[Insert index here!]

Analysis

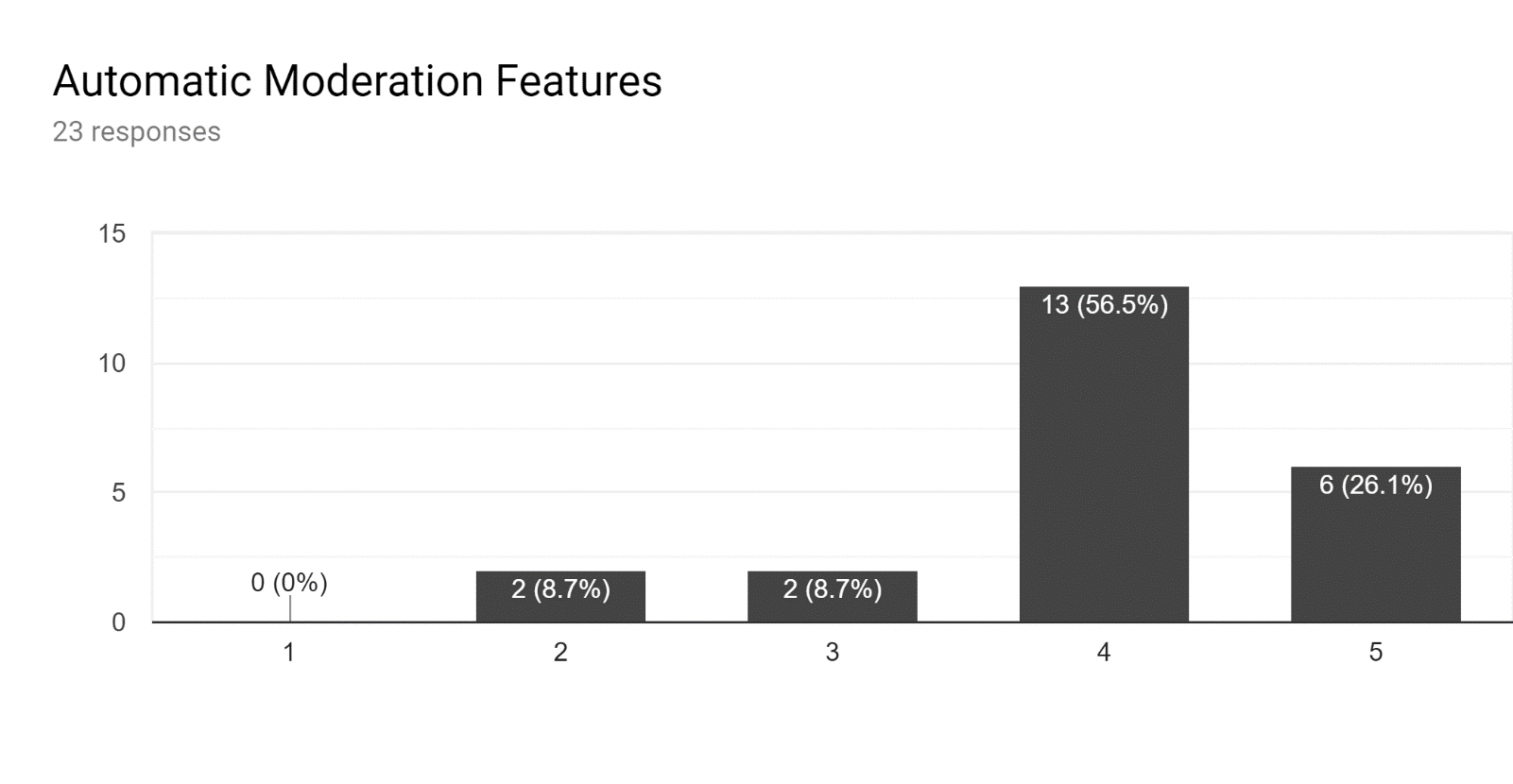
Project Background

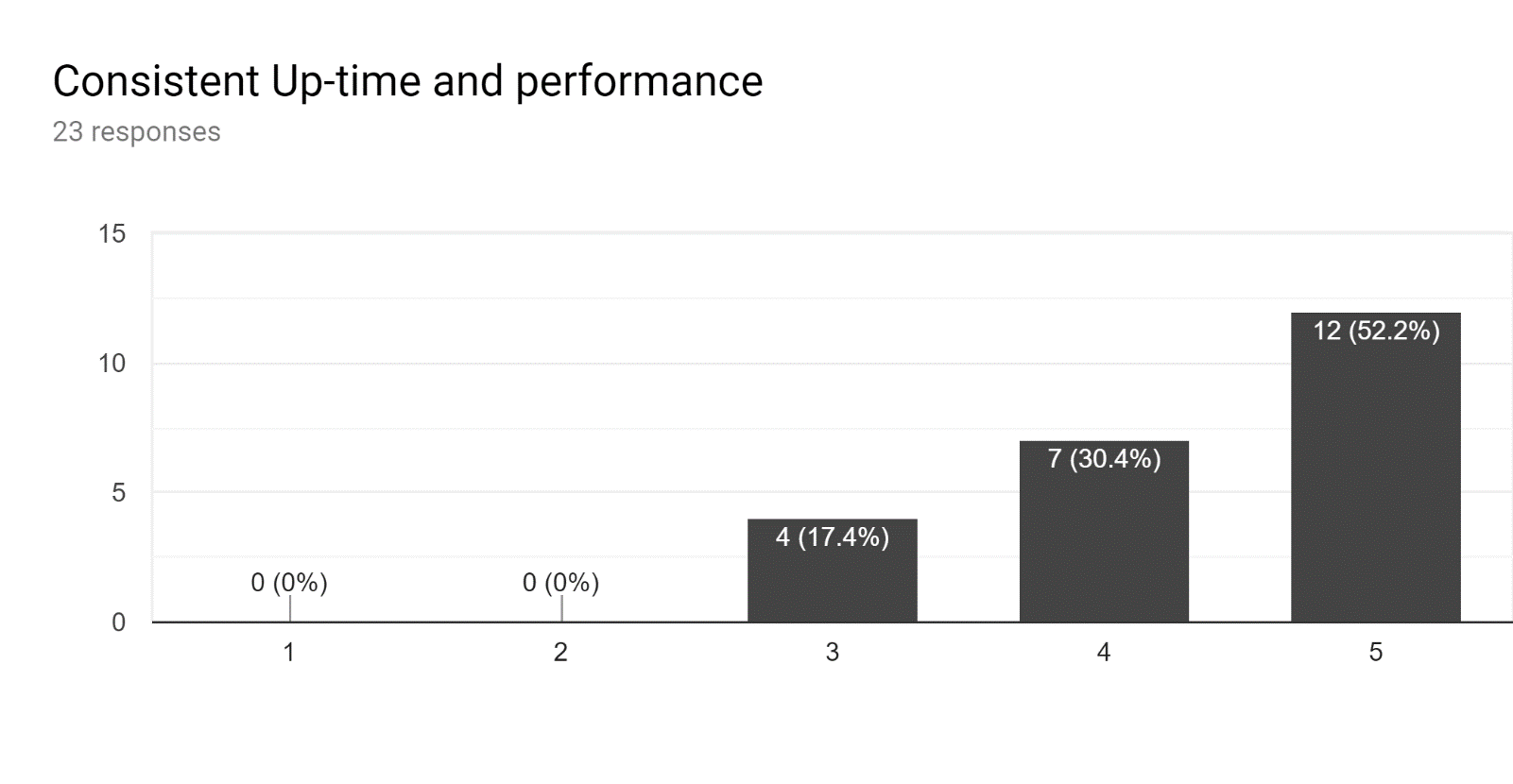
Over the last few years, many people like myself have started using the now very popular Social Media Platform Discord. For those unaware, Discord is a massively popular communication app, that allows for ‘servers’ that contain both text and voice channels. Users can join these servers and are managed by a rather complex permissions system as to fine tune what users can do what where, oftentimes governed by ‘tags’ called roles (more on these later). Due to the large numbers of users, manual moderation quickly became inefficient. In response, discord mad public their REST/WebSocket based API which people used to build projects such as artificial users (‘bots’) coded in various languages. Overlord is one such Bot and seeks to implement features that often require multiple bots into one convenient and powerful package with the aims of being able to be easily selfhosted and modified by end users seeking to add custom functionality. To this end, I will be aiming to package it into a docker container. (see <https://www.docker.com/> for more information.)

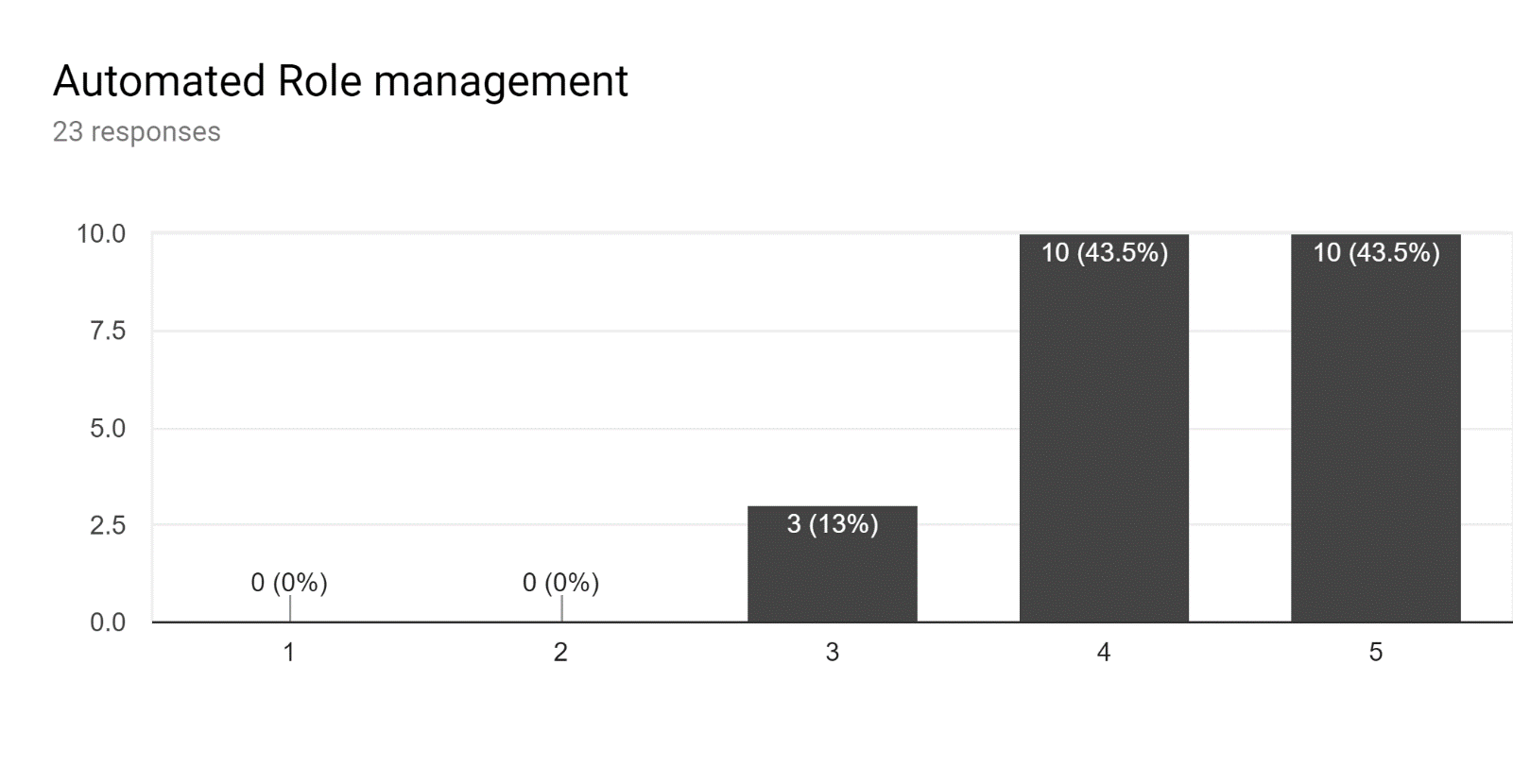
My end users will be both server administrators/moderators and general server Users. To this end, I want to ensure that the management backend is simple enough for first-time set and forget Admins, but also powerful enough for those seeking to add custom functionality, whilst being robust and easy to use for the other server members. It also seeks to differentiate itself by offering a wide range of functionality normally only available through the addition of numerous other bots used on conjunction, whilst being completely open source (All files placed on a public GitHub repository)

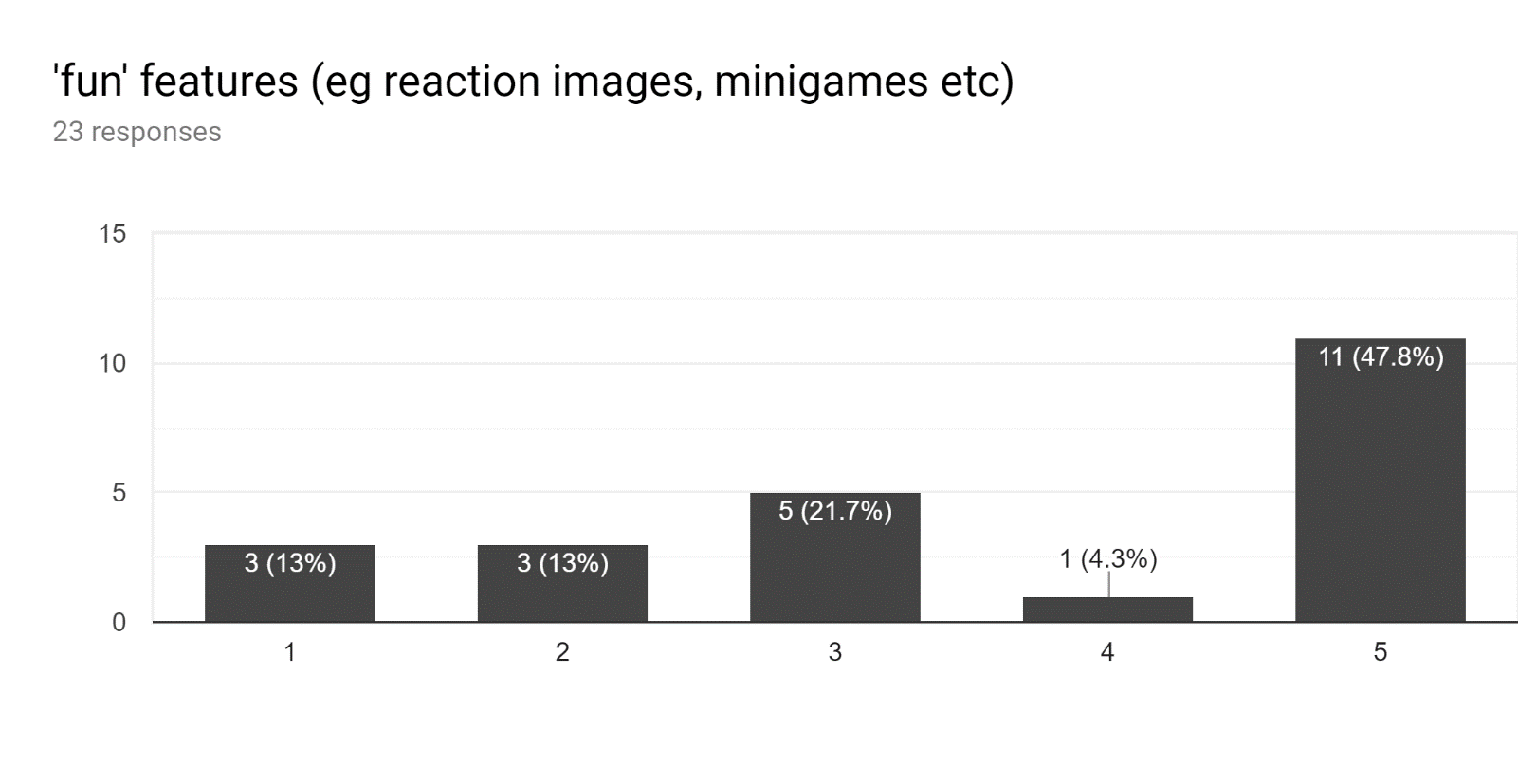
End user Survey

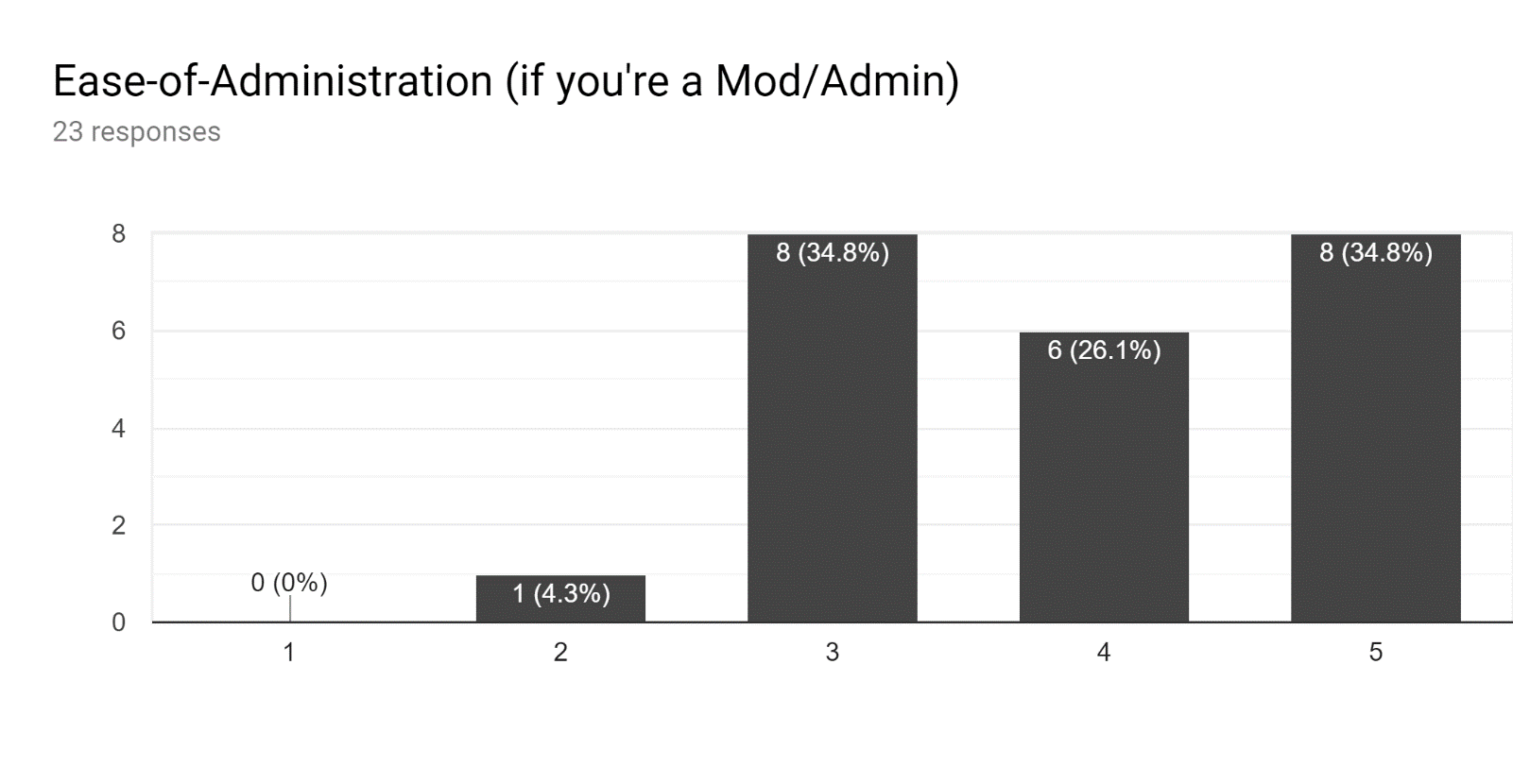
I created and distributed a simple survey on google forms, allowing people who frequently use Discord to answer on a scale of 1-5 (1: least important, 5: most) about what they think are the most important features of a Discord Bot, as well as some open-ended questions for other functionality. This information will be key for establishing my base development priorities. I got a limited number of responses, but the trends were very interesting to examine in regards to my developmental priorities. It showed that potential end users place a higher demand on more simple Features (such as automatic moderation) than more ‘advanced’ features (such as the ability to add custom commands) which is behaviour that I’d expect given the 1:100+ admin : generic user split and the differing priorities of each group. The graphs of the data collected from the surveys is as follows:

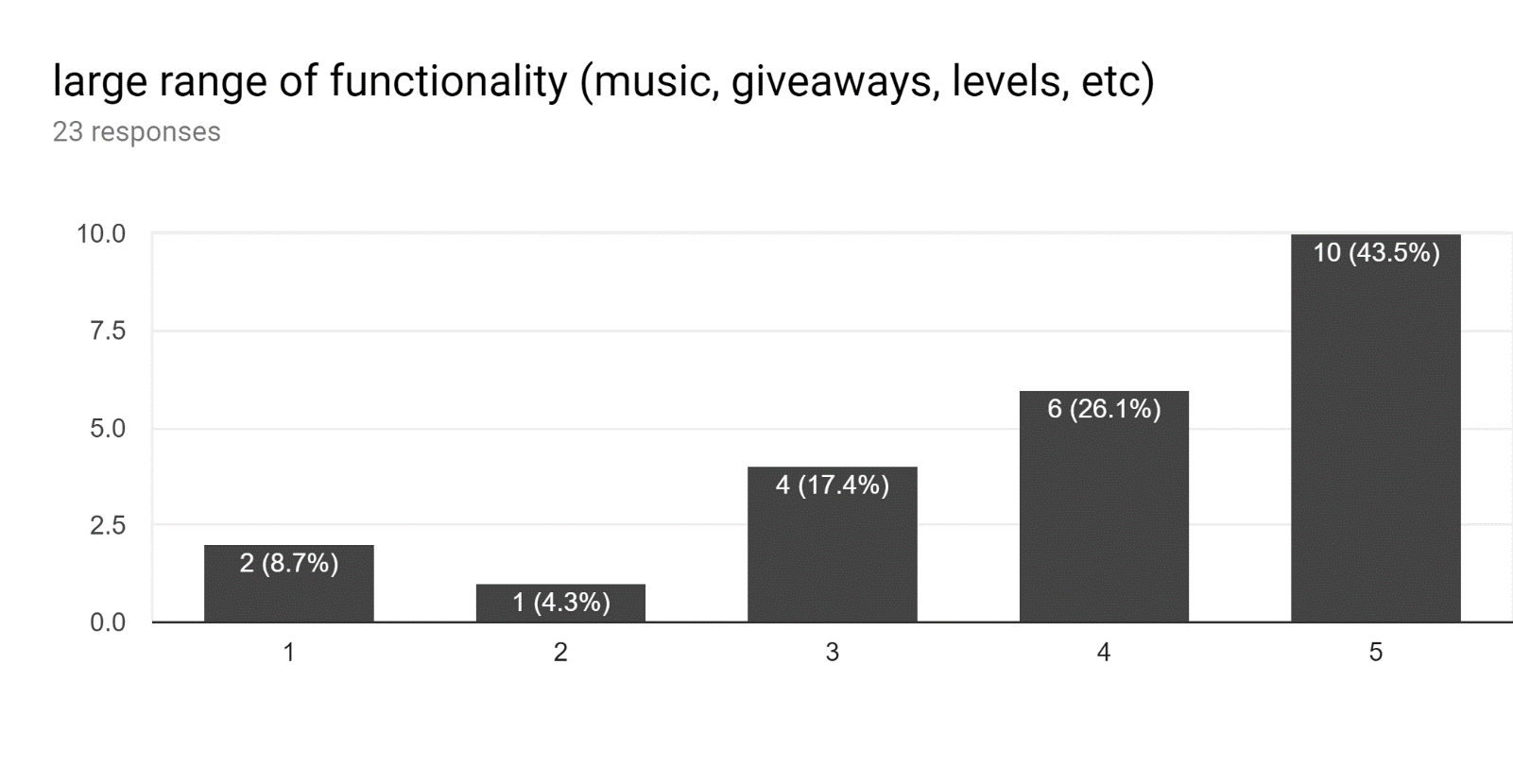


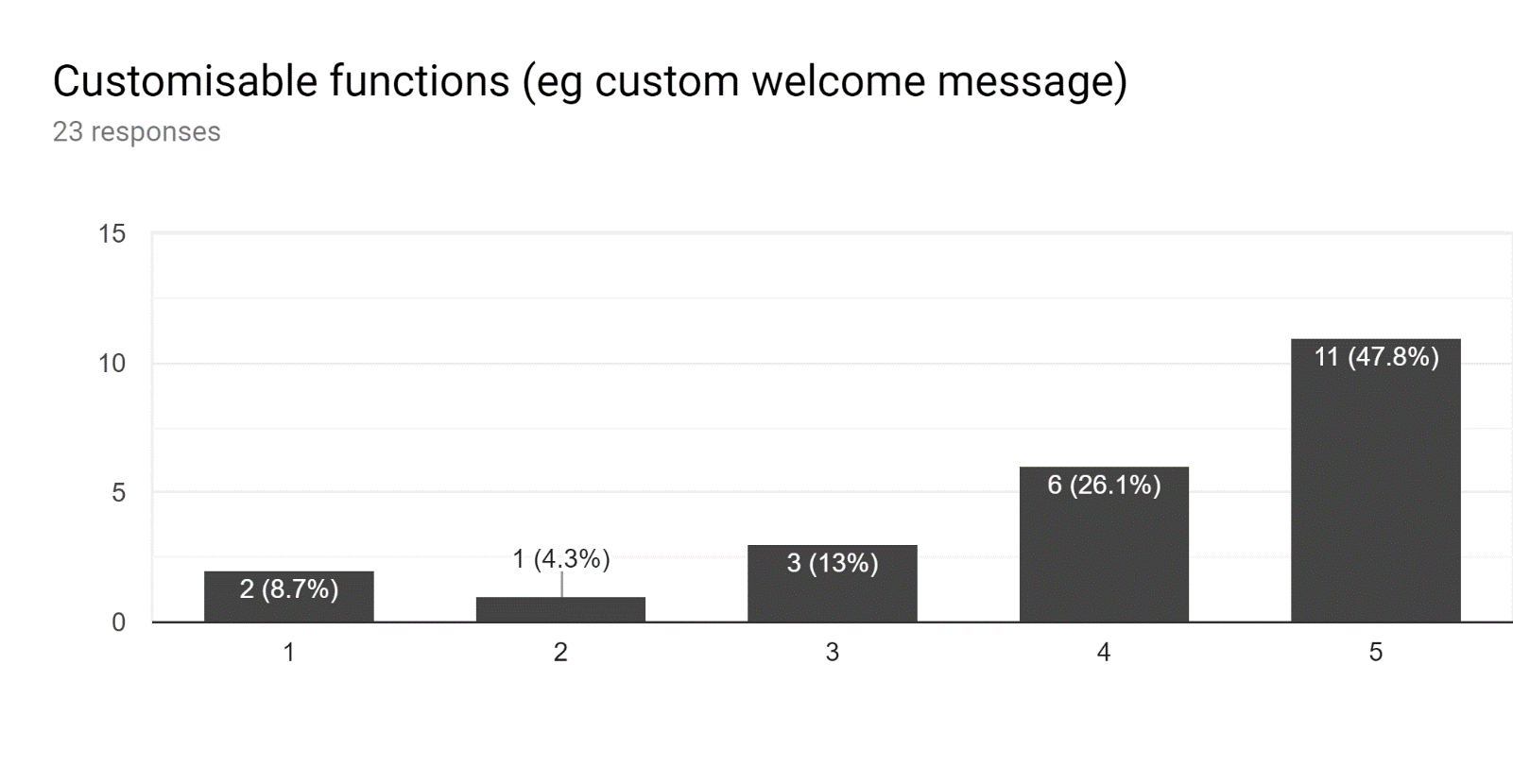












The data from the surveys shows a clear set of initial priorities for development, primarily uptime and Automated management, as well as a wide range of functionality. Using this and my findings from my existing solution analysis, I will create a set of development priorities for Overlord.

Existing Solution analysis

Due to the popularity and the demographic of people that discord is aimed towards, there are quite a few existing solutions – some of which are far more popular than others. To prevent succumbing to “second system syndrome” as well as to evaluate what other solutions have done right, Using a popular discord bot ‘ranking’ website (<https://discord.bots.gg/search?sort=guildcount>), here are overlord’s Top 5 Competitors:

1. MEE6 – this bot primarily focuses on ease-of-use and administration, focusing on polished ‘basic’ features over more advanced functionality. Possessing an easy OAuth dashboard for cross-guild management, as well as a very intuitive user interface. The overwhelming popularity of this bot further proves my earlier assertions that focusing on more ‘basic’ features will be a better use of my time.
2. Dyno – Dyno is generally perceived to be one of the, if not the, best all-round bot for both advanced and basic users. It includes a somewhat complex dashboard, but offers an inordinate amount of functionality, including Highly customizable and advanced auto moderation/management features, as well as plenty of other functionality, backed up by a nigh-impeccable User Experience (UX). It is undoubtedly my biggest inspiration for the feature set/execution for my bot.
3. Pokécord – Pokécord is primarily Minigame based that focuses on the idea of allowing users to catch and battle Pokémon through just a text-based interface. Like MEE6, it has a very intuitive Command / ‘menu’ structure as well as providing users with similar commands if they happen to misspell the command (dynamic command reporting)
4. ZeroTwo – ZeroTwo is a ‘hybrid’ bot that strikes a balance between moderation and more ‘fun’ features such as Minigames, with automated role allocation, moderation, and custom bot-specific user ‘profiles’. Like MEE6, it has a web dashboard, and like Dyno, it has an excellent User Experience element (command design and reporting)
5. GiveawayBot – as the name suggests, this bot’s main (and only) function is to facilitate the creation and management of giveaways through the usage of embeds and reactions (you’ll find out what these are and why they’re useful later!) with the bot having achieved the simple goal of doing one thing, but doing it really, *really* well.

Conclusion and further analysis

As previously stated, the Number of Bots present in the Discord ‘Ecosystem’ is staggering, nonetheless, only a handful of bots occupy 90% of the servers containing bots. Why is this? after analysing the common aspects of the top 100 discord bots, Simply enough, it’s ease of use above all else. It doesn’t really matter the breadth of functionality or the amount of customisability, **if it’s not easy to use, it (generally) won’t succeed.** With this in mind, I’ve had to take a big step back and reorganise my developmental priorities for Overlord to really hone in on this singular idea – unfortunately I’m in no position to host Overlord for 10,000+ servers, which is going to be the main detrimental factor for this bot’s success in terms of ease of use. It doesn’t mean that there’s not plenty I can do to make it as easy as possible to use. After evaluating these existing solutions and this key factor, I’ve created a list of development priorities:

|  |  |
| --- | --- |
| Concept | Focus points |
| Ease of use | dynamic reporting, good UX |
| uptime | PM2, resiliency (‘safe mode’), reporting systems |
| Moderation | Automatic content filters, punishments, content removal, permission management |
| Ease of administration | Automated role assignment, ability to preform actions on a user set, Permission systems |

Defeating Second System Syndrome at the drawing board

Second system syndrome is one of my greatest fears as I progress through the development of this project, for no other reason than it has killed so many projects (mercifully none of mine yet!). second system syndrome is where an existing set of solution(s) already exists, and someone like me comes along and aims to ‘do what [they] did but do it better’ then end up setting lofty goals they can never really attain given constraints, such as time or resources, oftentimes leading to a complete project failure. To mitigate this as much as possible, I’m going to ensure that my goals are both realistic and balanced with my other development priorities. To do so, I am going to plan out as many features as possible in advance, setting realistic goals for each, and most importantly **not deviating from these goals unless absolutely necessary.** It’s easy enough for me to get engrossed in something and overengineer it as is, but given the time constraints at hand, doing so would most likely render this project non-viable. Not to say finding a better solution/spending time once the core features are complete isn’t going to happen, I just want to ensure that my time developing this Bot is used as effectively as possible.

What will Overlord Do differently?

With Overlord, I want to produce an easy-to-use and administrate Bot, that’s completely open source and free, easily self-hostable, modular and thus entirely customisable. Currently there are no (or next to no) Bots that accomplish all of these things, but Overlord can accomplish these things primarily due to the lack of a hosting cost as it will be entirely selfhosted, meaning there’s no need to paywall ‘advanced’ features, or restraints on additional functionality due to security concerns. The focus on user customisability will be easily accomplished by the easy to use and understand data structures and systems that will make up the ‘core’ - as well as the potential creation of a ‘getting started’ guide for adding custom code into Overlord.

Secondary Feature analysis/Limitations

As Overlord is designed to be a High-performance application with minimal overhead, this imposes limitations primarily on Memory (RAM) usage and CPU cycles. As Discord.js (and by extension Discord itself) has a very efficient network layer that my bot will be using, network performance (apart from latency but there’s not much I can do about that) is less of an issue, as is non-volatile storage on the host machine (as the persistent Database is very efficient). One of the main ways I might be able to mitigate the memory residency/footprint is through a custom ‘cache garbage collector’ for the in-memory version of the database, which will unload infrequently used data from the memory constantly to reduce the amount of memory the database cache is consuming. I will however have to balance this with the CPU demands of a garbage collector but implementing it on a timer basis should solve this issue. Some planned automatic moderation features (mainly content recognition) will massively increase resource demands, but these features will be togglable.

I also need to ensure my Application is fully capable of being platform agnostic, as I want it to be easy to self host, even without the usage of Docker containers (which are inherently platform agnostic). To this end, I’m using platform agnostic relative paths for data files that the bot uses, with the rest of the project already being platform agnostic.

Overlord also needs to have as close to 100% uptime as possible – to ensure this is the case, I will be using a process manager for node called PM2, that has the functionality of collating logs, errors, and the ability to safely shutdown/restart apps in the case of an unexpected crash, as well as automatically restarting Overlord upon the host rebooting (on Linux systems anyway). I will also aim to integrate automatically generated logs with Docker’s Built in Logs manager.

This feature set allows me to ensure that even in the event that Overlord suffers from a catastrophic failure that results in it crashing, it’ll be able to restart itself and its internal integrity checks will hopefully sort the issue out. I’m also going to implement an Opt-out log aggregation (telemetry) system to allow me to aggregate error logs across all the instances of the bot running. Any information will of course be anonymised to the greatest extent that I can. This feature will be one of the last features I will implement though as It’s only useful or me for generalised bug fixes, and users can always raise issue tickets on the GitHub page whenever they encounter an issue.

Many of the above features will take a back seat in terms of developmental priorities, as core functionality of Overlord will naturally be the most important to develop first – but these features will be developed as soon as the primary feature set is completed.

Objectives and Performance Criteria

To more definitively quantify the goals for this project, I’ve created a first version of my Objectives and Performance Criteria table, which indicated what Functionality/Objectives I want for Overlord, as well as the Performance Criteria/solution requirements for each Objective. This table is as follows:

|  |  |
| --- | --- |
| Objective | Criteria |
| High-performance database | Low-latency persistent database capable of storing/manipulating information in JSON notation with an additional In-memory Cache |
| Platform agnostic code | Docker container and Architecture-neutral relative pathing |
| Program Resiliency | PM2 Application auto-start, log aggregation, and Automatic restart features |

Evaluating these performance criteria as well as evaluating the best solutions to accomplish what I want to for the Overlord project, I’ve decided to code the bot primarily in Node.js. Node.js is the best option for this project as the discord API wrapper (Discord.js) is by far the most well developed and documented API wrapper for Discord to date. Node.js is also designed for high performance web-based applications such as serving web pages /API requests, which is the type of behaviour Overlord will be performing. Despite having more experience in python compared to JS, I’m confident in my JS skillset and also seek to use this opportunity to develop my JS skillset further. I’ve chosen Docker containers as a secondary platform for their ease of virtualisation, as well as their low resource overhead in doing so, allowing me to make Overlord truly platform agnostic as all hardware can be virtualised to Overlord’s needs.

System Modelling

Overlord relies on and is part of a large set of operations to go from the user sending a message to the bot’s response. The easiest way to model the top-level of all this behaviour is through a process diagram, highlighting the most common event/data flows through the application

A screenshot of a cell phone

Description automatically generated

This flow shows the ‘most common’ data flow for Overlord (see the black trace line), with the specific example of the Bot receiving a message (which is also the most commonly performed operation). The colours represent associated data types/functions.

Discord.js Implementation and information

Discord.js is a Node.js wrapper for discord’s REST API. It allows for Overlord to more easily communicate to the REST API than a ‘bare-metal’ Direct to REST API procedure. The primary Operation Overlord will be performing revolves around the ‘message’ event. This event is triggered whenever any user sends any message in any of the channels of any of the servers, and as such has to be the most performant part of the bot due to it triggering with every single message sent in any of the channels. The flow of data and execution can be seen in the following flow diagram:A screenshot of a cell phone

Description automatically generated

For both discord (and to a lesser extend Overlord) a users permissions (what actions they can and cannot do in a specific channel) is defined by a permissions integer. For discord, just integer can be logically AND’d to deduce it’s component permissions from a single integer. Overlord will primary use an equivalent “Flag” system to indicate permissions.

Key project terminology and Concept

Events: Events are

system design:

as Overlord is a Bot that is rather data-heavy, moreso than some of it’s competitors at the very least. Thus, one of the most important design choices to make is the choice of database. I wanted a database system that has exceptional performance, on-disk persistency, and is easy for me (and users who want to develop their own code) to use. I ended up deciding on a few database solutions that work off key:value storage with support for complex objects and with both built-in In-memory cache and on-disk persistency. Here are my current options for a database engine:

1. Redis

* Redis is a very popular and extremely powerful large-scale database engine, designed for edge enterprise-scale deployments. Due to it’s popularity it has a large support community around it, with no end of plugins and libraries, some of which I would no doubt use. However, as Redis is not purpose built for this application it has a rather large overhead in terms of additional functionality my bot simply won’t use.

1. MongoDB

* Like Redis, MongoDB is a wildly Popular and Powerful Database engine also similarly designed for large scale deployments with multiple endpoints. It too has a large number of 3rd and 1st party integrations, many of which I would use for Overlord. As with Redis, it has a lot of overhead with unnecessary functions

1. ENMAP

-ENMAP (short for ENhanced MAP) is a relatively niche database engine almost exclusively designed to act as a database for Node.JS applications/Discord Bots. Due to it’s simple concept (enhancing the native map functionality for key:value storage) and somewhat limited functionality, it has very little overhead in terms of functionality overlord would not end up using. It may not be as popular and as heavily supported as the other options, but it is certainly closer to a purpose-built system than the other two.

To help in comparing the database solutions, I installed each on a ‘fresh’ virtual machine (Ubuntu 16.04 LTS from the official website, updated using APT), then installed Node.JS and ran a set of scripts to perform read/write actions to/from the database (running locally on the VM). The data and number of calls was identical for each VM. I used PM2 to monitor application performance whilst these tests were running. The results showed that ENMAP both had lower Idle and Load resource consumption, followed by REDIS and then MongoDB, with read/write access times being within margin of error of each solution. ENMAP was also able to be read/written to Synchronously rather than the other two that require asynchronous calls to set/get data. This is what I consider as a tie breaker feature that means that implementing the database within functions is significantly easier, especially when some of the functions are inherently Synchronous. Due to these reasons, I have chosen ENMAP for my database engine for this project – whilst it doesn’t allow for sharding (having multiple backend bot instances load balance a single bot – requires the database to be able to cope with multiple read/writes which ENMAP can do but it’s not very well supported) the containerisation of the bot and the idea of self hosting a bot by the end user for a low number of servers means this concern is rendered all by invalid with the design I have for overlord. If the user needs to ‘shard’ to continue operating the bot (which is something discord requires over 2500 guilds!), they can simply duplicate the existing container.