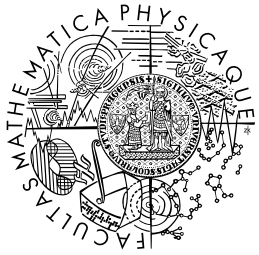

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Hoang Anh Tuan

Third-party Matchmaking Discord Bot for Valorant

Department of Software Engineering

Supervisor of the bachelor thesis: Peška Ladislav, Mgr., Ph.D.
Study programme: Informatics
Study branch: Programming and Software Systems

Prague 2025

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor, Mgr. Ladislav Peška, Ph.D., for his endless patience and help in every step of the thesis, my family for their unconditional love and support, and my friends for always being there for me.

Title: Third-party Matchmaking Discord Bot for Valorant

Author: Hoang Anh Tuan

Department: Department of Software Engineering

Supervisor: Peška Ladislav, Mgr., Ph.D., Department of Software Engineering

Abstract: One of the common tasks in online multiplayer games is matchmaking, i.e., assembling a game with concrete players and game settings. Players, in general, want to be matched with similarly skilled players while spending little time waiting for a match. Players may also have different preferences regarding maps, roles, or other settings. Often, matchmaking systems disregard such preferences.

In this thesis, we examine in-game matchmaking and existing third-party options for Valorant and other games. Based on our observations, we propose a matchmaking Discord Bot Valorant focused on improving the user experience. The Bot works on different Discord servers separately and provides slash commands for users to interact with it. We also improve matchmaking functionalities, such as player selection, player rating, and map selection.

Keywords: Discord Valorant matchmaking

Contents

Introduction

Valorant is an online first-person shooter (FPS) game. Players are matched to play together as a team of five to defeat the other squad. Valorant selects players for a match based on their matchmaking rating (MMR), which is evaluated by their performance. In a Valorant competitive mode, players compete for a rank rating (RR) that is used to compare with other players. Since Valorant does not provide an exact metric for calculating RR after a match, players are often frustrated by how little RR they gain for a win and how much they lose after a defeat. Valorant also does not reveal players' MMR, which impacts their RR.

In a high-stakes situation, as in the competitive mode, players want to play with the best possible teammates and compete against fair-play opponents. That is sometimes not the case because some players are not on their best behavior because of toxicity[Sara "Riot necrotix" Dadafshar(2022)], AFK[1] behavior [Sara "Riot necrotix" Dadafshar(2021)], or cheating[Chamberlain(2020)].

FPS games are often played on different maps. They are different from each other, which makes the game feel diverse. Because of that, players usually like one map better than another one. They have favorite maps and maps they want to play only on rare occasions. Many games provide an automatic map selection feature that does not consider player map preferences. This issue causes an unstable game experience since it is based on luck if players get primarily maps they like to play, the ones they dislike, or a healthy mixture of both.

For the reasons mentioned above, players seek third-party matchmaking options that are stricter with player-based regulations and have more transparent algorithms.

Motivation

My motivation for this bachelor thesis comes from regularly watching a live stream of a streamer, Tarik "tarik" Celik. He has been actively streaming Valorant since the game was in beta version in April 2020. Due to Celik's stream popularity, viewers would often queue snipe[2] his matches, forcing him to play with the same players repeatedly. Opponents who have found out they are playing against him could also stream snipe[3] him. He would occasionally accuse his opponents of doing this because of their additional information, which would not be possible to get in any other way.

Fed up with the situation, he created a private server, Pro City, on a social media platform, Discord[4]. The server featured professional Valorant players and players he trusted to play up to the standards of the professional players. The server uses a third-party matchmaking application to replace the official in-game Valorant matchmaking.

[1] Abbreviation for Away From the Keyboard; absence of a person during an activity, e.g., not moving and not reacting in a game for an extended time

[2] Stream viewers trying to forcefully get into the match with the streamer using the live broadcast

[3] Using stream for malicious reasons, e.g., revealing player positions that are not obtainable through the game

[4] `https://discord.com/`

Goal

Our goal for the thesis is to create a matchmaking Discord Bot that is more transparent with its inner algorithms. The Bot should be easy to use and help users in each matchmaking step. Player selection should be as fair as possible while limiting the time needed for the players to get into a match. The rating system should be easy to understand to eliminate player confusion with rating updates after matches. Map selection should take into account players' map preferences.

In this thesis, we introduce Valorant and its matchmaking system. Then, we look into other games and explore third-party matchmaking options. After that, we investigate and implement matchmaking algorithms focused on user experience.

1. Related work

In this chapter, we explore other games and their official matchmaking processes. In doing so, we better understand matchmaking to optimize it for our thesis. We also investigate possible social gaming platform alternatives for Discord.

1.1 Counter-Strike

Counter-Strike[1] is an online FPS game franchise. Counter-Strike 2 (CS2) is the latest installment of the series. It is free to play and was released on 27 September 2023, the same day its predecessor, Counter-Strike: Global Offensive (CS:GO), was discontinued.

CS2 currently has nine maps for its competitive modes. These maps rotate through an active-duty map pool, which is a list of maps played in competitive modes.

1.1.1 Competitive modes

In the official competitive matchmaking, players are compared using a value CS2 coined as an ELO rating. CS2 has not provided the exact variant of the rating system, but CS:GO used a rating system based on the Glicko rating system [Glickman(1995)].

CS2 offers two types of competitive matchmaking: Premier and Competitive. Premier mode is available only for players with a Prime status, which can be obtained only through purchase, while the Competitive mode is free to play.

Premier mode

Premier mode splits players into two teams, and then players enter a veto system. In the veto system, the teams decide on map selection factors, starting with a higher-rated team.

Phase	Description	Seconds spent
1	Higher rated team chooses the order of team bans	12
2	First team bans 3 maps	25
3	Second team bans 2 maps	25
4	First team selects a map between the last two maps	25
5	Second team selects their starting side	12
6	Preparation phase before a match	11

Table 1.1: List of phases of the veto system and time assigned for each phase

As Table 1.1 shows, players spend 110 seconds in a veto system. If we include queue time and match loading, players spend at least two minutes in the Premier mode matchmaking process.

[1] `https://www.counter-strike.net/`

Competitive mode

In the Competitive mode, players are guaranteed to play on one of the maps they select before queuing up[2]. Then, after completing ten games on a given map, they receive an ELO rating for each map separately. This player rating method is more accurate in predicting player skill on a map than a player rating for every map, like in Premier mode. Players can avoid maps they do not want to play at the cost of increased queue time[3] based on popularity and the number of selected maps.

1.2 League of Legends

League of Legends[4] (LoL) is a multiplayer online battle arena (MOBA) game developed by Riot Games[5]. Although LoL is highly cooperative, players cannot communicate with their teammates using a team voice.

LoL does not provide concrete algorithms for matchmaking and player rating systems. However, the matchmaking goal is to create two teams with similar win chances for a match [Gortok(2018)] using players' MMR. The player rating system uses tiers and divisions separate from the MMR [Riot Phroxzon(2021)].

A team in LoL consists of five players in different roles - Top lane, Mid lane, Jungle, Bot lane, and Support. Before queuing up, players choose a primary and a secondary role they want to play the most. However, it is still possible that the matchmaking system assigns roles to players they did not select to ensure faster queue times. Players are then guaranteed that the system will assign them their selected role in the next match.

1.3 Social platforms for gaming

Players require an online place to communicate and socialize with other players. Discord is one of the biggest social gaming platforms to date, and it was first released in 2015. This section looks into the platforms used before Discord and the current alternatives.

1.3.1 Ventrilo

Ventrilo[6] is a voice-over IP (VoIP) software with limited text-based communication capabilities. It was first released in 2002 and was designed for gamers. Its main strength is low CPU requirements so players can communicate in Ventrilo with an unnoticeable impact on game performance. Ventrilo server has a tree-like structure, where categories can be inner or leaf nodes, and voice channels can be only leaf nodes. Categories divide servers mainly for access rights. Users cannot join voice channels in a category or subcategories without the correct permissions. Users can join a server by simply entering the IP address and a password

[2]Players join a queue to be chosen for a match
[3]Time spent in the queue waiting for a match
[4]`https://www.leagueoflegends.com/`
[5]`https://www.riotgames.com/en`
[6]`https://www.ventrilo4.com/`

the server owner set up beforehand. The vital feature is that a user with the IP address and the password can join the server and connect or disconnect to voice channels without the help of other users. This quality-of-life feature is present in most social gaming platforms with voice call functionality. Ventrilo's downsides are limited chat capability and a limited free version. Users can create a free server that only eight users can join at the same time.

1.3.2 TeamSpeak

TeamSpeak[7] works similarly to Ventrilo. However, the free server can hold up to 32 users. The difference with Ventrilo is that the user must use their hardware to host a server.

1.3.3 Guilded

None of the above-mentioned platforms have custom bot support through their API. Most of the alternatives that support bots are mainly used in workspaces like Slack[8] or Telegram[9]. One of the gaming alternatives with this feature is Guilded[10]. Guilded is very similar to Discord. While Guilded has an estimated 2 million active monthly users [Dickinson(2023)], Discord announced that they have over 200 million active monthly users [Takahashi(2024)].

[7] https://www.teamspeak.com/en/
[8] https://slack.com/
[9] https://telegram.org/
[10] https://www.guilded.gg/

2. Background analysis

In this chapter, we inspect Valorant, examining each part of the matchmaking. Then, we look into third-party matchmaking options for games. We research chess rating systems, which are popular and easily applicable due to their limited number of input parameters. Finally, we summarize the issues and takeaways to improve our thesis.

2.1 Valorant

Valorant is a round-based shooter game in which players form a team of five to fight against another squad of five. There are many different game modes with unique tasks, but this thesis will focus on the main gameplay used in Valorant's competitive game mode.

2.1.1 Game introduction

In this section, we introduce Valorant and explain its gameplay. Then, we break down its matchmaking into parts and examine them separately.

Main gameplay

In the main gameplay, the Attackers team's objective is to eliminate the enemy team, the Defenders. Their other option to win is planting a unique device, spike, on planting sites. The spike's countdown is triggered when the Attackers plant the spike. It explodes after the countdown, and Valorant's video cinematic implies that the spike was designed to steal valuable resources from the Defenders [VALORANT(2021)].

Meanwhile, the Defenders aim to defend planting sites from the Attackers by eliminating them or defusing the spike before the explosion.

Whoever fulfills their objective first wins the round. Teams switch sides after twelve rounds, and players win when they reach at least thirteen rounds with a two-round lead. If the score is 12-12 (i.e., the game reaches 24 rounds without any team having a two-round lead), then teams switch sides every round until one team earns a two-round lead or the players agree to draw.

Competitive mode

For the best gaming experience, Valorant uses an MMR system that ensures a balanced game with minimal player-skill differences. The system stores two ratings for each player: a hidden MMR that specifies players' skills based on their previous performances and a visible rating for leaderboard ranking.

MMR is an integer that the matchmaking system uses to compare players [Dev(2021)]. In theory, players with higher MMR are better than players with lower MMR.

Visible rating is divided into rank, tier, and RR. Players are divided into ranks based on their progress in the competitive mode. Most ranks are split into three tiers, and players in most divisions have RR between 0 and 100. Players

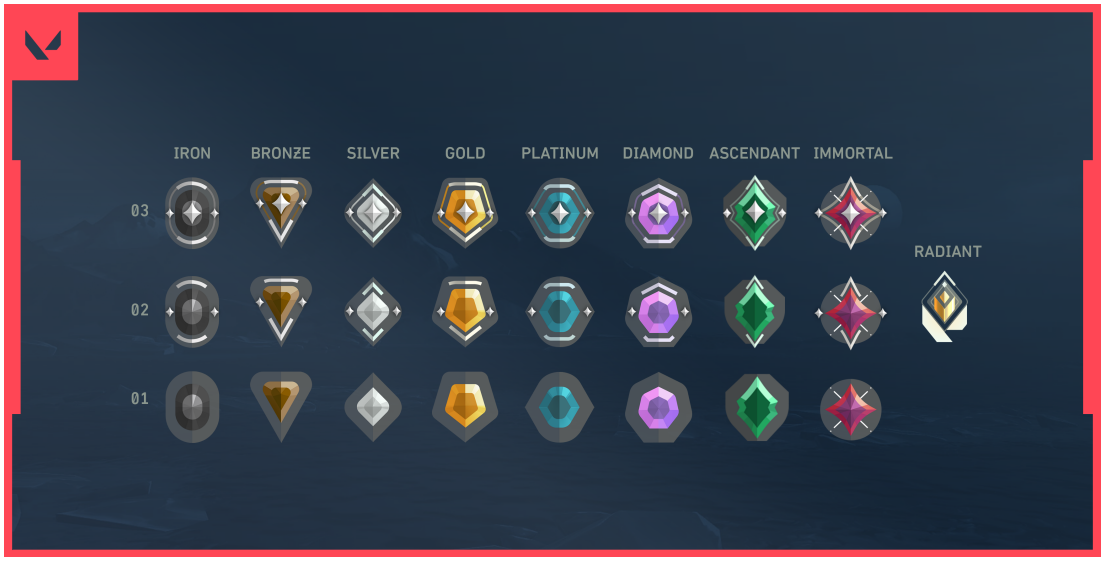

Figure 2.1: Valorant ranks from worst to best (left to right) and tier (bottom to top)

who reach over 100 RR are promoted to a higher tier. If they are already in the highest tier of the rank, they are promoted to a higher rank. Players who lose a few games with 0 RR are demoted to a lower rank.

Note: Valorant uses a term *rank* for one of the values of the visible rating [whatacoolwitch(2023)], but also as a visible rating itself that is comparable to MMR [meowzers(2023)]. In this thesis, we will use the term *rank* as one of the values of the visible rating, e.g., *Iron*, *Bronze*, *Silver*, ..., and we use VR to reference the visible rating as a whole, e.g., *Immortal (Tier) 3, 25 RR* that can be compared with MMR

VR and MMR are not always equal or approximately equal. However, their values should converge closer together after each game. The amount of RR players gain or lose depends on their performance and MMR.

Let teammates p_1, p_2 have visible rating VR_1, VR_2 and matchmaking rating MMR_1, MMR_2. Assume that

$$MMR_1 = MMR_2 \wedge MMR_1 - VR_1 > MMR_2 - VR_2$$

(i.e., the difference between MMR and VR of the player p_1 is smaller than the difference of the player p_2). If they win together and perform similarly, $p1$ will receive more RR than $p2$.

Players can guess how close their VR is to their MMR by how much they gain or lose after a game. If the values approximately correspond to each other, then their MMR is close to VR. If their VR gains are significantly higher than the VR deduction, their VR is much less than MMR. Otherwise, if a player loses significantly more VR after a loss, their VR is greater than MMR.

Episodes and Acts

A calendar year in Valorant is divided into two episodes, each around six months long. An episode is further split into three acts. Valorant adds new cosmetics and features or balances existing game mechanics between these periods. More extensive updates are rolled out at the start of episodes. Meanwhile, minor game tweaks or bug fixes are released between acts.

Episodes are also important for player rating resets. The MMR system slightly reduces players' VR and MMR. Players' VR is hidden for the next five games.

Characters

Players select a character (denoted as *agents* in Valorant) before each match. A team in the main gameplay can only have unique characters. The game provides a growing selection of characters divided into four roles according to the type of their abilities: initiators, sentinels, controllers, and duelists. A typical team composition requires one character for each role, and a fifth character is determined based on a map or a team preference. That is why the position is called flex.

Characters have five unique abilities, including a signature ability and an ultimate. Every round, characters get a signature ability free of charge. The ultimate ability is charged by planting or defusing the spike, dying or killing an enemy, or capturing an *ultimate orb* in strategically contested places on the map. The number of charges that the ultimate ability requires depends on the strength of the ability, and it ranges from 6 to 9 charges. The remaining abilities must be purchased before the round starts.

Maps

There are currently eleven maps for the main gameplay[1], each designed to have a unique feature listed in Table 2.1.1. Only seven of those maps are in the active map rotation [Robles(2024)], which changes every Episode. Players can only play these maps in game modes except in custom games, where players can choose any map.

Map	Unique feature
Bind	One way teleport gates
Haven	First map with three sites
Split	First map with vertical zip-line
Ascent	First map with breakable horizontally opening doors
Icebox	Horizontal zip-line
Breeze	Trap door that opens when a player is near
Fracture	Attacker's spawn site is split
Pearl	First map with no unique feature
Lotus	Revolving doors
Sunset	None [2]
Abyss	Some places without outer boundaries

Table 2.1: List of all maps and its unique features

[1]Valorant offers additional maps for different game modes

Matchmaking

Players join matchmaking by queuing up. Then, Valorant selects a map and ten players from the queue for a match. Players enter the character selection phase with their teammates. They can communicate with each other and non-bindingly select a character they want to play. When a player is sure of the character, the player can lock in the selection, making the character unavailable for teammates. When a time limit of the character selection ends, and someone has not selected or locked in any character, the selection fails, and the rest of the players are returned to the queue. Otherwise, players begin to load into a match, and at the end, players' ratings change based on their performance and the match outcome.

Fair-play reinforcement

Valorant detects cheaters[3] using in-game reports and an anti-cheat software called Vanguard. In-game reports are generally reviewed manually. Not every player who receives a report is punished. In August 2020, Valorant developers reported that 3% of the player base received a cheat report, and only 0.6% of the player base received a cheat report from more than one player. Of course, not everyone who received a cheat report cheated, so Valorant provided more stats: 53% of caught cheaters were banned before they were reported, and 60% of players with 20 or more reports were banned after a review [Chamberlain(2020)].

Since Valorant has not provided specific numbers, we cannot calculate how many players were caught cheating. However, some third-party applications estimate game player bases. One of the apps, Activeplayer[4], revealed its statistics, which showed approximately 20,760,144 active players in August 2020 [Chamberlain(2020)]. For a rough estimation, let us assume that every player who received more than one cheating report is a cheater. Combined with the number of active players, there could have been 124,560 cheaters in summer 2020.

Valorant API

Valorant provides API[5] for developers to use in-game data to create public user applications. It is possible to retrieve data about finished matches, including player statistics like damage dealt and ability usage[6]. However, the project has to follow the Valorant Developer API Policy[7] and must be submitted for a review before developers can access the data. Moreover, automating custom match creation violates Riot Games Terms of Service[8]. Riot Games does not allow scripts to interact with Riot Services in any way or for any purpose.

[2]In the development phase map had an acid pool that caused agents to take damage.

[3]Players with malicious intent use third-party software to gain an unfair advantage over other players.

[4]`https://activeplayer.io/`

[5]`https://www.riotgames.com/en/DevRel/valorant-api-launch`

[6]`https://developer.riotgames.com/apis`

[7]`https://developer.riotgames.com/docs/valorant`

[8]`https://developer.riotgames.com/docs/valorant`

Similarity to Counter-Strike

Counter-Strike and Valorant are often compared because their main gameplay and objectives are very similar but only with different terminology: two teams, Terrorists and Counter-Terrorists, are fighting over two bomb sites. The Terrorists try to eliminate every Counter-Terrorist or plant a bomb and then prevent it (a C4 explosive) from being defused. Counter-terrorists' objective is to protect bomb sites. The main difference between these games is that characters in CS do not have abilities. However, they can buy different types of grenades instead.

Because of the similarities between these two games, the games are often compared.

2.1.2 Player rating system

The main issue with the player rating system is that it is non-transparent. Valorant only provided a rough idea about the rating calculations described in Section 2.1.1. This prevents gamifying the system; that is, players modify their gameplay to gain more favorable statistics for the system, e.g., if dying less would be favorable, players would play safely and bait their teammates to die first instead of helping them. It also hides their know-how so their competitors cannot use or improve the system for their purposes.

Apex Legends[9] is an example of a FPS game with a transparent player rating system [EA(2024)]. During a competitive season in 2023, developers modified their player rating system to favor players who survived longer than players who defeated more opponents. Players modified their gameplay to survive longer instead of defeating everyone and winning the game. A player could achieve the highest rank, Apex Predator, simply by hiding and surviving without dealing a single damage to enemies [Wilde(2023)].

Players often get frustrated with the player rating system because they think they deserve more RR for a win or less RR decrease from a loss. The difference between VR and MMR hugely impacts the player rating update, where MMR is not visible to players through official Valorant sites.

The issue with VR and MMR reduction at the end of Episodes occurs shortly after the start of the following one when players try to regain their MMR. Players who have already regained their previous MMR are often matched with players who have not. That creates uneven matchmaking, and a team with more players who have gained their MMR back is at a disadvantage.

2.1.3 Player selection

A balanced matchmaking system is crucial for the best gaming experience. Players want to spend little time in a queue while still getting into a match with similarly skilled players. If they were matched to play against stronger players, they would be rightfully upset, and playing against weaker players feels unfulfilling.

[9] `https://www.ea.com/games/apex-legends`

Missing role selection

Before players join a queue, they cannot choose their preferred roles. Moreover, in character selection, players are not assigned a role [VALORANT(2024)]. Therefore, players can choose any characters their teammates have not locked in yet. This increases the diversity of possible team compositions and is fully utilized in a professional scene where teams create unique compositions by picking multiple agents from a role based on the team strategy. We can observe this in the most important international Valorant tournament in 2024, Valorant Champions. EDG, a Chinese team, came up with an unusual composition. Their team composition consisted of two controllers, two initiators, and one duelist, thus omitting an agent from a sentinel role [VLR(2024)].

Although a flexible team composition benefits the competitive scene, it has drawbacks in an uncoordinated competitive mode. Players often meet each other in-game for the first time and tend to pick the most aggressive agents to make the most impact. They also select agents they like or want to play with. In the character selection phase, players commonly race to lock in the agent they want to play. Duelists Jett and Reyna, and the most aggressive controller, Clove, are the most picked characters across the ranks. Players often race to pick one of these agents instead of balancing players' character preferences and optimal team composition [Tracker(2024)].

Optional team voice communication

A massive part of Valorant gameplay is team voice communication. It is essential not only for professional teams but also for casual players. The players are more likely to succeed if they communicate and coordinate their gameplay. Team communication is also vital for gathering information and predicting enemy movements.

Voice communication is not enforced in any Valorant game mode, and it is not against the rules to not communicate. Therefore, players can play without communicating or even listening to the team communication, which makes teamwork harder.

Disruptive player behavior

Players can be matched with everyone with a similar MMR. Players cannot add players to a blacklist[10]. This missing option can be reasoned to minimize the queue time. However, in free-to-play games, players tend to break the code of conduct.

Cheating. Cheating is a massive issue in almost any game of any type. It is also challenging to distinguish skilled players from cheaters. Undetected matches with cheaters can also skew the ratings of involved players. If cheaters win, their teammates are rewarded with undeserved player ratings.

[10]List of players to exclude from getting to the exact match

Smurfing. Smurfing is when a player queues up on an alternate account with a lower MMR than the player's main account. Let us say this player queues a competitive match on an alternate account. Valorant will create a lobby that does not consider the player's actual MMR, and the game will be uneven. Players often use alternate accounts to play with friends of lower MMR.

Toxicity. When players are frustrated in a game, players often express their frustration through toxic messages. That is an unwanted behavior according to Valorant developers [Sara "Riot necrotix" Dadafshar(2022)]. Each player has a communication rating, which evaluates how adequate the player's communication with others is. Players can receive different types of punishment based on the offense's severity, ranging from a warning to a permanent ban from playing the game.

Queue sniping. Queue sniping can be quickly done by visiting a streamer's live broadcast and joining the queue exactly when the streamer joins the queue. It gives the queue sniper a higher chance of getting into the exact match if they have a similar MMR.

Stream sniping. Stream sniping is when someone in the streamer's enemy team visits the streamer's stream to gather information about the game (strategy, players' buys, and location on the map) that is not visible to the stream sniper in the game, thus gaining an unfair advantage.

Missing community matchmaking

Players cannot choose a subset of players they want to play with, which would benefit professional players by allowing them to get practice comparable to professional matches.

In competitive mode, players cannot select the role they want to play, which means professional players often play roles that are not their primary role in their professional team. Moreover, players often select roles they want to play instead of roles the team needs. The team composition might have many missing pieces, making the team weaker from the character selection only.

2.1.4 Map selection

On Valorant's release, map selection was entirely random. It was not optimal because players would sometimes play the same map multiple times in a row, creating a map streak. Therefore, the map selection was improved and optimized to reduce the occurrence of map streaks [Chang(2022)]. Valorant did not specify if a streak ends or is considered long-term when a player gets a different map in the following selection.

However, the map selection does not consider how much players like or dislike specific maps. Therefore, players can get streaks of their favorite or least-liked maps, causing an uneven game experience.

A fully automated map selection is the most convenient method for players, as it chooses the map for them and does not require time for a map selection phase.

However, the benefits come with a sacrifice of flexibility. If the method favors one map because most players in a match have not played it in recent games, it might disadvantage a minority of players who have played the map recently.

Valorant provides only modes with fully automatic map selection, except for custom games, in which players must select a map and players and split them into teams.

2.2 Third-party matchmaking options in games

In this section, we observe how third-party application takes advantage of the matchmaking shortcomings of their respective games.

2.2.1 NeatQueue

NeatQueue[11] is a Discord Bot for custom matchmaking for a variety of games. It supports different configurations, e.g., the number of teams in a match, the number of players in a team, and player selection. We describe the matchmaking steps of NeatQueue configuration used in Pro City for Valorant

0. Players without a rating are assigned an initial rating of 1000.
1. Players join a queue in a Discord server.
2. When the queue reaches ten or more players, the first ten players are selected, and a lobby with voice communication is created.
3. When chosen players join the lobby, two captains are randomly selected from four players with the highest rating.
4. Captains alternate choosing players for their team.
5. Captains alternate removing maps until one map is chosen to play.
6. Discord Bot moves players from the lobby into separate team voice channels.
7. Players decide on who creates a custom lobby in Valorant.
8. Players invite each other to the lobby and set up parameters for the match.
9. When both teams in chat confirm their team is ready, the lobby creator starts the game.
10. Players submit the winning team.
11. The Bot updates player ratings.

Player ratings reset to the initial rating every month. This NeatQueue configuration heavily relies on players' participation during matchmaking. The process of player selection and removing maps is time-consuming. However, NeatQueue does offer automated map selection[12]: random and least frequent. Player selection can also be automated[13] based on players' rating or randomly.

[11] `https://www.neatqueue.com/`

[12] `https://docs.neatqueue.com/#/?id=map-selection`

[13] `https://docs.neatqueue.com/#/?id=teamselection-set`

2.2.2 FACEIT

FACEIT[14] is a platform for a gaming community that focuses on high-level competition. It supports many games, including Valorant, but it is most famous for hosting Counter-Strike games. FACEIT has a custom matchmaking system, a player rating system, private servers for matches, and even an Anti-Cheat client. It has free and two paid tiers, giving players a premium experience thanks to more balanced games, preferred maps, and rewards. Players can use FACEIT for matchmaking on a browser, or they need to download a desktop app.

FACEIT uses a term *Elo* for a value describing players' skill strength. In this section, we will use the term *rating* instead to not be confused with the Elo rating system described in Section 2.3.1.

Players join a queue to start a matchmaking. When FACEIT creates a potential lobby, players have to accept it. Then, two captains are randomly selected while prioritizing players with a paid tier. Captains then ban maps until one map remains. Then, FACEIT provides a command that players enter in-game to connect to the match. Match parameters, like selected map and team assignment, are automatically set. That is possible because FACEIT hosts matches on their servers. Furthermore, FACEIT receives and handles match data without needing input from players.

Upon creating an account, players receive an initial rating of 1000. Matchmaking uses players' ratings to balance a match, and it is updated based on match results and player performances. FACEIT also does not provide an exact variant of the rating system [Raphael(2024)].

FACEIT in Valorant

FACEIT does not have a custom matchmaking and player rating system for Valorant as they have for Counter-Strike. However, users can create and join community Valorant tournaments or community hubs. Tournaments have a single elimination bracket system, and players can only join as a team. Meanwhile, Hubs are for players to create community matchmaking. Players play against each other in the hub, and the matchmaking process is almost identical to Counter-Strike. Since FACEIT does not host Valorant matches on their servers, they cannot control match creation and in-game statistics. Players must create a custom game and ensure other players join the lobby. After the match, a captain enters the outcome, and the opposing team's captain confirms the result.

2.2.3 Champions Queue

Third-party matchmaking is not popular in LoL because of the high demand of players for reasonable queue time due to strict role selection (unlike in FPS games like Counter-Strike or Valorant). However, Champions Queue[15] was created for amateur and professional players as an alternative in-game competitive matchmaking. Its purpose is to create a better practice environment. It was hosted on a private Discord server using a Discord Bot, where players can communicate with each other. These matches were hosted on special servers to ensure low latency.

[14] `https://www.faceit.com/`

[15] `https://championsqueue.gg/`

Players can play in the Champions Queue for a limited period outside of team practices and official esports game times. Champions Queue activity was paused in 2024 due to low player participation [Gafford(2024)]. However, it returned unofficially for a LoL international Worlds 2024 tournament [Apples(2024)].

2.3 Chess rating system

Chess rating systems are popular because they use only game outcomes and players' ratings for their calculations. For this reason, it is easily adaptable and applicable to other games, for example, in FPS games CS:GO and Team Fortress 2[16] [Glickman(1995)]. It is also used in a Multiplayer Online Battle Arena (MOBA) game DOTA 2[17] [2(2024)].

2.3.1 Elo rating system

International Chess Federation[18] (FIDE) has assigned Arpad Elo to develop a rating system under their development program. Elo then published the Elo rating system in the 20th century [Elo(1986)]. The Elo rating system was adopted by FIDE in 1970 and by the US Chess Federation[19] (USCF) in 1980. Each of them slightly altered the system for their own calculation needs.

USCF rating system

USCF implemented a US Chess rating system for USCF Chess Events. We are looking over the USCF Chess rating system because it is one of the ratings that uses equations to calculate all its variables and coefficients. The rating updates works as follows [Mark E. Glickman(2020)]:

$$r_{new} = r_{old} + K \cdot (s_a - s_e)$$

where r_{new} is a rating after the game, r_{old} rating before the game, s_a actual score of the game and s_e expected score of the game.

$$s_a = \begin{cases} 1 & \text{if player won} \\ 0.5 & \text{if draw} \\ 0 & \text{if player lost} \end{cases}$$

$$s_e = \frac{1}{1 + 10^{(r_{opponent} - r_{old})/400}}$$

where $r_{opponent}$ rating of the opponent before the game. The interpretation of the equation is as follows: if the opponent has a 400 rating more than the player, then the opponent has a ten times higher winning chance than the player [Berg(2020)].

Moreover, the Elo rating system depends on a constant K, which defines how much a game outcome should impact a new rating. Table values between [8,46]

[16] `https://www.teamfortress.com/`
[17] `https://www.dota2.com/home`
[18] `https://www.fide.com/`
[19] `https://new.uschess.org/`

initially suggested the coefficient based on the number of games played. Instead, the USCF modification of the Elo rating system uses an equation for calculating the constant K

$$K = \frac{800}{N' + m} \tag{2.1}$$

where N' is the number of total games and m is the number of games in the tournament for which the rating is calculated. This formula is used when $N' + m \geq 10$. Thus, K is limited to $K \leq 80$. The maximum value of K needs to be limited so the game outcome does not overly impact the player rating.

The minimum rating value is 100, and the initial rating is calculated using different chess ratings supported by the USCF. If a player has no conversible rating, then the player is rated by their age at the end of the tournament:

$$\text{USrating} = \begin{cases} 100 & \text{if Age } < 2 \\ 50 \times \text{ Age} & \text{if } 2 \leq \text{ Age } \leq 26 \\ 1300 & \text{if Age } > 26 \end{cases}$$

2.3.2 Glicko rating system

Glicko rating system [Glickman(1995)] was designed to improve the Elo rating system [Glickman(1999)], which struggled with a time difference of players' last Elo rating update. For example, a player who recently achieved an Elo rating of 1700 is rated equally to one who reached the same Elo rating ten years ago. The Elo rating might be less accurate with a more considerable time difference. For this reason, the Glicko rating introduces a new variable, Rating Deviation (RD). It reflects the accuracy of the Glicko rating based on the last updated rating time. Glicko assumes that players are rated in rating periods, which can be defined by the user. It can be a period of one grating ame, a tournament, or a series of tournaments. RD is then dependent on uncertainty between rating periods. Based on each player's Glicko rating and RD, player ratings are updated.

Player rating update

1. Determine RD for the start of a current rating period based on the RD at the end of the previous rating period:

   (a) if a player does not have a rating r yet, then $r = 1500, RD = 350$

   (b) otherwise $RD = min(\sqrt{(RD)^2_{old} + c^2}, 350)$

   Where c is an uncertainty constant between rating periods. Glicko proposes calculating the uncertainty constant using the following equation:

   $$RD_{max} = \sqrt{RD^2_{typical} + p \cdot c^2}$$

   Where $RD_{max} = 350$, $RD_{typical} \approx 50$, and p is the number of time periods for a player's rating to be as unreliable as an unrated player's rating. Glicko rating penalizes frequent players because their decreasing RD after each game results in more minor rating updates over time.

2. Determine post-period rating and RD:

   Let $r_1, r_2, \ldots, r_m$ and $RD_1, RD_2, \ldots, RD_m$ are pre-period ratings and rating deviations of m opponents and $s_1, s_2, \ldots, s_m$ are outcomes against each opponent where $s_i \in \{0, 0.5, 1\}$. The players' updated rating r' and rating deviation RD' are calculated using the following formulas:

$$r' = r\frac{q}{1/RD + 1/d^2}\sum_{j=1}^{m} g(RD_j)(s_j - E(s|r, r_j, RD_j)) \tag{2.2}$$

$$RD' = \sqrt{\left(\frac{1}{RD^2} + \frac{1}{d^2}\right)^{-1}} \tag{2.3}$$

   where

$$q = \frac{\ln 10}{400}$$

$$g(RD) = \frac{1}{\sqrt{1 + 3q^2(RD^2)/\pi^2}}$$

$$E(s|r, r_j, RD_j) = \frac{1}{1 + 10^{-g(RD_j)(r - r_j)/400}}$$

$$d^2 = \left(q^2 \sum_{j=1}^{m} (g(RD_j)^2 E(s|r, r_j, RD_j)(1 - E(s|r, r_j, RD_j))\right)^{-1}$$

$E(s|r, r_j, RD_j)$ describes the probability of the player with rating r of defeating the player with rating r_j and rank deviation RD_j. d^2 represents the uncertainty of players' rating and variability due to the game outcomes. $g(RD_l)$ regulates influence of the corresponding rating r_l based on RD_l in the Equation 2.2.

Glicko 2

Glicko 2 is a successor to Glicko, which considers player performance consistency. Besides rating deviation, Glicko 2 also tracks rating volatility, which describes player rating expected fluctuation. If a player has a stable performance and suddenly starts winning against noticeably stronger opponents, then the player's volatility is high. Otherwise, players have low volatility if they perform similarly over time. The Glicko 2 algorithm exhibits few distinctions from Glicko because game outcome alone is insufficient to indicate inconsistent player performance.

2.4 Recommender systems

Recommender systems are algorithms designed to suggest relevant content to users. These systems are important because users have an increasing number of available choices. They can quickly get fatigued by the sheer number of options or be displeased with the service if they cannot find relevant content. Services

can easily retain and prolong user attention by recommending suitable content. We can utilize these methods to recommend maps to a set of players.

Recommender systems need data to understand user preferences. The easiest way to obtain high-quality data is to get user feedback on specific content. Users can provide information about whether they liked or disliked the content or, on a scale, how much they enjoyed it. Furthermore, data can be obtained via implicit feedback, such as the history of consumed content, how long users spend on specific content, or search patterns.

The applications of recommender systems are endless, most notably in e-commerce product recommendations, streaming service content suggestions, and social media content recommendations.

Collaborative and content-based filtering are among the most common recommender systems. Collaborative filtering suggests content based on the preferences of similar users. It predicts that users with common interests similarly like or dislike other items. Meanwhile, if a user likes specific content, then content-based filtering recommends similar content that another user likes. Instead of applying one recommender system, it is quite common for algorithms to implement a hybrid approach of multiple filtering methods.

2.4.1 Group recommender systems

The algorithms we covered so far cannot effectively recommend content for a group of users. For example, an algorithm must consider the preferences of every user when a group of friends wants to watch a movie together. For this issue, we use group recommender systems, which try to balance the preferences of each individual in a group. Some of the best-performing methods based on an experiment by Judith Masthoff [Masthoff(2010)] were Average (prefers content with best average), Least Misery (selects a content based on the lowest rating across users), and Average without Least Misery (chooses content with the best average preference without items below a certain threshold).

However, the above-mentioned approaches do not consider whether the chosen content favors specific users. They select content fairly at the item level and do not check if everyone prefers selected items. The potential deficiency of selected items for specific users is not reflected in the following item selection.

Exactly-Proportional Fuzzy D'Hont's Aggregations

Exactly-Proportional Fuzzy D'Hont's Aggregations
(EP-FuzzDA) [Malecek and Peska(2021)] is a greedy algorithm. It selects items based on how relevant the chosen items are to users and considers the relevances in the following item selection.

The algorithm, taken from work by [Malecek and Peska(2021)], tracks users' relevance to selected items and tries to balance them in the following item selection. User priority can be modified using user weights. For example, if certain users do not prefer the item in the previous selection, we can increase their weights for the following selection to enforce long-term fairness.

Algorithm 1 Exactly-Proportional Fuzzy D'Hont's Aggregations

Input: users $u \in U$, items $i \in I$, relevance scores $r_{u,c} \in R$, number of items k, user's weights $v_u; \sum_{\forall u \in U} v_u = 1$

Output: ordered list of group recommendations L_G of length k

$L_G = []; TOT = 0; \forall u : r_u = 0$
for $j \in [0, \ldots, k]$ **do**
  for $i \in IL_G$ **do**
    $TOT_c = TOT + \sum_{\forall u \in U} r_{u,c}$ ▷ item's total relevance
    $\forall u \in U : e_u = max(0, TOT_c \cdot v_u - r_u)$ ▷ expected utility of an item
    $gain_c = \sum_{\forall u \in U} min(r_{u,c}, e_u)$
  end for
  $c_{best} = argmax_{\forall i \in I}(gain_c)$
  append c_{best} to L_G
  $\forall u \in U : r_u = r_u + r_{u,best}$ ▷ updates user relevances of selected items
  $TOT = \sum_{\forall u \in U} r_u$ ▷ recalculates total relevance of selected items
end for
return L_G

2.5 Discord

Discord is an online social platform for communities to chat and talk. Initially designed for gaming purposes, it was later rebranded as a place for everyone to talk and hang out [Citron(2020)]. Currently, Discord has over 200 million active users monthly [Patel(2024)].

2.5.1 Structure

Users can create or join servers. They consist of text and voice channels that can be divided into categories. Servers can be private or public. Public servers are searchable using their name or filtered by their category. In public servers, users can browse and read the messages in the channels without joining. Private servers are not searchable or viewable unless users join first. Users can join the server only by receiving an invitation link. Invitation links can have an expiry date or a maximum number of uses after which the invitation is no longer valid. Only after joining can the user view and interact with channels.

If a user creates a server, the user becomes a server owner. The server owner has all permissions enabled. The server owner can allow or deny members specific permissions using roles. For example, they can read or write only in specific text channels, join or talk in voice channels, view specific categories, create or give roles to users, and create or delete channels. Discord has a *linear role hierarchy*: Upon creation, roles are assigned an integer position. Roles with lower positions cannot modify roles with higher positions, e.g., change permissions or assign roles to users. If roles have equal role positions, they are sorted by role ID.

2.5.2 Discord Applications

A Discord application enables developers to create a Discord Bot. A Discord Bot can customize a server, send and manage messages, and interact with users.

Users with *Manage server* permissions can add Bots to the server using Bot's invitation link. Before joining the server, Bots request specific permissions to function correctly. Users can interact with Bots using commands. Bots are treated as users who joined the server but with a unique tag, `BOT`, next to their name.

Interactions

The user starts interacting with a Bot using commands in allowed text channels on the server. There are three types of commands:

- slash commands - after typing a forward slash / in a text channel, Discord offers users available commands in a server
- user commands - when a user right-clicks a selected user on a server, Discord shows a context menu with *Apps* section, in which the user can select a command and create interaction with the selected user
- message commands - works the same way as user commands, but with messages in text channels

The Bot then receives a command interaction with the user, guild, and interaction data. After processing the data, the Bot replies with a message and sometimes with buttons, select menus, or other message components for additional interactions. The Bot interaction reply messages are primarily ephemeral, meaning they are visible only to the user interacting with the Bot, and the user can also hide the message in contrast to the regular messages sent by users that are visible to everyone.

Discord API

Discord provides Discord API[20] for developers to create Discord Apps. With Discord Apps, developers can create Discord Bots. Discord Bots can be added to multiple servers while maintaining its functionality. Discord Bots are added to a server like a user with a special `BOT` role and tag. They can be added to multiple servers.

Discord API has two main parts. HTTP API is a REST API for interacting with core Discord resources like servers, channels, users, and messages. At the same time, Gateway API enables developers to receive event data on servers in real-time. Gateway API maintains a persistent websocket connection between the Discord app and servers with the Discord App. Discord Apps can listen to emitted server events, e.g., a member joined or left the server, or a member is now online or offline.

Working directly with the Discord API would be very challenging for basic developers looking to build a Discord App. That is why Discord's website recommends some third-party libraries that are actively maintained and popular in communities[21].

[20] `https://discord.com/developers/docs/quick-start/overview-of-apps#what-apis-can-apps-use`

[21] `https://discord.com/developers/docs/developer-tools/community-resources`

Discord.js is one of the most popular libraries listed by Discord. It allows developers to create a Discord App using a more object-oriented method than the Discord API. It is a Node.js module that covers all of Discord's API features. Working with this library is more straightforward and more time-efficient.

2.6 Takeaways

Finally, we set goals for the thesis based on the issues raised in this chapter. The main objective of our thesis is to create a Discord Bot for third-party matchmaking focused on user experience.

Role-based matchmaking in LoL could be applicable in Valorant upon modifications. However, community matchmaking inside a Discord server drastically lowers the player base count on which queue time is heavily reliant. Applying role-based matchmaking or other player selection filtering would increase the queue time even more. Nevertheless, community matchmaking puts server owners in complete control of the player base and can easily remove disruptive players described in Section 2.1.3.

CS2 lets players decide which map they want to play in competitive modes. However, players spend a nontrivial amount of time doing so. We want to allow players to vote for the map they want to play but limit time in the map selection process. To make the process faster and easier for players to decide, we select the three most favored maps based on player preferences. Players can update their map preferences to reflect the changes in their liking over time. We use group recommender systems to evaluate their preferences. We also maintain long-term fairness in the map selection process for all players.

We want to collect all possible match data to determine player performance accurately. However, Valorant does not allow using Valorant API for alternative matchmaking and player rating systems[22]. Therefore, we have to rely on players to provide match result data. We do not want to burden players with entering various match details, and we will require players to submit only the game outcome. For this reason, we opt to modify a chess rating system that calculates player ratings using past player ratings and game outcomes. Because our rating is based only on winning or losing, we can make our rating system transparent without undesirable impacts on players' in-game behavior.

We use the NeatQueue Discord Bot matchmaking process, which complies with our requirements. We adjust the matchmaking steps as discussed above. Furthermore, we select one of the players to create an in-game custom lobby, and since players submit the result, we require teams to agree on it as implemented in FACEIT.

[22] `https://developer.riotgames.com/docs/valorant`

3. Problem analysis

In this chapter, we propose a solution that respects the analysis and goals stated in Chapter 2.

3.1 Bot behavior

After the Bot joins a server, an event `guildCreate` emits, and the Bot creates informational and game categories. The informational category hosts text channels for users read-only: `#general` for matchmaking updates, `#match-history` for logging resolved matches, and `#reports` for highlighting disruptive players.

We reserve the game category for temporary lobby channels, i.e., lobby chat text and team voice channels. The Bot will handle channel creations and deletions without needing help from members. Since a category can hold up to 50 channels, the Bot may create more game categories if existing categories cannot hold needed channels for a new lobby.

The Bot also creates admin roles and assigns them to the server owner. We discuss admin roles in Section 3.1.4 below.

After the Bot's startup, an event `clientReady` emits, during which the Bot will scan all the servers it is a member of. If anything is out of order for proper usage, the Bot tries to resolve it and notifies the server owner in direct messages. The Bot can be stripped of the permissions needed during its uptime. Therefore, we listen for `GuildRoleUpdate` event and check for permission changes.

If a player leaves the server, we remove the player data in `guildMemberRemove` event. When the Bot is kicked from a server, we handle an `guildDelete` event and delete created roles, categories, and channels within them.

To add quality-of-life features, we handle `userUpdate` and check for username changes to update player data. Next, we observe `PresenceUpdate` event to check if a player changed status to offline during the matchmaking process.

When users interact with the Bot, we process an `interactionCreate` event. We cover the procedure in Section 4.1.3.

3.1.1 User interactions

The Bot provides slash commands for members to use in text channels. We do not need to restrict sending commands to specific channels because the Bot's replies are ephemeral. Suppose a slash command involves other members or is channel-dependent. In that case, the Bot will require the sender to provide the channel as an argument. In addition, the response will be non-ephemeral and will mention involved members.

The Bot's features do not require user or message commands. Since we provide only slash commands, users do not have to remember the command type they want to use and where to locate them.

Attachment A.1 lists all available commands. Additionally, users can browse them on a server using the command `/help`. In this thesis, we look closely into the most notable commands from the user perspective in Section 3.1.2, 3.1.3, and from the admin perspective in Section 3.1.4.

3.1.2 Ideal matchmaking process

In an ideal matchmaking process, illustrated in 3.1, players join a queue using the command `/queue` and leave the queue with `/leave-queue`. Players are notified about their missing map preferences and are encouraged to fill them. Players can add or change map preferences anytime via `/change-map-preferences`.

The Bot creates a new lobby when the queue reaches ten players. Then, the Bot creates text and voice channels in a suitable game category. It mentions the players in the text channel. Discord will create a notification when users are mentioned. The Bot creates a map selection message to vote for one of the three maps selected based on the players' map preferences. The message includes buttons for each map, and the Bot tracks player votes.

Players verify their presence by joining the lobby's voice channel, emitting an `voiceStateUpdate` event. While waiting for players to join, players can vote for a map they want to play. When all players join, then the Bot selects a map. If no one votes, an algorithm, described in Section 3.3, will select the best. Otherwise, players will play on the map with the highest vote count. We randomly choose one map if a subset of maps ties in votes.

Then, the Bot requests one of the players to create an in-game lobby and send a party code in the text channel for other players to join. Finally, the Bot informs players that the match can start, assigns players to teams, and moves them to separate voice channels. The players are informed about the selected map, team compositions, and sides. The message also includes two buttons for players to select the winning team. A player from one team must submit, and another player from the opposing team must confirm it (the whole winner selection diagram in case of a submit rejection is depicted in Figure 3.2).

When the game outcome is confirmed, the Bot announces the winner and player rating updates in the text channel. The Bot also logs the match in the `#match-history` channel. Lastly, players can rate how much they enjoyed playing the selected map. If they vote, then their map preference is updated. Every channel created for the match is deleted after a delay, allowing players to review the result.

3.1.3 Unwanted matchmaking scenarios

This section deals with situations in which players do not cooperate or disagree.

A match starts when players from a lobby join the voice channel. Players can substitute a player who does not respond and has not joined the voice channel yet by `/sub-player`. We allow player substitution only when a match has not started, i.e., the details have not yet been decided. The substitute can have a different player rating and map preferences. Thus, the Bot would have to recalculate the match details, which can result in different team compositions and map selection. The changes can make players feel disadvantaged. We allow players to substitute only in a lobby to eliminate situations where players substitute for a higher-rated player to create a higher-rated team.

Let us assume that teams disagree on the game outcome. In that case, we reset the winner selection and remove players that voted from the next attempt. When the following pair of players cannot confirm the winner, the Bot requests help from admins. Then, admins investigate the situation and resolve the case.

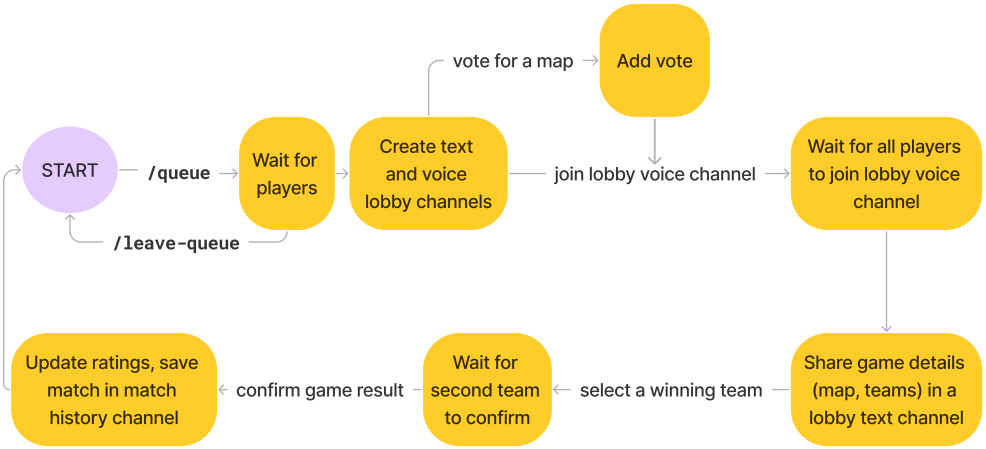

Figure 3.1: Ideal matchmaking loop from the user perspective

When an unexpected situation occurs that players cannot handle themselves, they can ask the admins to resolve it. If the problem is unsolvable, the admin can cancel a lobby or a match. Cancellations will not impact player statistics.

3.1.4 Admin roles and commands

We provide admin roles and commands to modify server matchmaking, such as resetting player statistics, managing matchmaking issues or resets, and managing disruptive players. The Bot creates roles for Admins and SuperAdmins and assigns them to the server owner.

Admins can cancel lobbies and matches using commands `/cancel-lobby` and `/cancel-match`. They can also decide a match result using `/set-match-result`. They can submit it in every stage of the winner selection scenario described in Figure 3.2 to prevent situations in which players sabotage the selection by not submitting or confirming the game result.

If Admins decide a player is too disruptive, they can ban the player via `/ban-player`. Banning players remove them from a server and cannot rejoin the server unless the server owner revokes the ban in the moderation server settings. Note that Discord already implemented the ban command. We implemented our ban command to make the Bot self-contained. Furthermore, we require a reason for the ban compared to the built-in command, which is optional.

Admins can reset leaderboard and player statistics with the command `/reset-act`. The term *act* is selected to resemble the *act* in Valorant as mentioned in Section 2.1.1. When executing the command, admins select maps for players until the next reset.

SuperAdmins can do the above-mentioned operations that an Admins can. Furthermore, they can add or remove Admins using `/add-admin` or `/remove-admin`. This role can seem redundant if assigned only to server owners. However, if the server owner does not want to manage admins, they can select other members to be SuperAdmins.

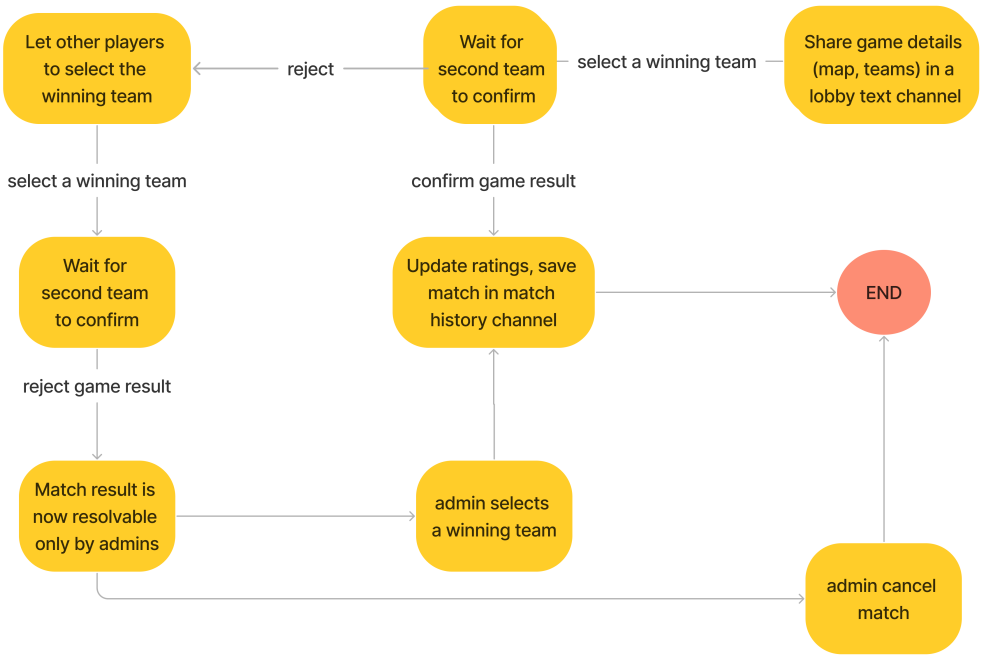

Figure 3.2: Diagram of every winner selection scenario

Server members with role management permission can create a role with an identical name. To avoid this issue, the Bot saves and stores the role ID when creating the role. In addition, we cannot make admin commands visible only to admin roles through Discord API. It is possible only to show specific commands based on needed permissions, but admin roles do not require any. In addition, if we set admin commands visible based on permissions, other roles may have the same permission. Thus, they will be visible and executable by members with these roles. However, server owners can only manually set the Bot's command visibility for admin roles. We will not require server owners to do so. Therefore, the Bot will check for Admin roles before executing the command logic.

3.2 Player selection

Before starting a new game, we must split players fairly into two teams. Instead of automating the process, we can choose captains, e.g., by selecting two players with the highest rating. Then, they choose their teammates one by one. However, this process takes a nontrivial amount of time. In addition, captains often select players with the highest rating or based on the subjective opinion of the players. They can feel uncomfortable and less valuable or discriminated against by being chosen towards the end.

That said, we opt for an automated player selection with a greedy approach. Players will be sorted by their rating from highest to lowest and, one by one, assigned to an incomplete team with a lower average rating. For example, if a team t_1 has three players with an average rating of 1200 and a team t_2 has two

players with an average rating of 1300, then the next player will be moved to t_1 even though t_1 currently has more players than t_2. This method gives t_1 a chance to balance the average team rating by adding an unassigned player with the highest rating, while t_2 will be assigned a player with a lower rating in the following steps.

Lastly, we decide on side selection randomly since players change sides after halftime of a game.

3.3 Map selection

Each map is different and has unique features, and therefore, players can have enormously different preferences. Player map preferences can also change over time, for example, by learning the map and thus having more fun on it.

We use the EP-FuzzDA algorithm for map selection as discussed in Section 2.4.1. Given users $u \in U$ (in our case, players selected for a match), items $i \in I$ (available maps in a server), relevance scores $r_{u,c} \in R$ (player map preferences), number of items $k \in \mathbb{N}$ (number of maps) and user's weights $v_u; \sum_{\forall u \in U} v_u = 1$ (player's weights) the output is an ordered list of group recommendations L_G of length k (best k maps based on the player map preferences). We now adjust the algorithm to meet our needs.

Firstly, we need to resolve missing player map preferences. Since players are not required but only encouraged to fill every map preference before matchmaking, values can be unavailable. We temporarily assign values based on the average map preference values, the player's average preference values across maps, and average map preference values across players. For players $u \in U$ and maps $i \in I$, missing values $r_{u,i} \in R$ we get R' from R by

$$r'_{u,i} = \begin{cases} r_{u,i} & \text{if } r_{u,i} \text{ is available} \\ \mu + b_u + b_i & \text{otherwise} \end{cases} \tag{3.1}$$

$$b_u = \begin{cases} mean(u) - \mu & \text{if } mean(u) \text{ is available} \\ 0 & \text{otherwise} \end{cases}$$

$$b_i = \begin{cases} mean(i) - \mu & \text{if } mean(i) \text{ is available} \\ 0 & \text{otherwise} \end{cases}$$

$$mean(u) = \sum_{i \in J \subseteq I} r_{u,i}$$

$$mean(i) = \sum_{u \in V \subseteq U} r_{u,i}$$

where μ is an average of values in the matrix R, for a player u is J subset of maps for which $r_{u,i}$ is available and for a map i is V subset of players for which $r_{u,i}$ is available. If a player u has not set any map preference, then $J = \emptyset \implies b_u = 0$; analogically, if players have not set map preferences for a map i, then $V = \emptyset \implies b_i = 0$. When the matrix R has no values, μ cannot be set. In that case, we return random k maps.

We must likewise recalculate map preferences based on recent player match history. For every player, we track recently played maps. Then, we use the

history to reduce map preference values. The more a map was played, the more we reduce the value. For players $u \in U$, maps $i \in I$, relevance scores $r'_{u,i} \in R'$ and count of recently played maps for players $n_{u,i} \in [0, 5]$ we get R'' from R' by

$$r''_{u,i} = \frac{r'_{u,i}}{(n_{u,i} + 1)!} \tag{3.2}$$

Lastly, we must adjust the player's weights to ensure long-term fairness. If the EP-FuzzDA did not meet a player's map preferences in previous matchmaking, i.e., the player map preference of the selected map was suboptimal, then we want to increase the player's chance of getting the map with a higher map preference by increasing the player's weight. We achieve this by tracking player accumulated map preference shares of total map relevance, i.e., what fraction of total item relevance was the user's map preference in previous matches. At the end of a match played on map $i \in I$, for each player $u \in U$ we update acc_u

$$(acc_u)_{updated} = \begin{cases} \frac{r''_{u,i}}{TOT_i} & \text{if } acc_u \text{ undefined} \\ acc_u + \frac{r''_{u,i}}{TOT_i} & \text{otherwise} \end{cases} \tag{3.3}$$

where TOT_i is the total relevance of the map i. acc_u is undefined if the player u has not yet played a match.

However, that is not enough because players can have a different number of matches played. To determine player weights, we introduce a new variable, average share. We calculate the average share simply by dividing the accumulated share by the number of matches played. When a player does not have accumulated share, we assign the mean of the average share of other players with accumulated share defined. Finally, for players $u \in U$, their accumulated share acc_u and number of maps played p_u, we get player weights by normalizing average shares

$$v_u = \frac{avg_u}{\sum_{u \in U} avg_u} \tag{3.4}$$

$$avg_u = \begin{cases} \frac{\sum_{v \in V} acc_v}{|V|} & \text{if } acc_u \text{ undefined} \\ \frac{acc_u}{p_u} & \text{otherwise} \end{cases} \tag{3.5}$$

where V is a subset of players with a defined accumulated share.

In summary, before selecting the best maps using EP-FuzzDA, we edit players' map preferences R by predicting missing values and lowering the preferences of specific maps if players have already played them in recent history. Then, we calculate players' weights based on their accumulated shares of total map relevances in previous matches.

After the match, players can rate the map, which will update their map preference using the arithmetic mean. Thus for player $u \in U$, selected map $i \in I$, player's map preference $r_{u,i} \in R$ and player's rating of the map u after the match a_u:

$$(r_{u,i})_{updated} = \begin{cases} \frac{r_{u,i} + a_u}{2} & \text{if } r_{u,i} \text{ is defined} \\ a_u & \text{otherwise} \end{cases} \tag{3.6}$$

3.4 Rating system

One rating, instead of a private MMR and a public VR, makes it easier for users to understand and track. We also want to select a simple algorithm that does not require complex equations that an ordinary user would need help understanding. As mentioned in Section 2.6, we cannot extract player data from Valorant API. We must rely on player input, so we only evaluate players based on the game results. Therefore, we inspired our calculations from chess rating systems because of its popularity and simplicity.

Furthermore, we will use the Elo rating system instead of the Glicko rating system. Even though the Elo rating system is slightly less accurate over time, it is much easier to understand. Moreover, we can limit rating deviation through time by regularly resetting the player ratings. We chose the USCF rating system, which works for one player when competing against another. In our thesis, a team competes with another team. We calculate a new rating for each player separately and count an opponent's rating as a mean of players' rating. For a team $T \subset U$ and for players $p \in T$ with rating r_p we calculate team rating T_r as

$$T_r = \frac{\sum_{p \in T} r_p}{|T|} \tag{3.7}$$

With this change, teammates with lower ratings are rewarded with more ratings for a win and vice versa.

For calculating K using Formula 2.1, we need the number of total games by a player N' and the number of the player games in the current tournament. If we consider an act as a tournament and reset players' statistics after each act, then $N' = 0$. Because we eliminated N', we have to adjust the formula so K is definable for all m and reasonable for small m. We modify the formula so that K is limited to 80 as in the USCF rating system

$$K' = \frac{800}{10 + m}$$

where N is games played. Thus, the coefficient for the first game (e.g., $m = 0$) is:

$$K' = \frac{800}{10 + 0} = 80$$

We will keep the minimal rating value to 100 and set the initial rating for all players to 1300.

To sum up our modifications, for a player $p \in T \subset U$ with m played matches, opponents $q \in T_o; T_o \cap T = \emptyset$ and game outcome $s_a \in \{0, 1\}$

$$\text{if } (r_p)_{old} \text{ is undefined, then } (r_p)_{old} = 1300$$

$$r_{new} = \min(100, (r_p)_{old} + K' \cdot (s_a - s_e))$$

$$K' = \frac{800}{10 + m}$$

$$s_a = \begin{cases} 1 & \text{if player won} \\ 0 & \text{if player lost} \end{cases}$$

$$s_e = \frac{1}{1 + 10^{(T_o - (r_p)_{old})/400}}$$

$$T_o = \frac{\sum_{q \in T_o} r_q}{n}$$

4. Developer documentation

We introduce implementation for the high-level proposal of the thesis described in Chapter 3. The additional purpose of this documentation is to help readers understand the structure and the code for modifications and additions for their purpose.

The main part of the thesis is implementing the Discord Bot client in JavaScript using a Node.js module, Discord.js. We define slash commands and the Bot behavior after user interaction. We provide methods to be executed after specific events are emitted. All available slash commands are listed in Appendix A.1 and handled event emission in Appendix A.2. The matchmaking logic is separated into controllers. However, the client works only with the Manager interface, as shown in Figure 4.1. Data are stored in a database and retrieved only when users need them. After that, we update the database data if needed. In memory, we only store matchmaking settings (e.g., available maps), player data in matchmaking, and frequently used Discord IDs of channels and roles.

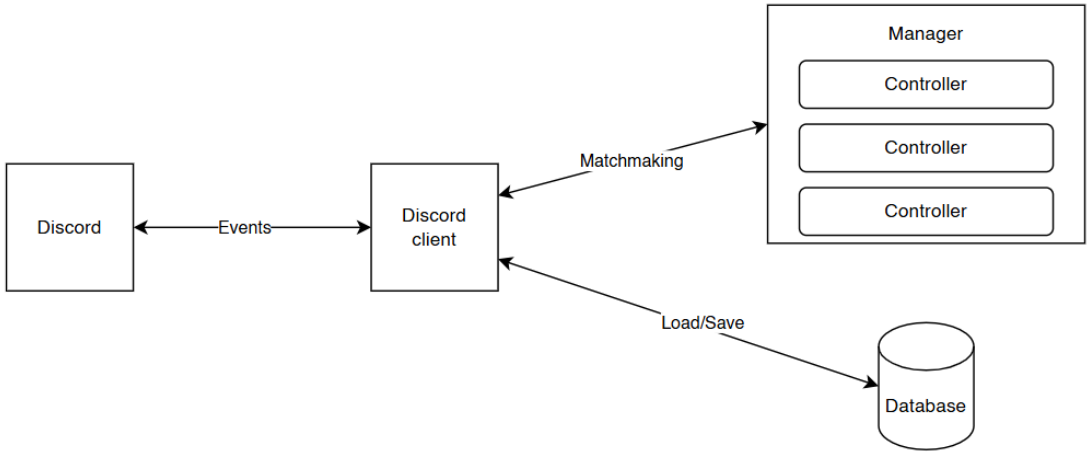

Figure 4.1: Discord Bot architecture

The project is hosted on Render[1] as a Web Service and the code is published in GitHub repository.

4.1 Discord Bot

In this section, we describe the design of our Discord Bot, the project structure, and data handling. We follow the rules and patterns set by the Discord.js guidelines[2].

Discord.js and Discord API refer to a Discord server with the term *guild* in the code and documentation. We also use it in this chapter to avoid confusion. Note that Discord is developing a new feature, Guilds[3] which is currently an

[1] `https://render.com/`
[2] `https://discordjs.guide/`
[3] `https://support.discord.com/hc/en-us/articles/23187611406999-Guilds-FAQ`

experiment. Nevertheless, the experimental feature is not directly related to the term.

4.1.1 Discord Developer Portal

Discord Bot has to be registered in the Discord Developer Portal[4] to be available for users. In the portal, we set the Bot's properties like name, description, and permissions, which the Bot requires from Discord. The portal also generates the application ID, secret token, and the Bot's installation link, which are all important for developers' and users' needs.

We want to request permissions only for the actions that the Bot needs. We do not want to require more or even request Administrator permissions[5], because that will make the server owner fearful of undesirable actions. Permissions are self-explanatory thanks to their names: Ban Members, Manage Channels, Manager Roles, Move Members, Send Messages, View Audit Log (history of administrative actions), and View Channels.

Lastly, in the Bot settings, we give permissions for Presence and Guild Members' intents to be able to listen for events regarding user presence and guild members.

4.1.2 Project structure

The Discord.js guide[6] helped us create the project and the code structure. In this section, we describe the required files for proper module functionality.

Package file In **package.json** specifies project information (e.g., name, description, license) and developer information, like which scripts to run, project dependencies, and developer dependencies.

Environment file stores project secrets, e.g., keys and tokens, with the `dotenv` package, which loads secrets from the **.env** file. **.env** should not be committed or viewable by anyone.

In this thesis `.env` file contains these secrets:

```
# SQLiteCloud secrets
SQLITECLOUD_API_KEY=
SQLITECLOUD_CONNECTION_STRING=

# Discord Bot secrets
DISCORD_CLIENT_ID=
DISCORD_TOKEN=
```

[4] `https://discord.com/developers/applications/`

[5] Administrator permission grants a Bot every permissions and allows the Bot to bypass all other permissions

[6] `https://discordjs.guide/`

Main file of the project is **./index.js**. We create a Discord client, instance of `Client`[7], defining intents using `GatewayIntentBits` enumeration[8]. Our Bot needs these intents:

- `Guilds` - role and channel management
- `GuildMembers` - guild members management
- `GuildVoiceStates` - voice state update tracking
- `GuildPresences` - presence update tracking

Then, we create a client for a database *SQLiteCloud*, which we later discuss in Section 4.2. We initialize an instance of `MatchmakingManager` for operating with player data and handling matchmaking in guilds. We analyze its structure in 4.1.5.

These created instances we pass down to event handlers described in Section 4.1.4. Finally, we log in the Discord client to Discord using the Discord Developer Bot token. Now, the client is ready to receive emitted events and interact with Discord resources.

Source directory We store custom modules (e.g., database, matchmaking, managers, and controllers) in **.src/** source directory. The module export methods are used in the Discord Bot client. The source directory also provides constants and utility functions to be used globally in **./src/constants.js** and **./src/util.js**. We expect developers to modify or replace mainly matchmaking modules (map selection, player rating, player selection). The replacement module has to implement exported methods with the same input and output parameters.

4.1.3 Commands

We keep each slash command in a separate file and export it as a module containing `data`, an instance of `SlashCommandBuilder` with which we describe the command, e.g., name, description, arguments. The module contains an execute function that uses the command's logic. When a user submits a slash command, an `interactionCreate` event emits. When handling the event, we call the corresponding slash command to execute the function. More about event handling can be found in section 4.1.4.

The function is always `async`[9] because every slash command requires an asynchronous response. In each command file we define `InteractionReplyOptions`[10] for the response. We decided not to put them together in one response file because the they are unique due to their specificity and, therefore, are not reusable in other commands in most cases. Moreover, storing all responses in one file would make the file very cluttered.

[7] `https://discord.js.org/docs/packages/discord.js/14.14.1/Client:Class`

[8] `https://discord-api-types.dev/api/discord-api-types-v10/enum/GatewayIntentBits`

[9] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function`

[10] `https://discord.js.org/docs/packages/discord.js/main/InteractionReplyOptions:Interface`

We modeled slash execute functions to have these parameters: `interaction` (information about the user interaction of type `Interaction`[11]), `args` (array of additional arguments), `sqlClient` (database client), `matchmakingManager` (instance of `MatchmakingManager`). Developers can decide to add additional arguments for resolving the slash command through parameter `args` which we demonstrate below.

If developers want to create a new slash command, we prepare a slash command file template **./commands/example.js**. For clarity, we recommend naming the file by the command name. When naming commands, we use a convention where we replace spaces with a dash. If a response is not directed at members other than the sender, we add the `ephemeral` flag to avoid overwhelm other members with command responses.

In addition to slash commands, we also define custom commands. They are called after the user interacts with message components[12], e.g., buttons and select menus. Note that not all message component interactions result in calling custom commands. However, custom command code structure and behavior are identical to slash commands, except `data` property is an object and not `SlashCommandBuilder` instance.

We set `customId`, a message components property, to determine how to handle the interaction:

- `command_${command-name}` to call a slash command execute function
- `${custom-command-name}` to call a custom command execute function

If we need to add additional arguments to the function, we append them to the `customId` using an underscore. For example, after an user interacts with a message component, we want to call a slash command `example` with additional arguments `a`, `b` and `c`, then the `customId` would look like this `command_example_a_b_c`.

The commands are registered to the Discord client in **./index.js**. We iterate through files **./commands**, **./custom-commands** directories and add them to and register commands to the client. We iterate through **./events** and register them in the client to listen.

User interaction

When a user interacts with the Bot, an event `interactionCreate` is emitted with a `BaseInteraction` argument. When handling this event, we distinguish two types of supported interactions in this thesis: slash command interaction and message component interaction. If the user submits a slash command, we call the corresponding async execute function of the slash command. If the user clicks on a message component, we investigate the argument's `customId`, which we defined in Section 4.1.3. Then, we call the corresponding slash or custom command with arguments from the `customId`.

[11] `https://discord.js.org/docs/packages/discord.js/14.14.1/Interaction:TypeAlias`

[12] `https://discord.js.org/docs/packages/discord.js/14.14.1/MessageComponent:TypeAlias`

We allow users to interact with the Bot only on the server because the Bot works for each server separately. Therefore, if the user interacts with the Bot outside servers (e.g., in direct messages), we do not know to which matchmaking server we should assign the user. The only exception is when the user is in the queue and changes status to offline. In this case, the Bot dequeues the user and sends a direct message informing the user about the action with a message component allowing the player to enqueue back. In this scenario, we include the server information in the `customId`.

Registering commands

After implementing the commands we register them to the client, we iterate through **./commands**, **./custom-commands** directories and register commands to the client. We iterate through **./events** and register them in the client to listen.

4.1.4 Emitted events

Discord emits all sorts of events, but we register only those we need with the Discord client. We differentiate two types of events: those that should be called once and the rest. This section lists and describes our usage of every emitted event. The event handler methods are stored in separate modules in **./events/** directory. The module exports `name`, which is one of the names in `Events`[13] enum. The exported event handler method is designed to have these parameters: parameters given by the emitted event `args`, Discord client `client`, database client `sqlClient` and matchmaking manager `matchmakingManager`.

We have mentioned events we are handling in Section 3.1, and we extensively describe them in Appendix A.2.

4.1.5 Data handling

We retrieve data with a lazy approach, e.g., we will retrieve data from the database only when they are needed at the moment. The Bot stores only some player data, i.e., Discord ID, games lost, games won, and rating. We opt not to store map preferences because the values and available maps can change throughout matchmaking (whereas the player data listed above do not change or only change once after a match). It is easier to manage data with this approach because the data in the database and the data in the Bot's memory are identical. Moreover, we want to limit memory usage due to the online hosting of the Bot, and this solution is also resistant to outages of Discord and the Bot and database hosting.

The matchmaking manager contains controllers storing guild, player, lobby, and match data. Matchmaking data are separated based on the player's current matchmaking status:

- `PlayersInQueue` - array of player data in queue

[13] `https://discord.js.org/docs/packages/discord.js/14.14.1/Events:Enum`

- `LobbyVoiceChannels` - dictionary of lobbies information separated by lobby ID
- `OngoingMatches` - dictionary of games data separated by game ID
- `GuildIds` - frequently used guild information (e.g. admin roles ID)

We use a Facade pattern to cover the controllers using a class `GuildManager`. The Facade pattern connects controllers into one class for one united front. It also enables the creation of additional methods in which controllers work together. By unitizing the controllers, we also increase extensibility. Suppose developers want to add a new controller. In that case, the developer does not have to add the new controller to functions where needed (possibly to every command execute function). Instead, with a Facade pattern, adding the controller to the `GuildManager` class is necessary. The downside of the Facade pattern is that the `GuildManager` combines methods of all controllers. We have carefully named methods to be clear and non-ambiguous. Discord Bot works on multiple works simultaneously, and we keep their data separate. Therefore, we must add another layer above `GuildManager`. We call the layer `MatchmakingManager`, and it will keep track of the current servers that the Bot is in, and the layer will call the corresponding `GuildManager`.

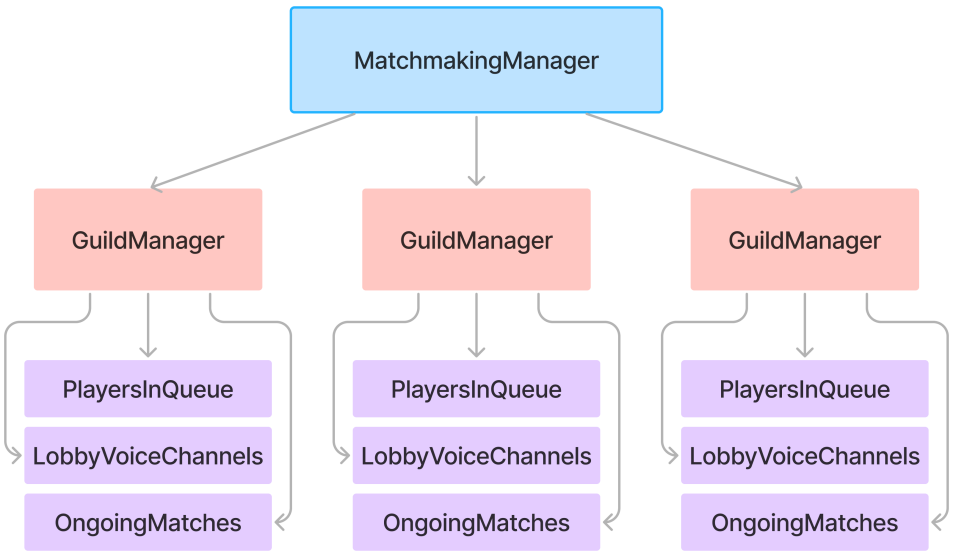

Figure 4.2: Matchmaking manager hierarchy

4.1.6 Functionality obstructions

As mentioned in Section 3.1.4, roles have a linear hierarchy, which means that the Bot cannot create a role that is higher than the position of the Bot itself. That means that at least the server owner can modify the Bot's role or

admin roles, which can impact their functionality. For this reason, on Bot's startup, e.g. on an event `clientReady` emission, on role update, e.g. on an event `guildMemberUpdate` emission, or on joining server (upon inviting to the server, the owner can refuse to give needed permissions, on an event `guilcCreate` emission, we check if all the needed Bot's permission listed in Section 4.1.1 are still available. If not, the Bot sends a direct message to the server owner about the situation and requests those permissions. Since admin roles do not require special permission, we only warn and check if roles are mentionable.

Furthermore, we listen for `guildRoleDelete` events and check if admin roles were deleted. In such cases, we create new admin roles and inform the server owner about the situation in direct messages. We do not check for the Bot's role deletion because the role cannot be deleted.

Lastly, we must prevent the Bot's channels (e.g., a game category with temporary lobby channels) from being deleted. That can be done by setting up permissions when creating them. We deny everyone except the Bot to manage these channels.

4.2 Data storing

The player data are split into three tables: player map preferences in `MapPreferences`, player map history in `MapHistory`, and player Discord data and leaderboard data in `Players`. Map preferences are stored separately because the map pool can frequently change and is easier to manipulate than storing map preference in `Players` table. Map history is also separate for flexibility of map history length. In our case, we keep track of the last five maps, but it can make sense to keep track of different lengths for different systems. Furthermore, in our case, keeping track of match details in the database is redundant since we are not manipulating that data other than keeping each player's score. However, the database architecture is constructed for future expansion.

Lastly, we store important IDs of Discord objects (e.g., role ID, channel ID) in `GuildIds` and available maps in `Maps`.

We chose SQLite because of its lightweight and file format. Since matchmaking in different servers is separate, we also want to keep data separate but with the same database structure. We keep each database/server in different files using the SQLite approach. The database is hosted on SQLiteCloud[14]. Its free tier meets all our requirements and does not limit the usage of our Bot.

We set the camel case convention for table names and the snake case convention for table columns.

4.3 Express REST API

REST API is designed for users to access data from a private database. The API is using Express[15] web framework. Users can retrieve data about players and their standing in Discord servers.

[14] `https://sqlitecloud.io/`

[15] `https://expressjs.com/`

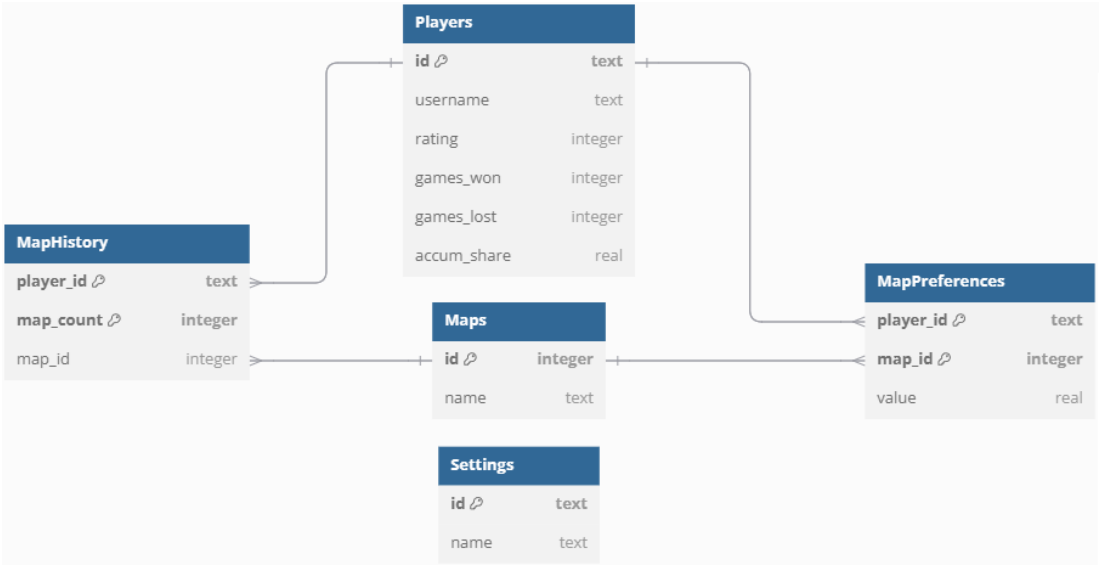

Figure 4.3: Database diagram

While defining routes depicted in Figure 4.4, we consider possible features, like match history. The routes are also quite robust due to eliminating redundant database accesses. For example, if we eliminated an intermediate route `/server/`, users could try routes like `/matchmaking/`, which would result in a database search for a server named *matchmaking*.

Leaderboard and player data are provided in three forms: without any clarification, is data represented as a website. Otherwise, users can retrieve data as short text or JSON by extending the route by `text/` or `json/`. The short text format was created for displaying data without requiring users to manipulate the data, e.g., in Twitch commands described in Section 5.5.2.

While designing routes, we consider possible route expansion with new possible features, like match history. We can add new routes to our REST API routes located in `express/express.js`, depicted in 4.4

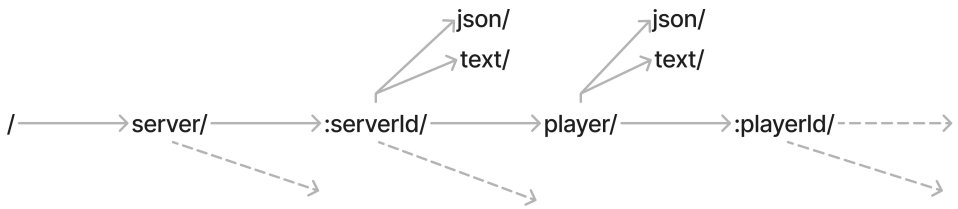

Figure 4.4: Current (full lines) routes and future (dashed lines routes of `https://valojs.onrender.com/`

We multipurpose the REST API to also serve as a website where players

can locate their data. The data from the database are rendered into Pug's[16] templates. Websites that do not require data from the database or any other data are viewed as static files.

[16] `https://pugjs.org/api/getting-started.html`

5. User documentation

In this chapter we show how to correctly work with our Bot. The source code is available in a GitHub repository and hosted on Render for the Bot to be added to servers.

5.1 Adding Discord Bot to your server

You can add a Discord Bot to your server using this permanent invitation link

`https://discord.com/oauth2/authorize?client_id=1322612015474147429`

Discord will asks you in which server you want to add the Bot and requests needed permissions. Make sure that you are aware of all the permissions the Bot needs for proper functionality.

5.1.1 Discord Bot on invite actions

After the Bot joins the server, it will create two channel categories: informational read-only category `VALOJS` with `#general` (for updates), `#match-history` (logging matches played) and `#reports` (tracking reported players) channels. Furthermore the Bot adds `VALOJS-GAMES-1` category to create temporary text and voice channels for matches. Since a channel category can hold up to 50 channels, the Bot will create more categories based on the match workload. The Bot will also add two roles to your server: `ValoJS SuperAdmin` and `ValoJS Admin`.

5.2 Admin roles and permissions

`ValoJS SuperAdmin` will be assigned to the server owner and will be able to do everything `ValoJS Admin` can do and add or remove `ValoJS Admin` roles to everyone on the server. Both roles position are below the Bot role position.

Note: We recommend moving `ValoJS SuperAdmin` role to the highest role position since it should be assigned only to the server owner or members with the highest permissions. Change `ValoJS Admin` role position accordingly since it has high impact commands, e.g. banning members.

Members with the `ValoJS Admin` role can ban players, which will kick the member and delete all records of the player from the leaderboard. They can also cancel lobbies and matches or decide match results.

Note: Since the admin slash command permissions are given to members with admin roles, members with roles with higher

positions in your server can also add or remove admin roles to themselves or others.

Admins can reset act which will reset player matchmaking data, but not their map preferences. Admins have to provide list of maps that will be played in the following act.

5.3 Slash commands

You can find a list of all available commands and their functionality in Appendix A.1. You can also list all available commands on Discord using the Bot command `/help`.

There are two types of slash commands: basic commands that anyone can execute and Admin commands, which are properly executable only with ValoJS roles.

5.3.1 Visibility of Admin commands

Admin text commands can be properly executed only with admin roles, i.e., `ValoJS SuperAdmin` and `ValoJS Admin`. Members without these roles can see and enter admin text commands, but the Bot will prevent them from executing their primary function.

Hide Admin commands

The Bot cannot hide admin text commands from members without admin roles, but you can do it with these steps (as shown in Figure 5.1):

1. Go to your server settings.
2. Locate `APPS` section and click on `Integrations`
3. In section `BOTS AND APPS` click on `ValoJS Bot`
4. In `COMMANDS}` section click on an admin command
5. Click on `Add Roles or Members` and choose if they can see it or not

Note: With these steps, you can also limit in which text channels users can use Bot's text commands, but almost every text commands (except text commands related to a match which can be properly executable in the match text channel) and their Bots replies are *ephemeral*, i.e., only visible for the user that sent the command.

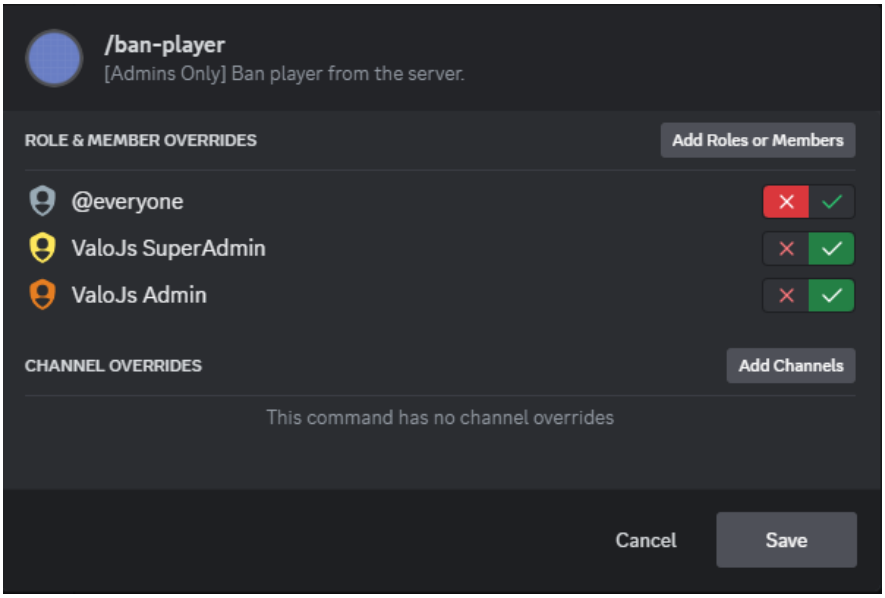

Figure 5.1: Example of overriding Discord commands

5.4 Getting into a match

In this section, we describe how to play with our matchmaking Bot. The process is also illustrated in Figure 5.2.

1. Simply write anywhere command `/queue`. The Discord bot will check if you already filled out your map preferences. If any of your map preferences are missing, then you will have the option to fill them out.

2. Bot will select the first ten players in the queue and start a lobby. The lobby has text and voice channels in the `ValoJS-GAMES` section. The Bot will create a text and voice channel to mention you and other players.

3. The Bot will choose the top three maps based on the map preferences of all selected players. You can vote and then join the voice channel tagged in the text channel.

4. The Bot will request a randomly selected player to create a custom match lobby and send a party code into the text channel for others to join.

5. After everyone joins the voice channel, the Bot will announce the map based on votes, divide players into two balanced teams, and move them to separate voice channels. Team sides are selected randomly.

6. The Bot will send a result message where any player after the match submits the game's result. Any player from the opposing team has to confirm the result.

7. The Bot announces players' rating updates based on the game outcome and sets a timer, after which all created channels for this match will be deleted.

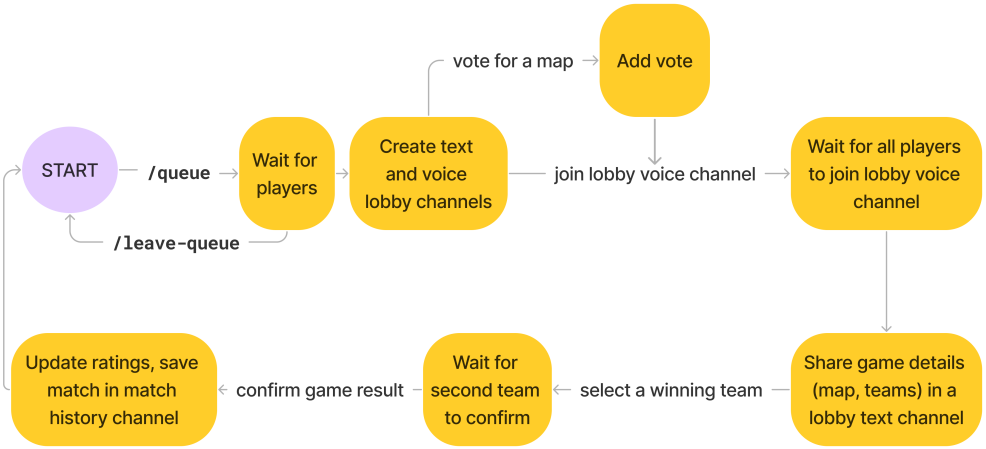

Figure 5.2: Example of a matchmaking loop from user perspective

5.5 Retrieve data using API

You can access your data outside the Discord server using the provided API. Anyone can access the data without any permission needed. You need to obtain the server and player username to locate the data. If you activate developer mode, you can find these IDs by right-clicking on the server or user profile and selecting `Copy Server ID` from the menu.

5.5.1 How to enable developer mode

1. Go to your user settings.
2. Locate `APP SETTINGS` section and click on `Advanced`.
3. Toggle on `Developer mode`.

5.5.2 How to request data

You can request leaderboard data containing each player's standings, rating, and match record on the server.

Leaderboard

Server leaderboards can be retrieved in many forms. However, the main request URL is:

`https://valojs.onrender.com/server/DISCORD_SERVER_ID`

This will display data as a website. You can add to the end of the link `json` for `JSON` output or `/text` for short text format.

Player data

For player data, you need both server ID and player username:

`https://valojs.onrender.com/server/DISCORD_SERVER_ID/player/PLAYER_ID`

Note: Do not mistake the Discord username for a display name. Each user's username is a unique Discord name. Meanwhile, the display name can be duplicated. In your user profile, your username is located below your display name. Changing your username will not impact your data, which are saved under your Discord ID and cannot be changed.

Usage

You can use the requested data, for example, for your stream. In the tutorial below, we show you the steps for setting up `!leaderboard` (depicted in Figure 5.3) command for Twitch using Fossabot[1]

1. Authorize Fossabot to you Twitch channel using this guide.
2. Go to your custom commands. Log in with your Twitch account.
3. Click on `+CREATE`, in `Name` write `leaderboard` and in `Response` write

```
$(eval $(urlfetch https://valojs.onrender.com/server/DISCORD_SERVER_ID/text))
```

You can set up `!stats` command similarly, and it will look like this in your Twitch chat:

Figure 5.3: Example of custom Twitch commands with data from API

5.6 Advanced options

If you are missing any features, you can edit the Bot and host it yourself. You can find everything you need in the Developer documentation chapter.

[1] `https://fossabot.com/`

Conclusion

We analyzed issues with official in-game Valorant matchmaking and set our goals for our third-party matchmaking app. We mainly focused on the rating system and map selection.

We discussed possible hosts for our app and landed on Discord. We designed and created a Discord Bot. We created a supporting AWS database and API. We hosted our Bot on Render and registered the hosted Bot on the Discord Developer Portal.

Finally, we created developer documentation for users to modify or enhance our Bot to meet their needs. We also created user documentation where users can learn how to add and use our hosted Bot on their servers.

Future work

We analyzed all the matchmaking issues and focused on the most impactful ones, in our case, the rating system and map selection. For larger-scale matchmaking bots, we recommend extending research on player selection from queue to match. With enough players, it is also possible to extend player selection based on role.

Our Bot has the most essential functionalities and commands. However, it can be extended to these characteristics:

- The project is developed with a low player base in mind. If the Bot is added to a server with many members, then a more sophisticated player selection can be implemented without drastically impacting the queue time.
- Players using our Bot can only queue up as a solo player. This can be extended to party queuing, which enables players to queue together, ensuring players play together in a team.
- Honoring system can improve our Bot. After a match, a player could honor a friendly player. The honor system then rewards players for good behavior.
- The Bot can also be extended to other Valorant in-game modes, and a separate rating system can be added for those, where players can compete against each other to be the best. In contrast, Valorant provides visible rank only for the competitive mode.
- The Express REST API can be separated from the Bot, providing a more crash-prune solution.

A. Appendix

In this appendix, we list available slash commands defined in the Discord Bot and all the events that the Bot handles.

A.1 User commands

We list available slash commands for the Bot. Everyone can submit admin commands, but their primary function is executed only for members with admin roles.

/add-admin SuperAdmins only. Gives a member a `ValoJS` admin role.

/ban-player Admins only. Bans a player from the server and removes the player's matchmaking data.

/cancel-lobby Admins only. Cancels an ongoing lobby. It will not impact player matchmaking data.

/cancel-match Admins only. Cancels a match. It will not impact player matchmaking data.

/change-maps-preferences Lists available maps, and players can change their map preference.

/example Example command for developers.

/help Lists all the available commands.

/leaderboard Show all players sorted based on their rating.

/leave-queue Removes a player from a matchmaking queue.

/maps Prints available maps.

/queue Adds a player to the queue.

/report-player Logs player in the report channel.

/reset-act Admins only. Resets player matchmaking data and sets a new list of available maps.

/set-match-result Admins only. Sets the result of an ongoing match.

/stats Prints player matchmaking data and map preferences.

/sub-player Replaces player in a lobby with another player.

A.2 Emitted events handling

We list events handled by the Bot and when they are emitted.

clientReady is emitted once the Bot is ready to work.

guildCreate is emitted when the Bot joins the guild.

guildDelete is emitted when the Bot is kicked from the guild or the guild is deleted.

guildMemberRemove is emitted when a member is forcefully removed from a member or left voluntarily.

presenceUpdate is emitted when a member changes presence.

voiceStateUpdate is emitted when a member changes voice state presence.

interactionCreate is emitted when the interaction is created.

guildMemberUpdate is emitted when a member changes appearance.

roleUpdate is emitted when a role is updated.

roleDelete is emitted when a role is deleted.

Bibliography

[2(2024)] DOTA 2. The new frontiers, 2024. URL https://www.dota2.com/newfrontiers.

[Apples(2024)] KC Apples. I'd like to announce that riot put their trust in me to run champions queue for worlds 2024 and the following preseason! we will be using our discord from before so if you're an accredited erl player and didn't get invited yet feel free to message me. expect games in 1 week., 2024. URL https://x.com/ApplesloI/status/1835340739619819640.

[Berg(2020)] Arthur Berg. Statistical analysis of the elo rating system in chess. *Chance*, 33(3):31–38, 2020.

[Chamberlain(2020)] Paul "Arkem" Chamberlain. Valorant anti-cheat: Cheater, reported!, 2020. URL https://playvalorant.com/en-gb/news/dev/valorant-anti-cheat-cheater-reported/.

[Chang(2022)] Brian "Riot South Korea" Chang. Valorant systems health series - map diversity, 2022. URL https://playvalorant.com/en-gb/news/dev/valorant-systems-health-series-map-diversity/.

[Citron(2020)] Jason Citron. Your place to talk, 2020. URL https://discord.com/blog/your-place-to-talk.

[Dev(2021)] Dev. Ask valorant - rank rating edition, 2021. URL https://playvalorant.com/en-us/news/dev/ask-valorant-rank-rating-edition/.

[Dickinson(2023)] Hunter Dickinson. Guilded vs discord: Which platform is better for your community?, 2023. URL https://whop.com/blog/guilded-vs-discord/.

[EA(2024)] EA. How apex legends™ranked works, 2024. URL https://help.ea.com/en/help/apex-legends/apex-legends/apex-legends-ranked-mode/.

[Elo(1986)] A.E. Elo. *The Rating of Chessplayers, Past and Present*. Second edition. ARCO PUBLISHING. INC., New York, 1986. ISBN 0-668-04721-6.

[Gafford(2024)] Travis Gafford. Inero speaks on the death of na's champion's queue, 2024. URL https://www.youtube.com/watch?v=vqmk0d_RPgE.

[Glickman(1995)] Mark E Glickman. The glicko system. *Boston University*, 16 (8):9, 1995.

[Glickman(1999)] Mark E Glickman. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 48(3):377–394, 1999.

[Gortok(2018)] Draggles & Socrates. Gortok, RiotIAmWalrus. /dev: Matchmaking real talk, 2018. URL https://nexus.leagueoflegends.com/en-us/2018/02/dev-matchmaking-real-talk/.

[Malecek and Peska(2021)] Ladislav Malecek and Ladislav Peska. Fairness-preserving group recommendations with user weighting. In *Adjunct Proceedings of the 29th ACM Conference on User Modeling, Adaptation and Personalization*, pages 4–9, 2021.

[Mark E. Glickman(2020)] Thomas Doan Mark E. Glickman. The us chess rating system, 2020. URL https://new.uschess.org/sites/default/files/media/documents/us_chess_rating_system_specs-2024-03-01.pdf.

[Masthoff(2010)] Judith Masthoff. Group recommender systems: Combining individual models. In *Recommender systems handbook*, pages 677–702. Springer, 2010.

[meowzers(2023)] meowzers. How rank rating (rr) is calculated, 2023. URL https://support-valorant.riotgames.com/hc/en-us/articles/20113235347347-How-Rank-Rating-RR-Is-Calculated.

[Patel(2024)] Nilay Patel. Discord ceo jason citron makes the case for a smaller, more private internet, 2024. URL https://www.theverge.com/24134914/discord-ceo-gaming-chat-teens-safety-moderation-decoder-interview.

[Raphael(2024)] Raphael. Faceit beginners guide, 2024. URL https://support.faceit.com/hc/en-us/articles/14996562458268-FACEIT-Beginners-Guide.

[Riot Phroxzon(2021)] Riot Sotere Riot Phroxzon, RiotIAmWalrus. Ask riot: Where's rod of ages?, 2021. URL https://www.leagueoflegends.com/en-us/news/dev/ask-riot-where-s-rod-of-ages/.

[Robles(2024)] Albert Robles. State of the maps - july 2024, 2024. URL https://playvalorant.com/en-gb/news/dev/valorant-systems-health-series-map-diversity/.

[Sara "Riot necrotix" Dadafshar(2022)] Brian "Riot South Korea" Chang Sara "Riot necrotix" Dadafshar. Valorant systems health series - voice and chat toxicity, 2022. URL https://playvalorant.com/en-us/news/dev/valorant-systems-health-series-voice-and-chat-toxicity/.

[Sara "Riot necrotix" Dadafshar(2021)] Lea "Riot aeneia" Hughes Sara "Riot necrotix" Dadafshar. Valorant behavior detection and penalty updates, 2021. URL https://playvalorant.com/en-us/news/dev/valorant-systems-health-series-voice-and-chat-toxicity/.

[Takahashi(2024)] Dean Takahashi. Discord ceo sheds light on future of gamer communication as users cross 200m, 2024. URL https://venturebeat.com/games/discord-ceo-sheds-light-on-future-of-gamer-communication-as-users-cross-200m/.

[Tracker(2024)] Valorant Tracker. Global agent usage and stats - valorant tracker, 2024. URL https://tracker.gg/valorant/insights/agents.

[VALORANT(2021)] VALORANT. Duality // official lore cinematic - valorant, 2021. URL https://www.youtube.com/watch?v=-M_r8MKQ3mo.

[VALORANT(2024)] VALORANT. Beginner's guide, 2024. URL https://playvalorant.com/en-us/news/announcements/beginners-guide/.

[VLR(2024)] VLR. Trace esports vs. edward gaming: Valorant champions 2024: Playoffs: Valorant match, 2024. URL https://www.vlr.gg/378823/trace-esports-vs-edward-gaming-valorant-champions-2024-ubqf/?game=180417.

[whatacoolwitch(2023)] whatacoolwitch. Valorant - competitive mode faq, 2023. URL https://support-valorant.riotgames.com/hc/en-us/articles/360047937633-VALORANT-Competitive-Mode-FAQ.

[Wilde(2023)] Tyler Wilde. Apex legends pro hits top rank without scoring a single kill: 'that is just so dumb', 2023. URL https://www.pcgamer.com/apex-legends-player-hits-top-rank-without-scoring-a-kill-i-mean-that-is-just-so-dumb/.

List of Figures

List of Tables