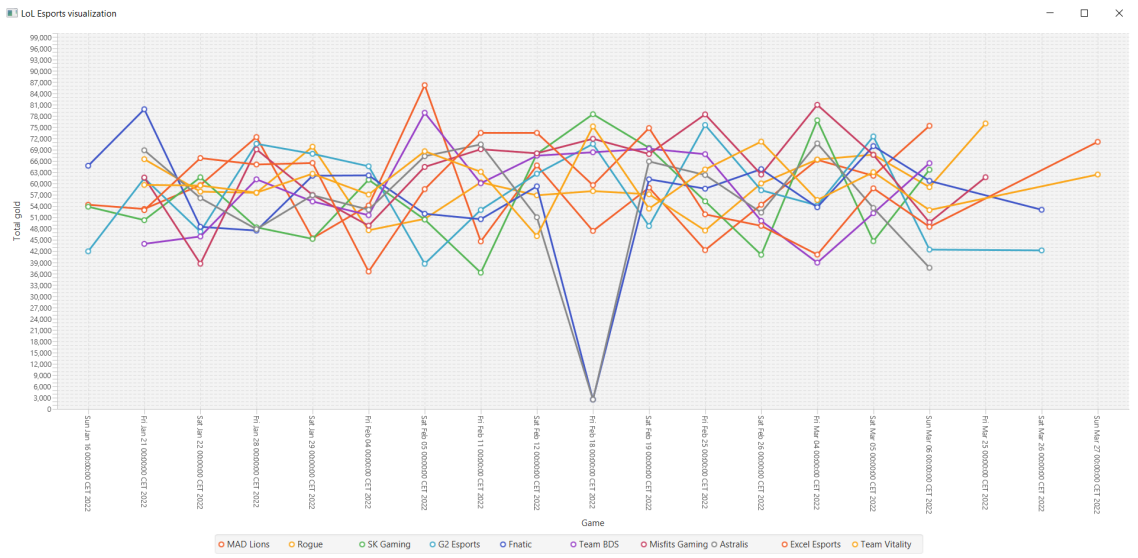



Figure 8: Initial graph example

This is sadly a problem on the API side and apart from manually filling the data, there is nothing we can do. The second problem was that for some reason, we also got some additional data from games from playoffs and some data from before - those are the lines that start a little bit sooner and the lines that end later than others. We have filtered the data so that only games from 21. January onward till March 6 are visible.

## 7.1 Graphs

In order to plot the data, we need to visualize it using some form of graph. For the application to be universal, we have defined a simple interface to work with our graphs - the GraphUtil interface. It defined the following functions:

- initGraph - to initialize the graph axes, axes labels and set graph size
- addData - to add data into the graph
- clearData - clear data from the graph
- updateYAxis - update Y axis when attribute to be visualize has changed
- addTooltips - adds tooltips to bars or nodes
- getGraph - returns current graph

### 7.1.1 Line chart

While being easy to implement and most people being familiar with the concept, after some initial plotting, it was becoming clear the graphs, where there was multiple series of data, were hardly readable. Especially when the lines crossed each other multiple times, it was not feasible to compare the data for average human.

### 7.1.2 Stacked bar graph

The stacked bar graph is still very easy to implement, in fact, the implementation does not really differ from the line chart. Nevertheless, it seems to be substantially more effective when polling multiple series of data. Even though it requires closer inspection to understand the data, it would still be our preferred way to plot the data. However, the user does not need to do so, as explained later in section 7.5.

## 7.2 Filtering by selected teams

User can select any teams he wants to be visible in the graph using the checkboxes in the bottom center part of the application. We chose checkboxes cause they are minimalist and easy to utilize for the user. We can see all the available teams checkboxes implemented in Figure 9 below.


Figure 9: Checkboxes of the available teams

The implementation is following: create multiple JavaFX checkboxes and name them using properties of the TeamInfo class. Then, setting a listener for every one of them, which alters the selectedTeams map property and updates the graph. This is illustrated in the code snippet below.

```
 // Creating toggles for each team (X axis)
HBox teamsToggle = new HBox();
teamsToggle.setSpacing(10);
teamsToggle.setAlignment(Pos.CENTER);
for (TeamInfo teamInfo : teamInfos) {
    CheckBox cb = new CheckBox(teamInfo.getFullName());
    cb.setSelected(true);
    teamsToggle.getChildren().add(cb);

    // Set listener for changes
    cb.selectedProperty().addListener((ObservableValue<? extends Boolean> ov, Boolean old_val, Boo
        selectedTeams.put(teamInfo.getTeamID(), new_val);
        updateGraph(selectedAttribute, gamesData, selectedTeams);
    });
}
```

## 7.3 Filtering by available attributes

Given there are multiple important aspects to achieve the win in the game, you can select and compare how each team performed in these categories. User can select one attribute which he wants to be visualized. This is done using radio buttons in the bottom center part of the application. The attribute is plotted in correlation to the day the game was played. We also need to set the Y axis lower and upper bounds, so that the data are displayed correctly. The actual radio buttons from the application can be seen in Figure 10 below.


Figure 10: Radio buttons of the available attributes

The implementation is similar to filtering from section 7.2, but this time using RadioButton class. We also need to create a ToggleGroup object as all the RadioButtons need to interact with each other.

```
// Creating radio buttons to select attribute (Y axis)
HBox attributesToggle = new HBox();
attributesToggle.setSpacing(10);
attributesToggle.setAlignment(Pos.CENTER);
ToggleGroup tg = new ToggleGroup();
for (GameAttribute gameAttribute : gameAttributes) {
    RadioButton rb = new RadioButton(gameAttribute.getInfoText());
    rb.setToggleGroup(tg);
    attributesToggle.getChildren().add(rb);
}
// Set first radio as checked
RadioButton first = (RadioButton) attributesToggle.getChildren().get(0);
first.fire();
```

```
// Setting listener for radio buttons changes
tg.selectedToggleProperty().addListener((observable, oldValue, newValue) -> {
    RadioButton rb = (RadioButton) tg.getSelectedToggle();
    if (rb != null) {
        String s = rb.getText();
        this.selectedAttribute = s;
        updateGraph(s, gamesData, selectedTeams);
    }
});
```

## 7.4 Tooltips

Sometimes, in order for the user to not need to checkup which bar or node belongs to which team, we have implemented tooltips that show up on mouse hover over bar or node belonging to the graph. To get an idea of how it works, we were hovering over the green bar in the middle of the screen in Figure 11 below, the dark rectangle is the tooltip.

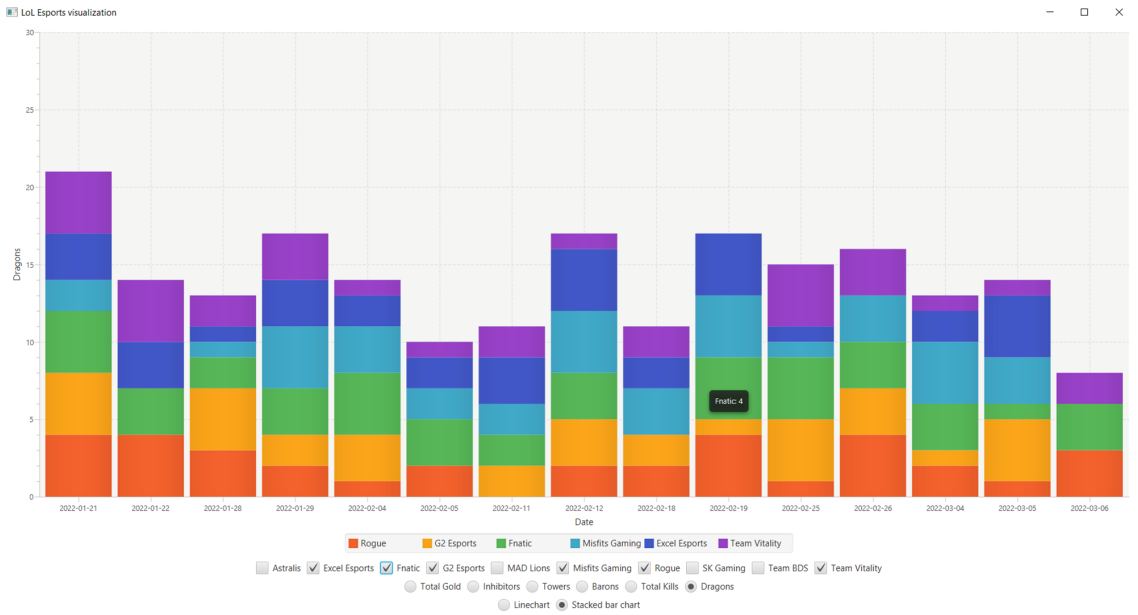



Figure 11: Tooltip - Fnatic has slain 4 dragons in their game played on 19th of February

The function addTooltips in class LineGraphUtil is responsible for this behaviour.

```
public void addTooltips() {
    for (XYChart.Series<String, Number> series : chartData.values()) {
        for (XYChart.Data<String, Number> entry : series.getData()) {
            Tooltip t = new Tooltip(series.getName() + ": " + entry.getYValue().toString());
            Tooltip.install(entry.getNode(), t);
        }
    }
}
```

## 7.5 Selecting graph visualization

In order to select between the linechart visualization and stacked bar graph visualization, the user can use the radio buttons in the bottom center part of the application.

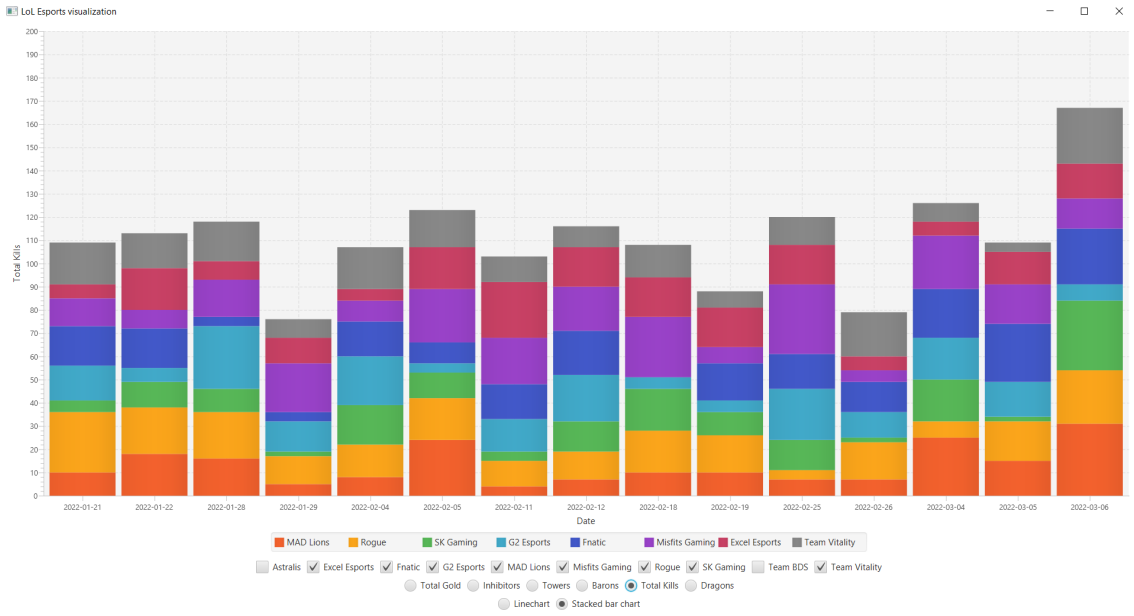
```
// Listener for requested graph type changes
barTypeToggleGroup.selectedToggleProperty().addListener((observable, oldValue, newValue) -> {
    RadioButton rb = (RadioButton) barTypeToggleGroup.getSelectedToggle();
    if (rb != null) {
        String s = rb.getText();
        if (s.equals("Linechart")) {
```

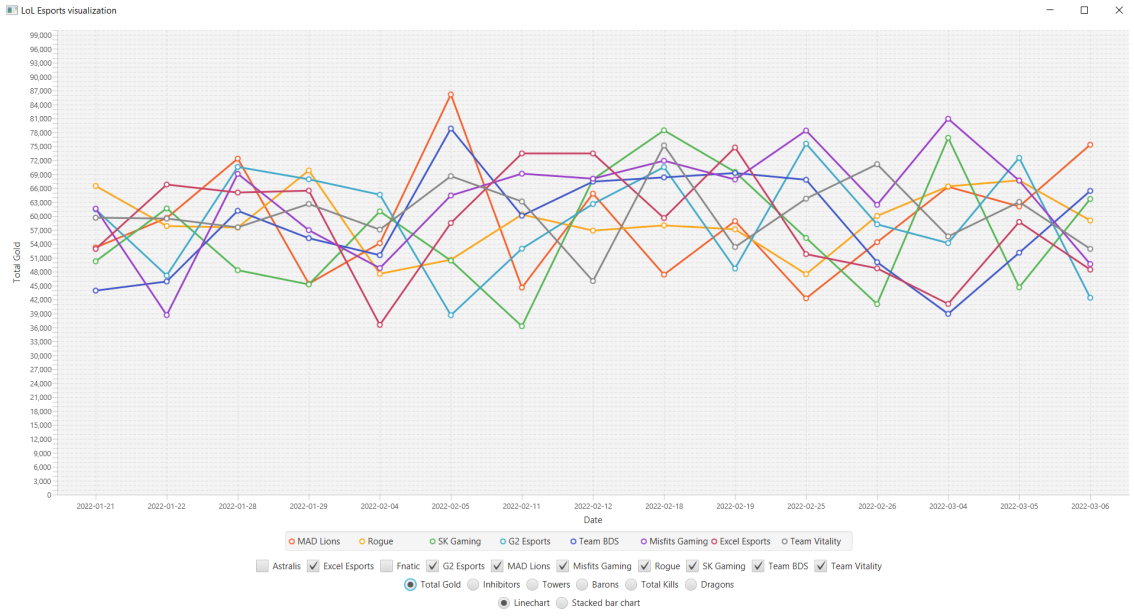
```
            GraphType.INSTANCE.setSelectedGraph(LineGraphUtil.INSTANCE);
            mainLayout.getChildren().set(0, GraphType.INSTANCE.getSelectedGraph().getGraph());
            updateGraph(selectedAttribute, gamesData, selectedTeams);
        } else if (s.equals("Stacked bar chart")) {
            GraphType.INSTANCE.setSelectedGraph(StackedBarGraphUtil.INSTANCE);
            mainLayout.getChildren().set(0, GraphType.INSTANCE.getSelectedGraph().getGraph());
            updateGraph(selectedAttribute, gamesData, selectedTeams);
        }
    }
});
```

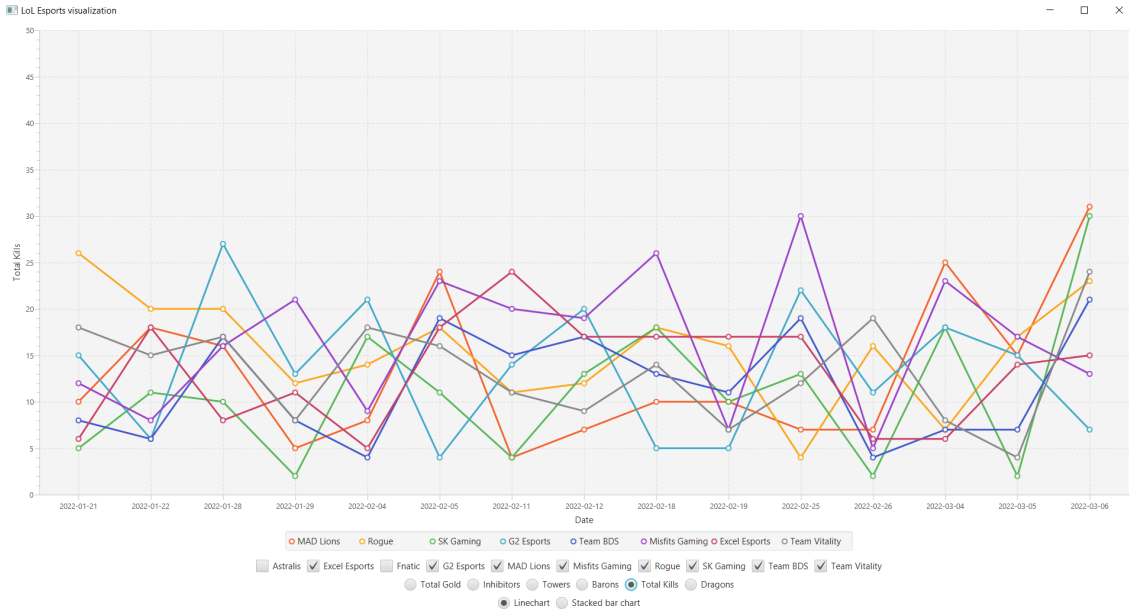
# 8 Finished work

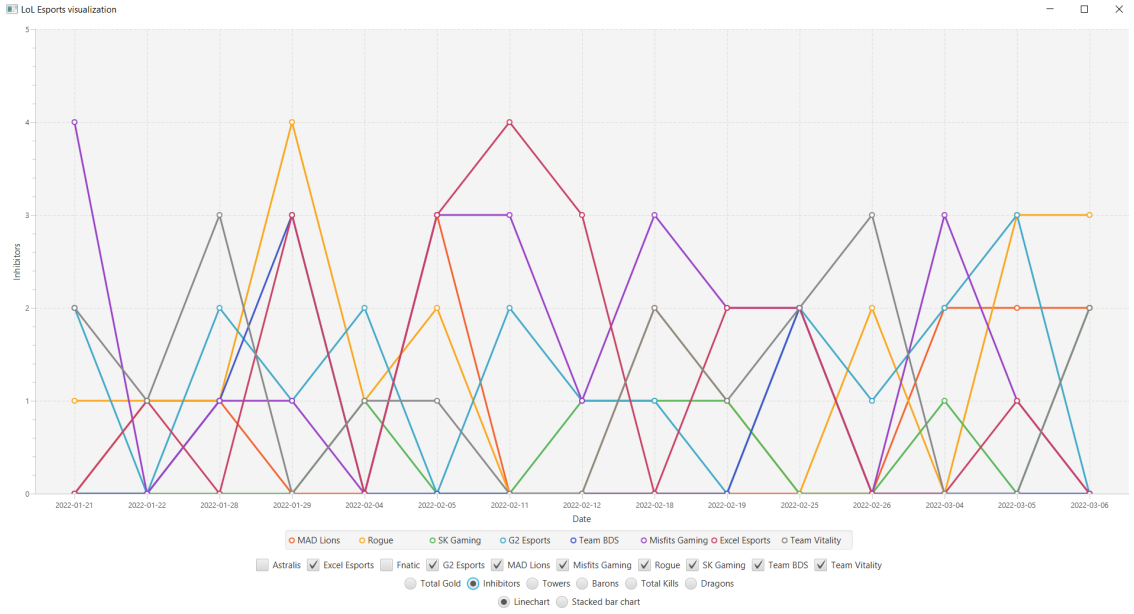
Apart from the files on the Google Drive folder (available at https://drive.google.com/drive/folders/19e81DxOZYXtoyfzHTEicdVR9t8DKeuhn?usp=sharing), the implementation is available at https://github.com/zdenduk/lolesports-visualization and https://github.com/zdenduk/lolesports-data.

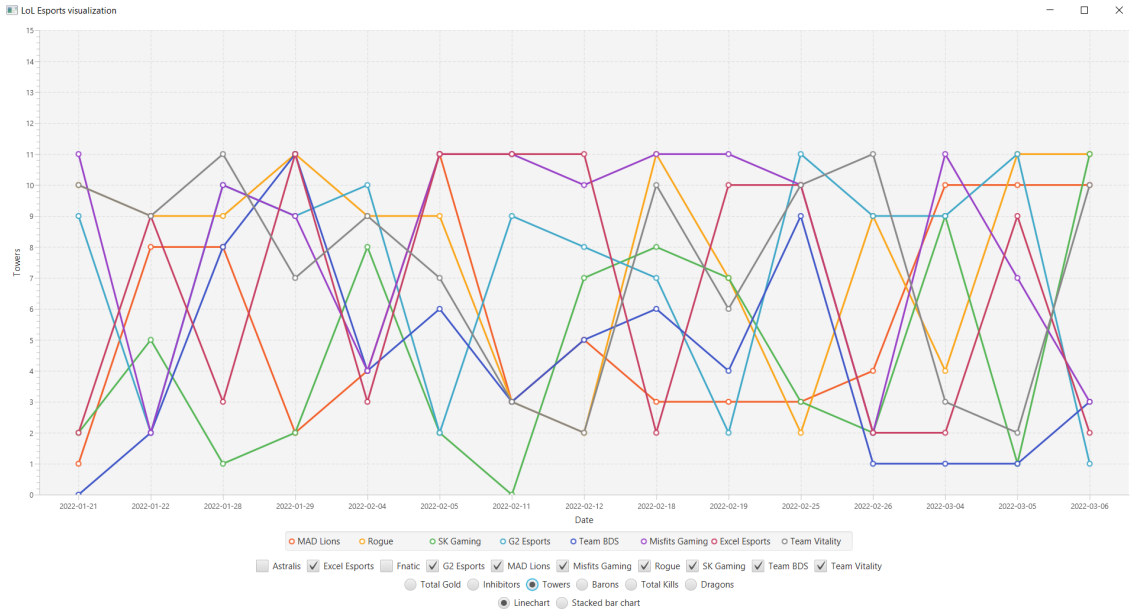
The screenshots of the working application follow:

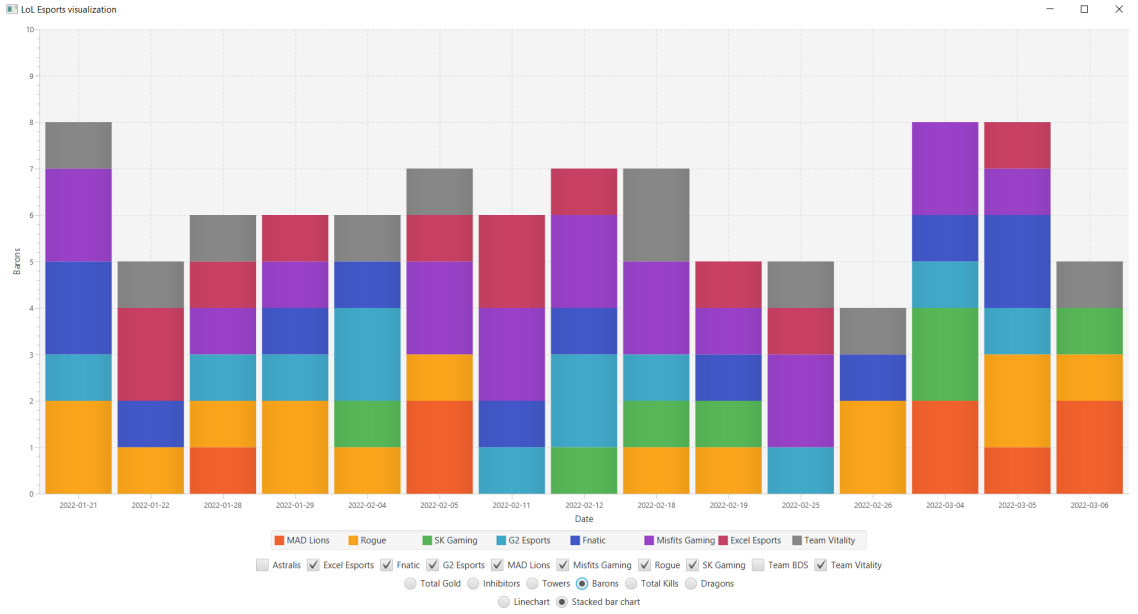


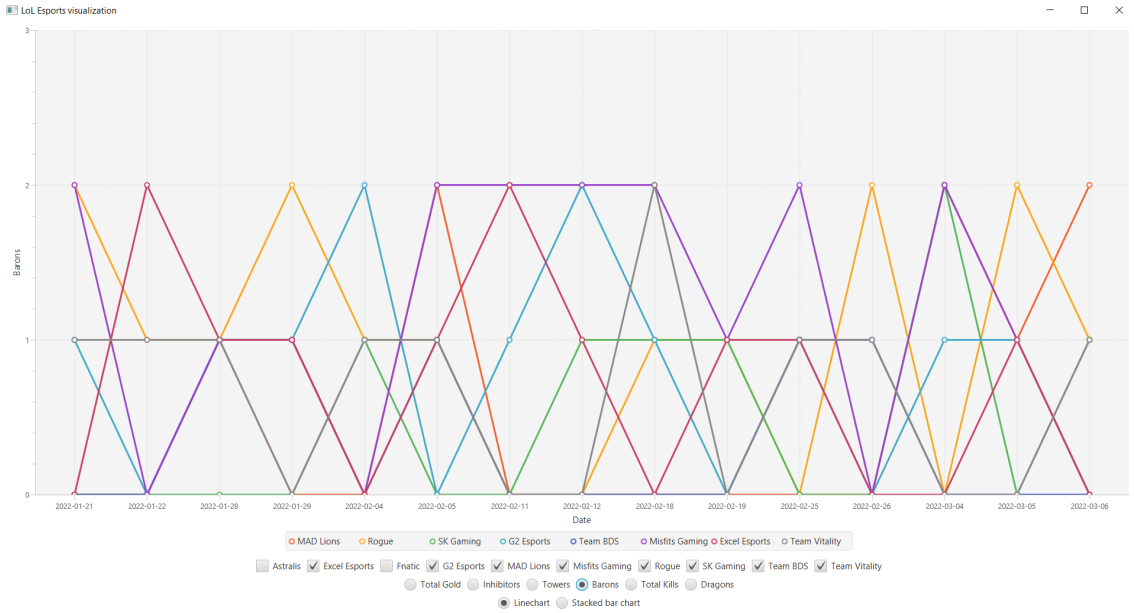



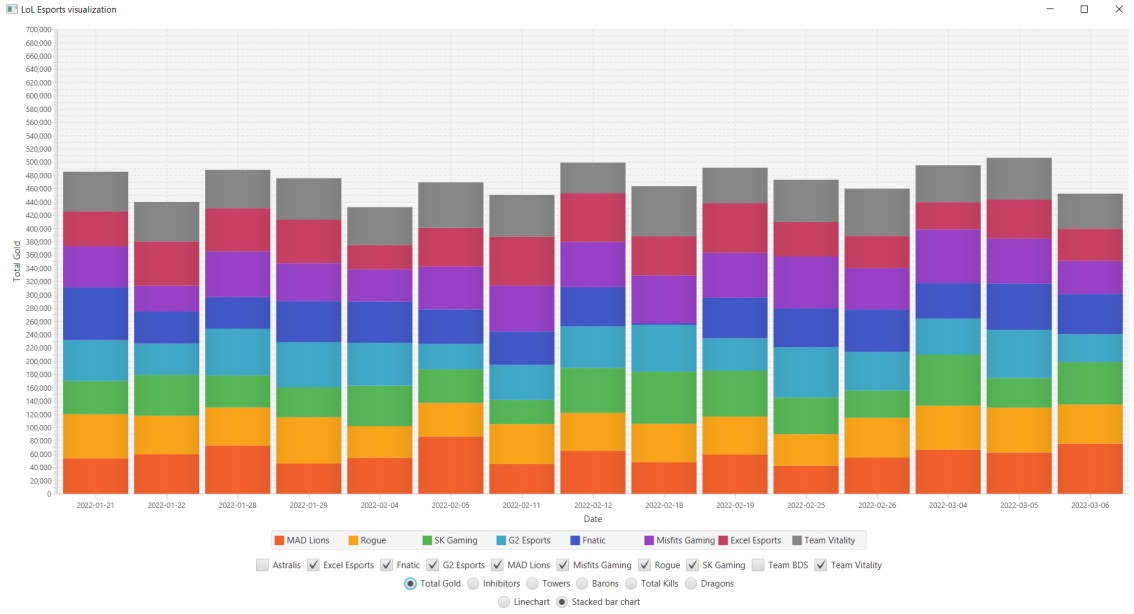

LoL Esports visualization
Total Kills
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Team BDS
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Fnatic
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart

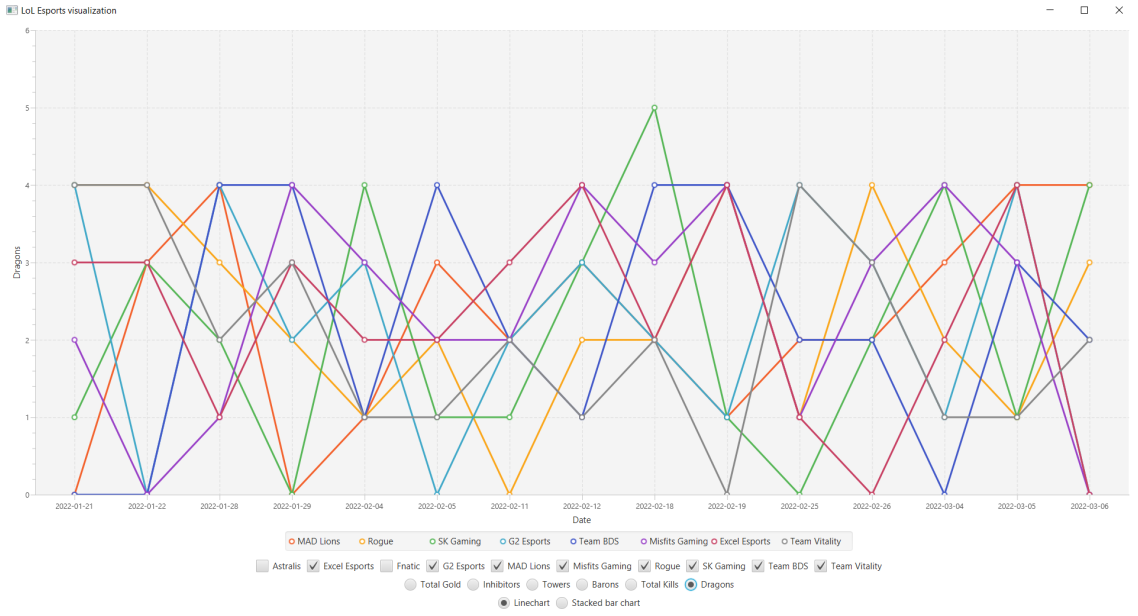

LoL Esports visualization
Inhibitors
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Team BDS
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Fnatic
Total Gold
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart

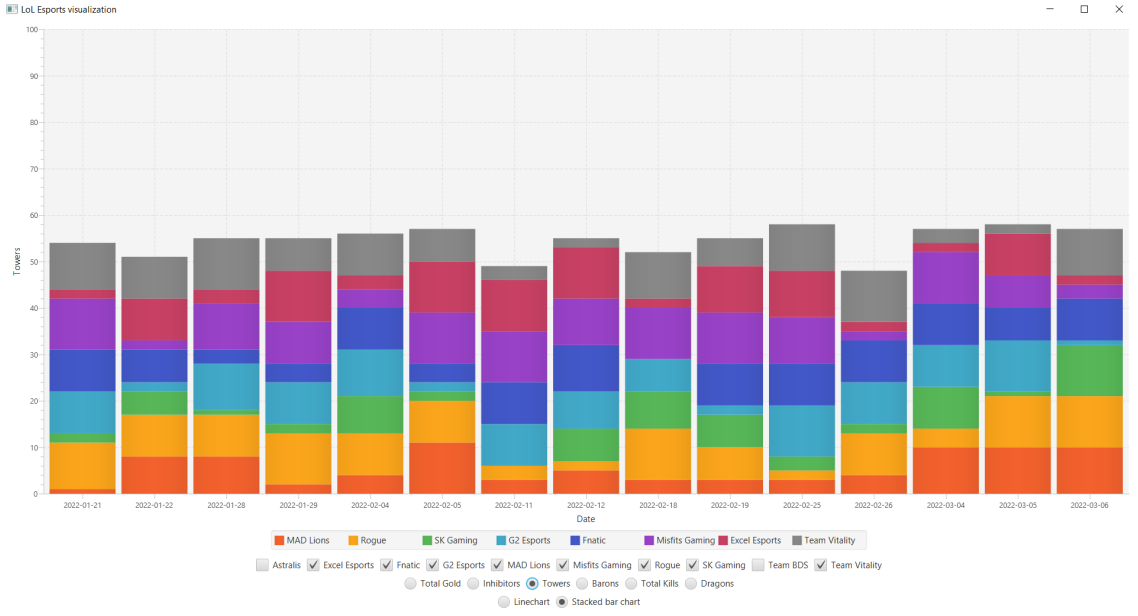

LoL Esports visualization
Towers
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Team BDS
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Fnatic
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart

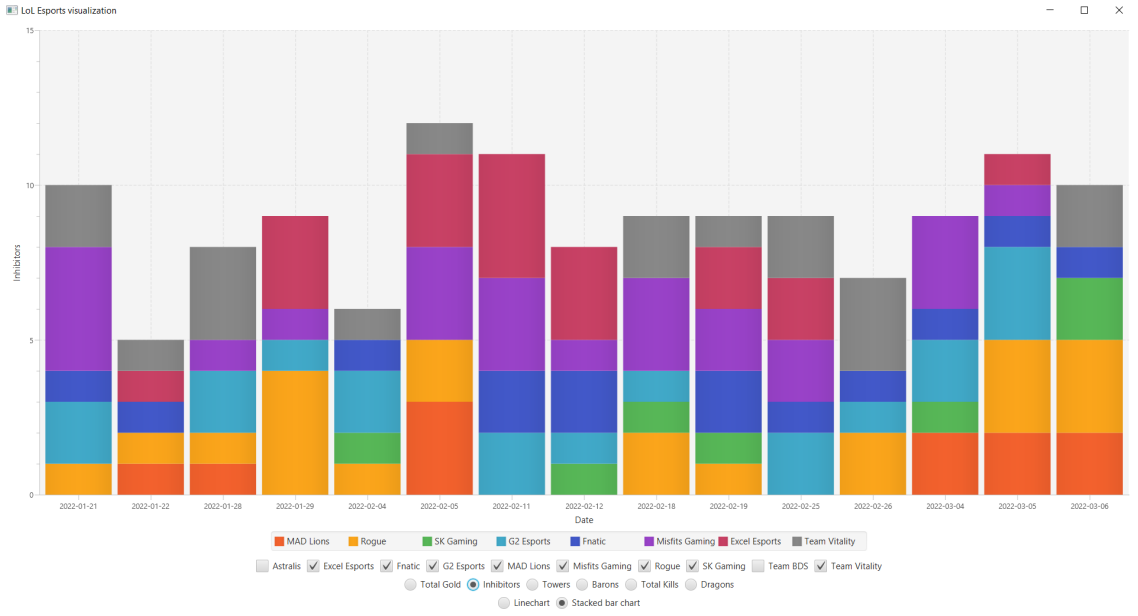

LoL Esports visualization
Barons
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Fnatic
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Team BDS
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart

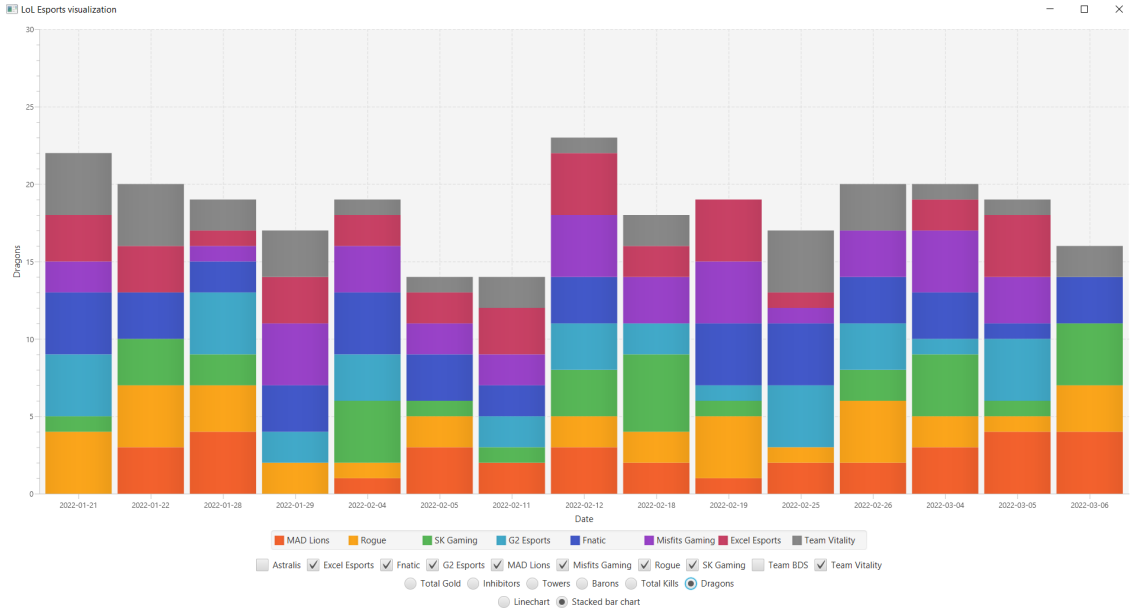

LoL Esports visualization
Barons
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Team BDS
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Fnatic
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart


LoL Esports visualization
Total Gold
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Fnatic
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Team BDS
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart


LoL Esports visualization
Dragons
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Team BDS
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Fnatic
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart


LoL Esports visualization
Towers
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Fnatic
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart


LoL Esports visualization
Inhibitors
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Fnatic
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Team BDS
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart


LoL Esports visualization
Dragons
Date
MAD Lions
Rogue
SK Gaming
G2 Esports
Fnatic
Misfits Gaming
Excel Esports
Team Vitality
Astralis
Team BDS
Total Gold
Inhibitors
Towers
Barons
Total Kills
Dragons
Linechart
Stacked bar chart

# 9 Summary

The data was polled from the API using python, however 1 file has incorrect information. All the files were then loaded to Java without any complications.

We have managed to meet the requirements for teams and attribute filtering. However, players visualization was not implemented. Instead of zooming on specific section of data, tooltips were added.

Some visualizations provide additional information, however, we feel like the visualizations could provide more information if we were able to calculate lenght of games.

# References


[1] Riot Games. Lol esports.

[2] Aastryx. Weak side, Mar 2020.

[3] Wikipedia contributors. Line chart — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Line_chart&oldid=1054026618, 2021. [Online; accessed 6-April-2022].

[4] Grigory Starinkin. Historical github contributions, Dec 2017.

[5] Plotly. Streamgraph with themeriver layout [1] [2]: Line chart made by empet.

[6] Michael Wohlfahrt and Jürgen Platzer.

[7] Google. Google/gson: A java serialization/deserialization library to convert java objects into json and back.

[8] Joe Littlejohn. Jsonschema2pojo.