September 11th, 2024, Portfolio Document

Abstract

TLDR; This is a full-stack web application for a game, Team Fight Tactics, to showcase core skills in Frontend, Backend, DevOps, Data Science, research (Data Science), and design. This weeklong project serves as a supplement and proxy of the wide breadth of skills accumulated over the past 8 years across multiple disciplines and industries.

Calculator linked here: https://tftpage-frontend-43814569847c.herokuapp.com/

Introduction

This project is a Full-stack web application for Team Fight Tactics unit calculations

This application allows users to calculate the expected (average) number of rolls and other values for units in the game Team Fight Tactics (TFT). Features include running simulations,

Game linked here: https://teamfighttactics.leagueoflegends.com/en-us/

Calculator linked here: https://tftpage-frontend-43814569847c.herokuapp.com/

This write up is written after version 2 was made generally available.

Game Description

TFT is an online auto battler game where users strategically "roll" for units and formulate a team to be the last player standing. A roll is a refresh of the game's shop in which the user can purchase units.

The units to be purchased follow a unique set of rules based on the cost, and the pool from which the unit can be purchased from. The multiplayer and common pool that users interact with lead to a probabilistic problem that this calculator aims to simplify.

Goal

This calculator looks only at the roll functionality and simplifies the different state transitions to help assist users to formulate a strategy or simply visualize the mathematical interactions of the game's unit selection design.

From a tech standpoint, this calculator was built to show a full-stack view of designing and building a web application and the effort so far.

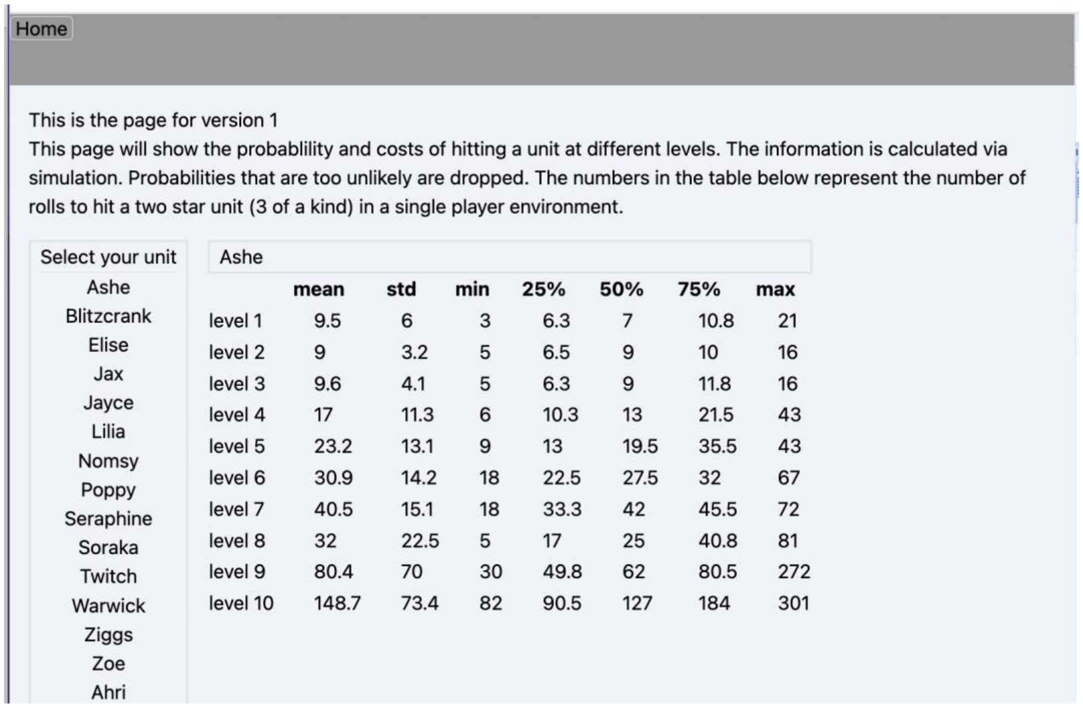
Tech Stack

- Frontend:
  - Major: React, Zustand, Tailwind CSS, Typescript, Vite
  - Minor: React Router, Serve
- Backend:
  - Major: Flask, SQLAlchemy, SQLite, RESTful APIs, Python (3.11)
  - Minor: Flask_CORS,
- DevOps:
  - Major: Heroku, GitHub, Git
  - Minor: Conda Environments, NPM, Node.js

- Design:
    - Major: UI/UX design
    - Minor: Hand drawn sketches, Microsoft Word (documentation and summaries)
- Research:
    - Major: Data Science, Competitive Analysis, Mathematical comparisons and validations

Screenshots

Version 1 (V1)

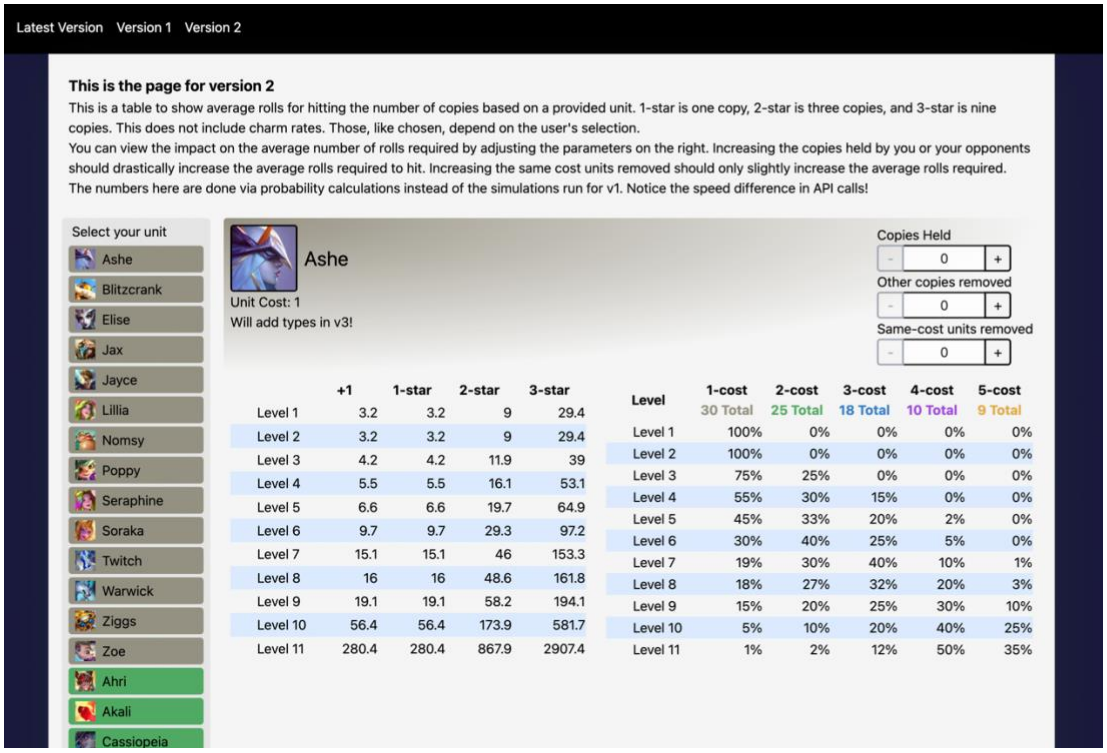


This is the page for version 1
This page will show the probablility and costs of hitting a unit at different levels. The information is calculated via simulation. Probabilities that are too unlikely are dropped. The numbers in the table below represent the number of rolls to hit a two star unit (3 of a kind) in a single player environment.

Ashe

	mean	std	min	25%	50%	75%	max
level 1	9.5	6	3	6.3	7	10.8	21
level 2	9	3.2	5	6.5	9	10	16
level 3	9.6	4.1	5	6.3	9	11.8	16
level 4	17	11.3	6	10.3	13	21.5	43
level 5	23.2	13.1	9	13	19.5	35.5	43
level 6	30.9	14.2	18	22.5	27.5	32	67
level 7	40.5	15.1	18	33.3	42	45.5	72
level 8	32	22.5	5	17	25	40.8	81
level 9	80.4	70	30	49.8	62	80.5	272
level 10	148.7	73.4	82	90.5	127	184	301

Version 2 (V2)



This is the page for version 2

This is a table to show average rolls for hitting the number of copies based on a provided unit. 1-star is one copy, 2-star is three copies, and 3-star is nine copies. This does not include charm rates. Those, like chosen, depend on the user's selection.

You can view the impact on the average number of rolls required by adjusting the parameters on the right. Increasing the copies held by you or your opponents should drastically increase the average rolls required to hit. Increasing the same cost units removed should only slightly increase the average rolls required.

The numbers here are done via probability calculations instead of the simulations run for v1. Notice the speed difference in API calls!

Ashe

Unit Cost: 1

Will add types in v3!

Copies Held: 0

Other copies removed: 0

Same-cost units removed: 0

	+1	1-star	2-star	3-star
Level 1	3.2	3.2	9	29.4
Level 2	3.2	3.2	9	29.4
Level 3	4.2	4.2	11.9	39
Level 4	5.5	5.5	16.1	53.1
Level 5	6.6	6.6	19.7	64.9
Level 6	9.7	9.7	29.3	97.2
Level 7	15.1	15.1	46	153.3
Level 8	16	16	48.6	161.8
Level 9	19.1	19.1	58.2	194.1
Level 10	56.4	56.4	173.9	581.7
Level 11	280.4	280.4	867.9	2907.4

Level	1-cost	2-cost	3-cost	4-cost	5-cost
	30 Total	25 Total	18 Total	10 Total	9 Total
Level 1	100%	0%	0%	0%	0%
Level 2	100%	0%	0%	0%	0%
Level 3	75%	25%	0%	0%	0%
Level 4	55%	30%	15%	0%	0%
Level 5	45%	33%	20%	2%	0%
Level 6	30%	40%	25%	5%	0%
Level 7	19%	30%	40%	10%	1%
Level 8	18%	27%	32%	20%	3%
Level 9	15%	20%	25%	30%	10%
Level 10	5%	10%	20%	40%	25%
Level 11	1%	2%	12%	50%	35%

Detailed Achievements

Frontend:

- Built a responsive and interactive webapp using the **React** framework, **Zustand** for state management, **Tailwind CSS** for fast class-based styling, **Typescript** for industry level safe typing, and **Vite** for module building, bundling, and hot module replacement for faster development.
- The webapp is designed as a single page app with lots of interactivity with **React-Router** and served with client-side routing in production with **Serve**.
- Specifics:
  - Scroll component is bug free with a percentage-fixed height.
  - Table component is designed and generalizable with cell level customizability. Also issue free with different state changes.
  - Debouncer implemented with Incrementer, high frequency interaction component, to reduce API calls on intermediate values. Reducing worst-case loads by 10x based on minimum Mac key-repeats when using the keyboard.
  - Dynamic parallel importing icons using Javascript Blobs.
  - Built a preloading component to consistently load dynamic colors with Tailwind. This optimization keeps code readable and loads custom styles on page load.

    - package.json management so the page can be rendered on Heroku or any other platform. This was done by changing the scripts.

Backend:

- Developed a **RESTful API** in **Flask** running **Python** 3.11 with an **SQLite** database and accessed with **SQLAlchemy** and **pandas**. Hosted on Heroku with a low maximum monthly cost.
- Set up the webapp to query the backend using **Flask_CORS** so the backend can be set up on a different host to reflect structure in an industry environment like Amazon (AWS).
- Implement **HTTP cache control** to reduce API calls on deterministic requests.
- Add rate limiter for non-deterministic requests
- Specifics:
    - Translated **Jupyter Notebook** written code used in researching and simulating the product in functions to return calculations on API requests.
    - Managed data in an SQLite database to be more industry similar instead of csv files which would have been much easier to adjust by hand. This database was sufficient for my use case and no further complexity was required for an app like this.
    - Designed database to handle the different game versions and patches.
    - Designed the APIs to prevent SQL injections by only handling bound parameters.

DevOps:

- Used **Git** for version control, uploaded codebase to **GitHub** and then deployed code to **Heroku** for hosting.
- Backend Python packages were managed with **Conda** environments (**venv**). Frontend packages were managed with **NPM**.

Design:

- UI/UX design drafted using hand drawn sketches. The behavior and expectations of the product were outlined and prepared beforehand.
    - There are multiple indicators to show unit cost and grouping by stylizing the colors of the page sub-header.
    - Stylized the table to be easily readable
    - Added in probability table similar to Riot’s original implementation to show cost, level, total amount, and probability in a single table.
- Documentation and summaries written **Microsoft Word** and then exported as PDF. This file is an example as such. Other documentations are kept as reference to track the

changes between each version, the decisions made, and the research on competitive analysis.

Research:

- **Competitive analysis** consisted of researching other similar products, comparing math computes amounts, ease of use, accuracy, compute speed, and UX. The detailed information and methodologies can be found in documentations and notebooks upon request.
    - I built a simulation accurate to the game mechanics and verified against others who also clearly stated the specific game mechanics. The simulation results were then used as baseline to compare expected valued and it showed several inaccuracies in others.
    - Some used **Markov chains** with calculations based on the power of their transition matrices. This scales O(k * n^3) where k is the power and n is from the size of the n x n transition matrix. Their approach used the expected number of units found based on iterations or rolls of the transition matrix but would take a long time because some rolls for 3-stars were up in the thousands of rolls.
        - I improved upon this by using an absorbing matrix and calculating the fundamental matrix, bringing down the calculations to a few O(n^3) calculations. Subsequently being able to show the entire range of levels within milliseconds even on the basic Heroku node. This also meant that I could calculate 10000x10000 sized matrices in Python before they could calculate a 64x64 matrix in C++.
        - My process also yielded a more accurate answer to the question of "how many rolls are required to acquire these number of units" instead of their question of "If I rolled X amount times, what is the expected number of units acquired".
        - Other benefits are the further flexibilities of calculations should I either expand the simulation or make it more complex such as calculating the probability of acquiring a team of units instead.
    - Another two failed by rounding too early resulting in expected rolls more than 10% higher than the simulation baseline. One was identified because they shared the code, another was inferred because the code was not shared but the resulting calculations were identical.
    - I identified probability theory flaws that they used with geometric distributions and hypergeometric distributions.
        - Geometric distributions should not be used unless only going for a single unit. And it does not accurately apply for repeated rolls for any

        subsequent rolls. The difference in error is small of a few percentages, but another issue introduced is that further calculations of 3-stars become dependent on the previous results which further slowdown their calculations.
        - Hypergeometric distributions were also used inaccurately since each slot in the shop is independent of one another and draws from the same pool. This means that the shop slots are binomial distribution based. Finally, since there is a weight of each tier of unit, the correct distribution to use is a noncentral multivariate hypergeometric distribution with partial replacement. Noncentral meaning weighted or sampling with bias, multivariate because there are 5 potential tiers to draw from, and partial because non-selected units are replaced while selected ones are not.
        - The method I used is putting the probability mass function of a binomial distribution as transition states to calculate the sums of average state changes from a fundamental matrix derived from the absorbing Markov chain model.
- To ensure API viabilities and know which APIs were more suitable to be cached versus rate limited, I timed the simulations and calculations at each iteration. With the research and improvements, I was able to handle the following: "*cost of unit x level x num of unit held x num of other taken x num of unit needed x matrix calculations"* in a scalable manner.
- Calculations, data extraction, and data loading were done using **Data Science** tools such as **pandas**, **NumPy**, **sqlite3**, and **mathematics** and **statistics**. They were done in a **Jupyter Notebook** but with structure and repeatability for easy conversion to **Python** functions to be placed in the backend.
- Edge cases and validations were done so bad user inputs were handled in the webapp.
- Screenshots and analysis of competitors were handled in a **literature review** as done for writing a research paper.

Conclusion

This project rounded up a glimpse of my skills thus far, with a focus in showing the core skills of each discipline that I accumulated working in different industries.

In future versions, I plan to add the follow core technologies and their next steps:

- User authentication
    - This would allow for storing custom data per user
    - Allow for a paid tier for more computationally accurate or advanced features

    - Implement JWT tokens for security, possible SSO features
- More mathematical computation solutions
    - This will enable more TFT calculation features such as selecting multiple units to calculate the probabilities and compare instead of single unit selections used in V2.
    - Enable faster comparison of different team compositions and therefore strategies.
- GDPR compliance
    - This in combination with user authentication will allow for saving the user's last state when using the app.
    - Makes the app more like industry standards
- Accessibility testing
    - Makes the app more like industry standards
- Tests
    - Enables peace of mind for CI/CD if the app ever needs to scale
    - Is more like industry standards
- Machine Learning
    - A mathematical computation solution that can make team composition recognition faster with computer vision. This also enables the idea of integrating with game plugins.
    - Can also calculate win/loss probabilities