Jonathan Wiseman, Ian Bowler, Matthew Grohotolski, Ryan Ehmann
CS 341
April 6, 2020

Looking for Jays (LFJ)

https://github.com/jonwiseman/LFJ

1. MOTIVATION

1.1. Background

1.1.1. Discord

Discord is a popular messaging and communication application catering primarily to the gaming community. Its use is widespread, with about 963,000,000 messages exchanged daily between users [1]. Discord operates primarily on a server-channel model: servers bring together large groups of people, with channels serving specific interests, topics, or purposes (such as voice or video chat).

In addition to human users, Discord also allows programmable "bots" to operate in servers, serving a range of functions. A bot's function can be as simple as posting images or replying to users when summoned, or as complex as performing managerial and organizational roles for the whole server.

1.1.2. Elizabethtown's E-Sports Team

This wide applicability immediately suggests an application for Elizabethtown's own E-Sports team, which primarily uses Discord for communication, event scheduling, and scrimmage signups. The team's primary focus is on League of Legends (LoL), an online, team-based multiplayer game with a robust ranking ranking system. In addition to LoL, the team has a desire to expand to additional games, including Rocket League (RL) and Counter Strike: Global Offensive (CSGO). Each of these games operates on separate clients: LoL has its own client, owned and operated by Riot Games [7]; RL and CSGO are both hosted on the Steam games client [8].

It is out of this context that we propose *Looking for Jays (LFJ)*, a Discord bot primarily focused on helping Elizabethtown's E-Sports team plan and organize their scrimmage events.

1.2. Goals

There are five primary goals for LFJ, motivated by both the team's personal needs and additional features that we believe would be useful:

1. Handling event signup
2. Balancing scrimmage teams
3. Posting reminders for registered events
4. Allowing members to self-assign to groups
5. Relevant statistics collection and querying

1.2.1. Handling Event Signup

The team needs to be able to create and manage upcoming events such as meetings, practices, and scrimmages. This ensures clear communication between members, and provides transparency to the whole group. LFJ will allow users to register for events by a simple voting mechanic, such as reacting to a message posted by the bot to the whole server; in this case, a reaction is a form of explicit feedback indicating attendance. Alternatively, LFJ might allow users to register for events using a simple command.

The primary measure of success for this goal is the ability of the bot to record responses and properly remind users of their commitments. This itself is contained in goal number 3, wherein the bot is required to remind users of upcoming events for which they have registered. If all users are consistently reminded of their events, then this goal will be considered met.

1.2.2. Balancing Scrimmage Teams

After a game event has been scheduled, it will be necessary to assign users to teams. For each of the three games with proposed coverage, this includes splitting the users into two evenly balanced teams. LFJ will assign users into teams based on skill level and, perhaps eventually, past performance.

The primary measure of success for this goal is user satisfaction. There is no easy way to assess how evenly balanced the assigned teams were. With sufficient data collection, it may be possible to assign a degree of fairness based on the outcome: a longer match with a more even score implies more evenly balanced teams. As such, simple surveys after each game can provide explicit feedback on LFJ's team balancing system.

1.2.3. Posting Reminders for Registered Events

After signups and team assignments are complete, users should be reminded of their commitments. This can be implemented as a simple message to each user that has registered for an event, with messages being sent out at customized intervals. LFJ will remind users of their commitments to attend events through a simple messaging service.

This goal's measures of success are consistency and accuracy. LFJ works as intended if every user that has registered for an event is sent a reminder exactly when they are supposed to be sent.

1.2.4. Allowing Members to Self-Assign to Groups

Each member of the team has specific games that they are interested in playing, and others that they do not want to play. Event sign ups should be limited to players to whom the events are relevant; this helps with team balancing and also reduces the amount of information that will need to be stored about each team member. LFJ will allow users to self-assign to groups that determine which events they receive notifications about and can participate in.

As a measure of success for this goal, members should be able to query the bot to determine their status in certain groups. For example, a team member might ask the bot to list which groups he belongs in. Secondly, team members should only be able to sign up for events related to the

games to which they belong; a user not authorized to sign up for an event should get an error message. Finally, newly created events should send notifications to everyone that belongs to the relevant group.

1.2.5. Relevant Statistics Collection and Querying

LFJ will keep track of important statistics for games, players, and groups for access control and to improve team balancing. The exact method of collection is currently not determined: it may be possible that an administrator has to manually enter statistics after each game, or users will be prompted to respond to a survey. The primary motivation of this feature is to move beyond team balancing by explicitly labeled skill level, possibly by creating a statistical or machine learning model to predict if a team is balanced or not. The nature of the statistics collected is also currently unknown: each game has statistics that are important to game outcome, and feature selection requires domain knowledge. For a discussion of the data structures used to store and retrieve this information, consult the Design section below.

Devising a measure of success for this goal is difficult, due primarily to the short-term nature of this project. Measuring satisfaction with collected statistics and their impact on team balancing will take time and team participation, which cannot be guaranteed. As such, we consider the use of statistics on matchmaking to be a reach goal. However, the access control aspect of stat collection is detailed above for goal number 4.

1.3. Existing Solutions

There are two primary existing solutions for the problems facing the Elizabethtown College E-Sports team:

1. The current solution: manual curation within the Discord app
2. Pre-made Discord bots

1.3.1. Manual Curation within Discord

This is the current approach utilized by the E-Sports team. Currently, a session is planned with a simple message that contacts everyone on the Discord server. As explicit feedback, users react to the message with emojis. This works well with informal events, as it is simple and easy to understand. However, it falls short of many of the desired goals stated by the E-Sports team: there is no event registration, no reminders about upcoming events, and no selectivity in who receives notifications. Perhaps most importantly, there is no team balancing for scrimmages; instead, this is handled on a case-by-case basis within the team.

This approach is clearly functional, but it does not take advantage of the automation and record-keeping potential of Discord's bots. Here, LFJ would represent an improvement by reducing human error and automating many functions such as registration, reminders, and record-keeping.

1.3.2. Existing Discord Bots

There are a number of existing Discord bots that cover a few of the stated requirements for this project. Below is a discussion of each of the goals above, and potential Discord bots that could satisfy them:

1. Handling event signup: Pollmaster [4], VoteBot [3], Simple Poll [10]
2. Balancing scrimmage teams: VoxLight [9]
3. Reminders for Events: DisCal [2]
4. Self-Assignment to Groups: none
5. Statistics collection and querying: none

Handling event signup is perhaps the goal that has the most outside coverage by existing Discord bots. However, these bots fail to integrate with other steps in the pipeline that LFJ covers. For example, the calendar application DisCal requires integration with Google Calendar, requiring a manual input step between event registration and notification sending; additionally, it fails to send notifications only to those users to whom the event is relevant. Finally, there are no bots that currently handle user self-assignment or statistic collection. The implementation of these backend services is unique to LFJ, and represents a significant departure from existing Discord bots.

2. APPROACH

2.1. Architecture

LFJ follows the Model-View-Controller architecture pattern. In this case, the model is the bot's back end: the databases and scripts that control the bot's behavior and contain the data that allow it to function; the view is Discord, the platform that displays the bot's results (such as messages); the controller is the Discord API that allows users to send and receive messages from the bot.

Looking for Jays is conceived of as a Python-implemented Discord bot, coded via the discord.py wrapper. Discord.py is a Python API wrapper for Discord's developer API. The documentation for discord.py is available in [6]; the documentation for Discord's official API is available in [5]. LFJ is conceptualized in three stages: user, bot, and backend. Figure 1 displays the interaction flow between users, the bot, and the backend:

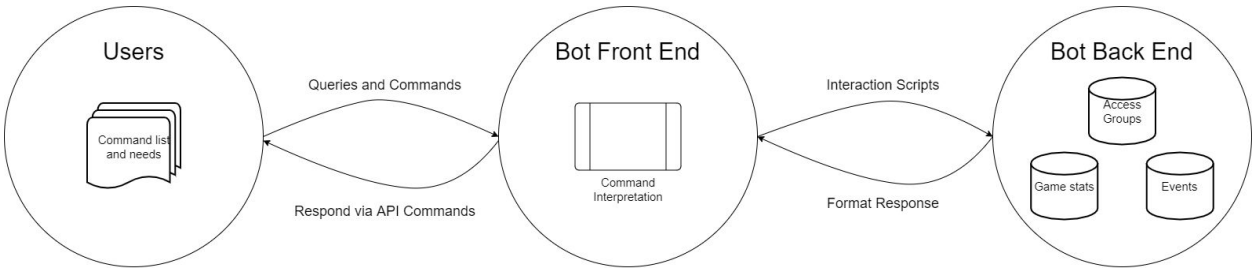

Figure 1: LFJ Overview

2.1.1. Users

Users constitute the members of the Discord server (the members of the E-Sports team, in this case). Users have clearly defined needs, listed in the Goals subsection above. Users are allowed to interact with the bot's front end in two ways: via simple text commands and via comment reactions.

Text commands are structured in a way that can be parsed by the bot. As of now, all commands must start with $COMMAND. The bot will only be able to respond to pre-defined text commands. Text commands will include querying the bot for user group status and updating user group status.

Reactions are explicit feedback to a comment posted by the bot. Many of the voting bots discussed in the Existing Discord Bots subsection above are operated in this fashion. We will follow a similar implementation, with reactions corresponding to specific voting choices. It is not yet certain if voting will be strictly binary (attending/not attending) or multi-class (such as voting for games to play or dates to schedule). Additionally, users can register for events with the user of simple commands.

2.1.2. Bot Front End

The bot's front end receives interaction from users and formats them for use in the bot's scripts. The bot's front end receives and transmits messages via the official Discord API, as monitored by the bot's back end scripts. Once the required information has been accessed and formatted by the bot's back end, it will be sent to users via the Discord API. Commands are processed and arguments are passed using the discord.py commands extension.

2.1.3. Bot Back End

The bot's back end consists of the bot's running scripts and database. The back end is interacted with primarily through MySQL queries, managed by the MySQL Python connector. Finally, the requested information will be formatted and sent via the bot's front end to relevant users.

2.2. Hardware Requirements

In order for the bot to run continuously and receive user commands, it will need to be hosted on an active server. This server will require constant internet access and power in order to provide reliable service.

Depending on the data storage implementation, it may also be necessary to run a database server in parallel with the bot. However, the exact nature of the storage is still being defined. There are two options: a relational database that incorporates each of the use cases above, or simple storage of facts outside of a database.

As of implementation, the bot and associated MySQL database are hosted on the same server.

2.3. Software Requirements

There are currently two software requirements for building and running LFJ:

1. An installation of MySQL to run the backend
2. The LFJ bot token

The MySQL installation is required to run the bot's backend, while the LFJ bot token is required to run the bot. Please note that the primary LFJ bot token is private; for users to download, build, and run the bot they should provide their own token.

In addition to the software requirements listed above, the running directory of an LFJ installation should have a configuration file called configuration.conf. The configuration file contains important database and Discord login information that allows the bot to function properly; its structure is as follows:

```
[Discord]
token =
events_channel =
prefix =
[Database]
username =
password =
host =
database =
```

2.4. Software Design

Figure 2 gives an overview of the system's use cases:

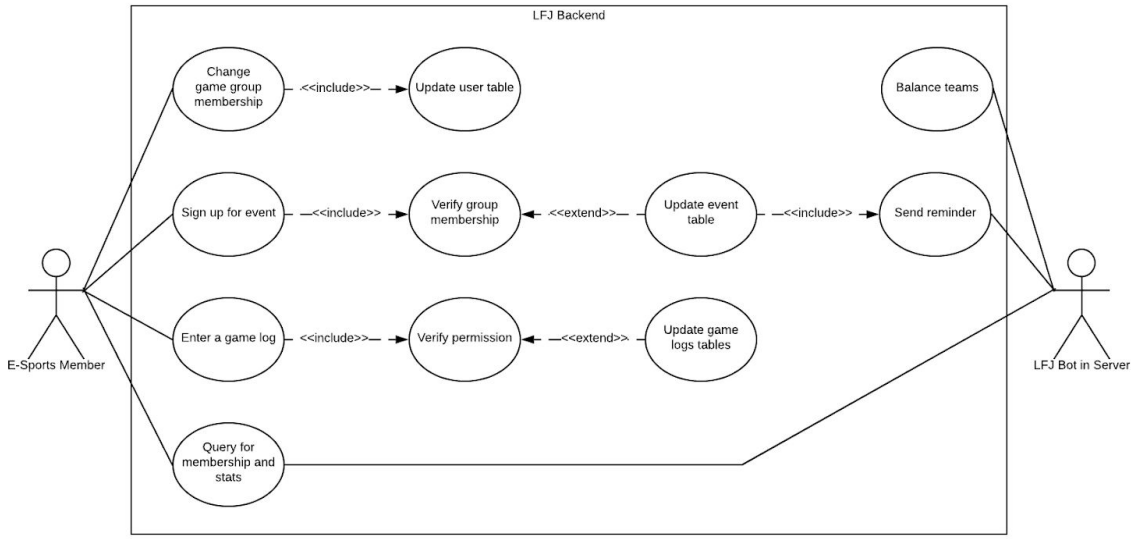

Figure 2: LFJ Use Case Diagram

There are two actors in the system: the E-Sports team member is the primary actor, initiating contact with the LFJ bot via text commands within the Discord server; the LFJ Bot is the secondary actor, responding to users in the Discord server. An E-Sports team member is able to:

1. Change game group membership
2. Sign up for an event
3. Enter a game log
4. Query for membership and stats

When a team member updates his game membership status (i.e. enters that he plays CSGO), then the user table in the database will be updated. When a user attempts to sign up for an event, the system first checks to make sure that he is a member of the relevant group. If he does, then the event table will be updated; the LFJ bot can then send reminders to the users in the server. When a user attempts to enter a game log, the bot will check to see if he has permission; if he does, then the game logs in the database will be updated. Finally, the user can query the bot for membership and stats, which will be displayed by the bot in the server.

The LFJ bot in the server can interact with users via messages. These messages can display the teams that have been created, remind users of their registered events, or be query responses.

2.5. Database Design

Figure 3 displays the bot's backend database design:

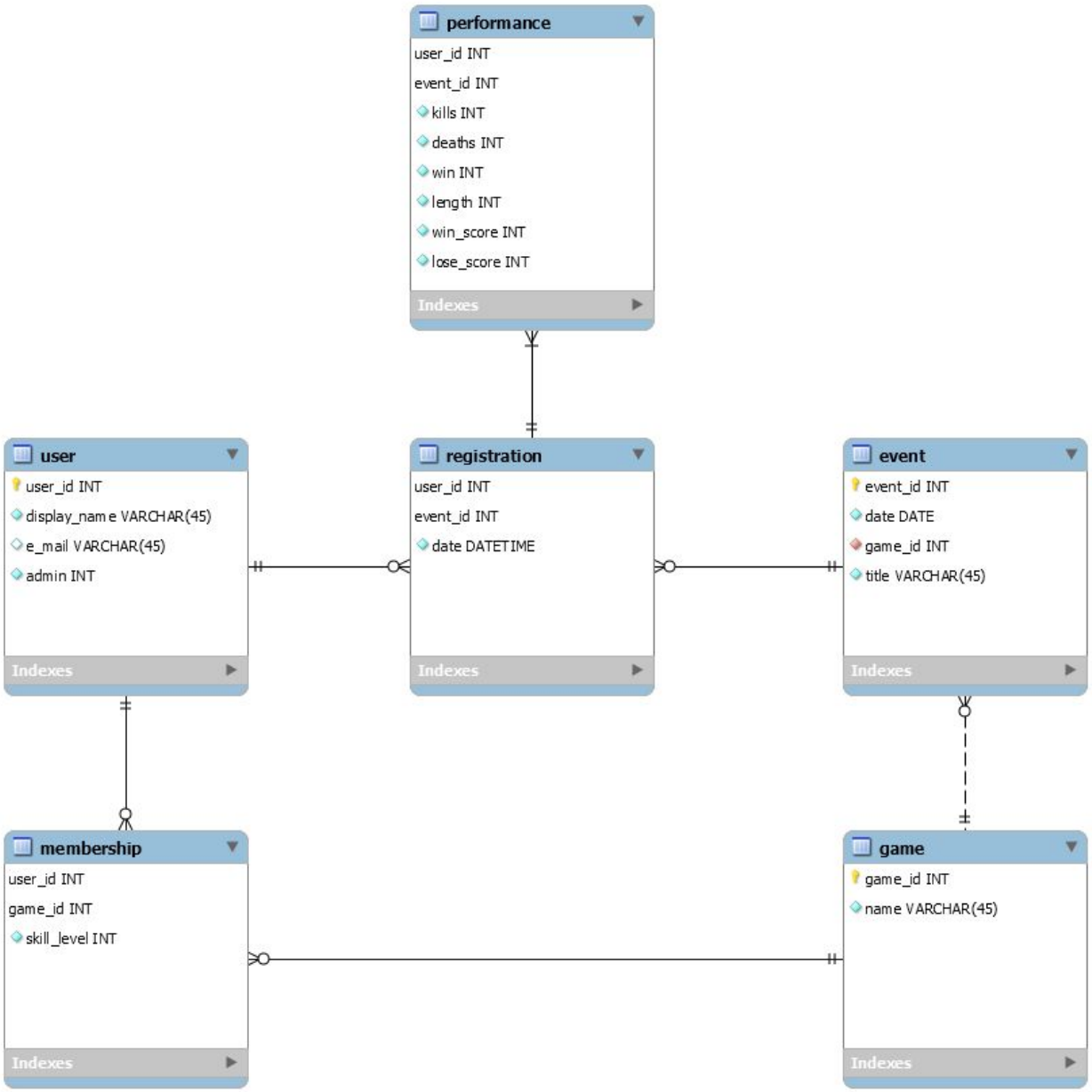

Figure 3: Bot Backend Database

The business rules for the database are as follows:

- A user registers for zero or more events; an event consists of zero or more users.
    - This n:n relationship necessitates a bridge entity: registration
- A user plays zero or more games; a game is played by zero or more users
    - This n:n relationship necessitates a bridge entity: membership
- An event is about exactly one game; each game may be played in zero or more events
- Each registration produces a performance; each performance matches exactly one registration

Below is an entity summary for the database:

- User:
  - user_id: the user's unique integer identifier
  - display_name: the user's Discord display name
  - e-mail: an (optional) way to contact the user
  - admin: 0 for no, 1 for yes; admins can change the database
- Event:
  - event_id: the event's unique integer identifier
  - game_id: the ID of the game being played in this event
  - date: the date of the event
  - title: name of the event
- Game:
  - game_id: the game's unique integer identifier
  - name: the game's title (ex. League of Legends)
- Membership:
  - user_id: the ID of the user that plays the game
  - game_id: the ID of the game that the user plays
  - skill_level: a measure of the user's skill at the specified game (used for matchmaking)
- Registration:
  - user_id: the ID of the user making the registration
  - event_id: the ID of the event being registered for
  - date: date of the event being registered for
- Performance:
  - user_id: the ID of the user who competed
  - event_id: the ID of the event that was competed in
  - kills: measure of positive performance
  - deaths: measure of negative performance
  - win: did the player win this event?
  - length: length of the match
  - win_score: score of the winning team
  - lose_score: score of the losing team

3. CHALLENGES AND RISKS

3.1. Challenges

There are three primary challenges involved in the creation of LFJ:

1. Acquiring domain knowledge of Discord and specific games
2. Learning Discord's API and associated Python wrapper
3. Data collection and storage

Acquiring domain knowledge is not particularly challenging, but is time consuming. There are a few things that we need to know about Discord in order to implement LFJ, such as: how bots are able to see reactions to messages, what the API limits for bots are, and how quickly interactions with the bot can be processed. Knowledge of specific games (such as LoL, RL, and CSGO) will

be necessary for statistical collection and team balancing. The exact nature of the statistics we will collect will have to be determined after deciding which features are important for each game.

Perhaps the biggest part of this project will be learning Discord's API structure and associated Python wrapper. The documentation for discord.py will be a good starting point, and there are a number of publicly available articles and examples to view and use as starting points.

Finally, collecting and storing data will be a major challenge of LFJ. Design of an appropriate database or utilization of fast and secure data structures for storing user, game, and event data is what will drive the bot's function. The first step will be to determine what form this data storage will take.

3.2. Risks

In addition to the general design challenges listed above, there are specific problems and risks associated with each goal. The subsections below deal with the risks facing each goal.

3.2.1. Handling Event Signup

There are two related risks associated with the bot's ability to handle event signups: user privacy and group voting. Both of these issues are related in that the voting actions of users may have influence on the voting actions of other users. For example, some users may be less likely to sign up for an event if nobody else has signed up yet; alternatively, popular events may draw lots of votes without real user intent to attend.

3.2.2. Balancing Scrimmage Teams

Balancing scrimmage teams suffers primarily from a lack of method and data. Users will have to self-ascribe their skill levels in order to achieve a simple balancing method. There is also very little way to measure the success of this goal, since it will depend on long-term satisfaction (explicit feedback) or analysis of game statistics (implicit feedback such as game length and final score).

3.2.3. Posting Reminders for Scheduled Events

There are two major risks for this goal: relying on users to check their Discord notifications and users potentially changing their votes after the polling period ends. It is entirely possible that event notifications will be completely ignored by users, leading to wasted effort on our part. Finally, it is possible that users will withdraw their attendance to an event or want to join after the polling period is finished. Sending notifications to users who have withdrawn will be unnecessary, and users who want to sign up later may not receive any notifications about the event.

3.2.4. Allowing Members to Self-Assign to Groups

The risks relating to this goal are primarily related to user access control. The largest risk is with users not being able to sign up for events due to not having permission. An additional risk is

users changing their status after signing up for an event and subsequently not getting relevant notifications. On one hand, it would be nice for users to be able to change their group status with a simple command; on the other, they may do this too often and break the backend database. Navigating these access control risks will have to be met with robust controls on which commands users are allowed to execute.

3.2.5. Relevant Statistics Collection and Querying

In addition to the challenge of relevant data collection, this goal also carries the long-term risk of usefulness. While we want to gather statistics to improve our matchmaking capabilities long-term, it may be possible that the statistics we choose to gather are irrelevant. Implementing this feature is obviously time consuming and design-intensive, so we must invest time up-front in background research and proper design.

3.3. Solutions

For each of the risks above, the subsections below propose preliminary design solutions.

3.3.1. Handling Event Signup

The risks associated with this goal are primarily related to privacy and ensuring independent voting. In order to mitigate the risks associated with visible voting, we propose hiding vote totals until the end of the voting period. This would ensure that users vote independently and in their personal interests.

There are a few drawbacks to this solution. First, this prevents users from seeing when other users are available; this lowers team cooperation and planning. Secondly, this requires a fixed voting period, the rigidity of which leaves no room for changes in users' personal schedules. Finally, this approach may actually lower user engagement by way of reducing cooperative planning.

3.3.2. Balancing Scrimmage Teams

In order to implement team balancing, users will have to self-ascribe their skill levels when enrolling in new groups. There are a few design considerations to consider here: what is the scale that will be used for self grading? Will users assign numeric ratings or categorical ratings for their skill? Should different games have different scales? Should we incorporate some user-supplied statistics, such as total time played for a specific game or some performance metric? Answering these design questions as we progress will help enhance matchmaking and mitigate the risks associated with its implementation.

3.3.3. Posting Reminders for Scheduled Events

There are two potential enhancements for getting notifications to users in a timely and accurate manner. First, the bot may have to repeatedly query its voting messages in order to get accurate vote tallies. Secondly, it may be possible for the bot to produce visualizations of events (such as a calendar or graphic showing user engagement) to increase user interaction.

3.3.4. Allowing Members to Self-Assign to Groups

Our primary design solution for access control risks is the introduction of a channel moderator with authority to approve user queries that change backend data. Some user commands and queries will be able to execute without moderator approval, such as querying group membership or event status. However, some user commands should be approved by the moderator, such as changing group status. The primary drawback of this approach is the potential bottleneck this creates; however, the small nature of the E-Sports team currently limits the bottleneck's impact.

3.3.5. Relevant Statistics Collection and Querying

These are perhaps the most difficult risks to mitigate, since they involve long-term planning and execution. Seeing the use of these statistics long-term will require dedicated collection by team members. However, we may be able to implement some automation that makes statistical collection easier and more streamlined. This may include post-event surveys to randomly chosen users to capture their measure of the game's fairness.

4. TENTATIVE TIMELINE

4.1. Development Model

For the completion of the LFJ project, we have chosen a scrum model. The scrum roles are as follows:

- Product owner: Elizabethtown E-Sports Team
- Scrum master: Ryan Ehmann (since he is a member of the E-Sports team)
- Development team: Ian Bowler, Jonathan Wiseman, Matthew Grohotolski

Figure 2: Sprint Overview

The division of work is as follows:

- Jonathan: database design, command interfacing, reminder implementation
- Ryan: team balancing, command interfacing
- Matthew: database implementation, user role assigning
- Ian: server setup, command interfacing

4.2. Timeline

Sprint 1: February 10 - February 24

- Register bot, create Discord channel for testing, host bot on server
- Design databases and determine which stats to collect for each game
- Acquire background knowledge (games and discord.py)

There are two primary goals for this sprint: to have a bot that we can interact with via simple commands and to have databases designed. The bot's responses to commands do not have to be meaningful, with us focusing instead on receiving and responding to events via the discord.py library.

Sprint 2: February 24 - March 9

- Users can join/specify which games they play (goes into database)
- Users can enter stats via bot queries (with admin approval)
- Administrator role that bot recognizes for changing/updating databases
- Admin enters game results into databases (through bot queries)

This sprint focuses on implementing the bot's ability to respond to text commands, as well as having implemented our database. The data in the database does not need to be accurate, real, or meaningful but should represent common use cases and allow feature testing.

Sprint 3: March 9 - March 23

- Implement event signups
- Implement basic matchmaking technique
- Possibly acquire real-world data

This sprint focuses on the two most important project goals: event signups and matchmaking. This is also the time to possibly start collecting real-world data, since our data storage will have been implemented.

Sprint 4: March 23 - April 6

- Code cleanup
- Testing (including giving it to the team for use)
- Debugging, bug fixes
- Presentation and report

This sprint focuses on completing testing, resolving bugs, and creating final presentations and reports.

5. Implementation

5.1. Build Automation

We have chosen to use Buddy as our automated build system. It is currently integrated with our Github repository, but as of now does not actually perform any real function. Our biggest goal with build automation is testing: unit testing on each command/SQL function on each new push into the repository to maintain a stable build.

5.2. Code Review

In addition to reviews on branch merge requests, we have elected to use Codefactor for automated code review. Our current Codefactor score is an A-, and the report is available in our Github repository.

5.3. Implemented Commands

The timeline above was a first approximation to the implementation process. However, the actual implementation of LFJ began about a week and a half behind the tentative schedule;

additionally, the tentative schedule did not account for breaks or the disruption in face-to-face meetings due to COVID-19. Nonetheless, there are currently eighteen available commands in LFJ, organized according to function.

User Queries

These are queries that allow interaction with the user table:

1. add_user
2. delete_user
3. query_user
4. set_email
5. set_admin

Game Queries

These are queries that allow interaction with the game table:

6. add_game
7. delete_game
8. query_game
9. set_game_name

Event Queries

These are queries that allow interaction with the event and registration tables:

10. create_event
11. delete_event
12. get_events
13. query_event
14. create_registration
15. delete_registration

Membership Queries

These are queries that allow interaction with the membership table:

16. create_membership
17. delete_membership

Miscellaneous Commands

17. help
18. Exit

Documentation of each command and its syntax is available at the Github repository.

5.4. State of the Project

As of April 6, goals 1 and 4 are completely met: users can create and register for events using the commands above. Additionally, goal 3 is almost complete, with users able to register for events and the database backend keeping track of users who register for specific events.

Goals 2 and 5 will be completed in the next 3 weeks. Team balancing is possible with implementation of membership commands and registration commands, and should not take particularly long.

5.5. Implementation Details

Here is an overview of each script in our repository:

1. Database/lfj.sql: a SQL script that creates the LFJ backend database and initializes it with one user: jon_wiseman#8494 as an admin
2. Scripts/bot_controller.py: the primary run script for the bot. Adds cogs (defined in the scripts below), parses the configuration file, and runs the bot
3. Scripts/helper_commands.py: contains command definition for the exit command, implementation of the helper function "check_admin_status," and errors relating to admin permissions
4. Scripts/user_queries.py: contains command definition of user commands (see above), functional definitions of SQL commands, and errors relating to the user table
5. Scripts/game_queries.py: command definitions of game commands, SQl commands, and errors relating to the game table
6. Scripts/event-queries.py: same as above but for events and registrations
7. Scripts/init_db.py: a helper tool for initializing the LFJ backend when the bot is not running

6. REFERENCES

1. Discord Inc. Company. (February 9, 2020). Retrieved February 9, 2020 from https://discordapp.com/company.

2. DreamExposure. DisCal-Discord-Bot. (August 6, 2019). Retrieved February 9,2020 from https://github.com/DreamExposure/DisCal-Discord-Bot

3. ForYaSee. voteBot. (October 17,2019). Retrieved February 9, 2020 from https://github.com/Votebot/voteBot

4. Matnad. Pollmaster. (September 1, 2019). Retrieved February 9, 2020 from https://github.com/matnad/pollmaster

5. Muddyfish. Discord API. (February 8, 2020). Retrieved February 9, 2020 from https://github.com/discordapp/discord-api-docs

6. Rapptz. Discord.py. (February 2, 2020). Retrieved February 9, 2020 from https://github.com/Rapptz/discord.py

7. Riot Games. Our Story. (2017). Retrieved February 9, 2020 from https://www.riotgames.com/en/who-we-are/values.

8. Valve Corporation. About. (2020). Retrieved February 9, 2020 from https://store.steampowered.com/about/.

9. Vox Light. Discord-Balance-Bot. (May 7, 2018). Retrieved February 9, 2020 from https://github.com/VoxLight/Discord-Balance-Bot

10. Wilhelmklopp. Simple-poll. (May 19, 2016). Retrieved February 9, 2020 from https://github.com/wilhelmklopp/simple-poll

6. FEEDBACK

(2/9/2020) All feedback has been addressed.

Link to original proposal: EtownLFG