Project Overlord

Jesse Cruz Wright

27 April 2020

CONTENTS

List of Figures 5
List of Tables 7
1 Analysis 8
1.1 Project Background 8
1.2 Project Outline 8
1.3 Client 8
2 Research 9
2.1 Client survey 9
2.2 Existing Solution Analysis 12
2.3 Initial Development Objectives 13
2.4 Technical Implementation 13
2.4.1 Performance Considerations 13
2.4.2 Language 14
2.4.3 Database 15
2.4.4 Commands 16
2.4.5 Project Architecture 17
2.4.6 Automated Content Moderation 20
2.4.7 Machine Learning Subsystem 20
2.4.8 Heuristic Module 22
2.4.9 Punishments 22
2.4.10 Audit Subsystem 23
2.4.11 Human Integration 24
2.4.12 User Experience (UX) 24
2.5 Technical Development Criteria 27
3 Programming 28
3.1 Prototype 1 – Basic Functionality 28
3.2 Prototype 2 – Embed Functionality 29
3.3 Prototype 3 – Modular Event Code 29
3.4 Prototype 4 – ENMAP Implementation 30
3.5 Prototype 5 – Database Initialisation 31
3.6 Prototype 6 – Required Permissions 33
3.7 Prototype 7 – Custom Logging Solution 34
3.8 Prototype 8 – Config.js Implementation 36
3.9 Prototype 9 – Shutdown Procedures & Fatal Error Handling 37
3.10 Prototype 10 – Discord Rich Presence 39

3.11 Prototype 11 – Implement Client Events 40
3.12 Prototype 12 – Command Processing Discriminator 42
3.13 Prototype 13 – Basic Modular Commands 45
3.14 Prototype 14 – Command Reloading 47
3.15 Prototype 15 – Scheduler and Persistence 50
3.16 Prototype 16 – Finalise Message Event 53
3.17 Prototype 17 – Moderation Heuristics 61
3.18 Prototype 18 – Attachment Recording 68
3.19 Prototype 19 – NSFW UGC Classification 73
3.20 Prototype 20 – UGC Toxicity Classifier 80
3.21 Prototype 21 – Automatic Moderation Backend 87
3.22 Prototype 22 - Punishment 102
3.23 Prototype 23 – Function Compilation 109
3.24 Prototype 24 – Develop Event Stubs 119
3.25 Prototype 25 – PM2 Integration 131
3.26 Prototype 26 – Finalise Help Command 132
3.27 Completed System Testing 136
3.28 Feedback from a live run on an active Discord guild 141
3.29 Finalised Structures 143
3.30 Evaluation 146
4 Appendix - Finalised codebase 151
4.1 Overlord.js 151
4.2 Config.js 155
4.3 Functions.js 155
4.4 Ecosystem.config.js 167
4.5 ./commands/eval.js 168
4.6 ./commands/help.js 168
4.7 ./commands/reload.js 170
4.8 ./commands/restart.js 170
4.9 ./events/attachmentRecorder.js 171
4.10 ./events/autoMod.js 173
4.11 ./events/guildCreate.js 179
4.12 ./events/guildMemberAdd.js 179
4.13 ./events/guildMemberRemove.js 180
4.14 ./events/guildMemberUpdate.js 181
4.15 ./events/guildUpdate.js 181
4.16 ./events/message.js 182
4.17 ./events/messageDelete.js 187

4.18 ./events/messageReactionAdd.js 188
4.19 ./events/messageUpdate.js 189
4.20 ./events/modActions.js 190
4.21 ./events/NSFWClassifier.js 198
4.22 ./events/scheduler.js 201
4.23 ./events/toxicClassifier.js 205
4.24 Requirements/Dependencies 210

LIST OF FIGURES

Figure 1 - Automatic Moderation Features Priority Graph 9
Figure 2 - Uptime and Performance Priority Graph 9
Figure 3 - Automated Role Management Priority Graph 10
Figure 4 - Wide Range of Functionality Priority Graph 10
Figure 5 - 'Fun' Features Priority Graph 10
Figure 6 - Ease of Administration Priority Graph 11
Figure 7 - Customise Functions Priority Graph 11
Figure 8 - Database entry - guild - schema 18
Figure 9 - Parent-child diagram of notable data structures 19
Figure 10 - Neural Network Confusion Matrix 21
Figure 11 - Graph representing a model of the points system 23
Figure 12 - Demonstration of Error Reporting UX 24
Figure 13 - Demonstration of Possible Embed Attributes 25
Figure 14 - Overlord Overarching Process flow 26
Figure 15 - Message event Overarching Process Flow 26
Figure 16 - Picture of Call-Response Behaviour 29
Figure 17 - Rich Presence within Discord 39
Figure 18 - Initialisation Message 39
Figure 19 - Lack Of Permissions Error Message 60
Figure 20 - General Error Message 60
Figure 21 - Unknown Command/Alias Error Message 61
Figure 22 - Automated Moderation Action Flow Diagram 87
Figure 23 - Example Action Embed 101
Figure 24 - Example Report Embed 101
Figure 25 - Example Notification Embed 101
Figure 26 - Example Audit Embed 102
Figure 27 - Embed with Member Reference 124
Figure 28 - Example Message Edit Audit Embed 126
Figure 29 - Message and Audit Embed 129
Figure 30 - Audit Log Entry 129
Figure 31 - Deleted Message Audit Embed 130
Figure 32 - Raw Event Property Diagram 130
Figure 33 - Example Help Embed 135
Figure 34 - Test 1 - Initialisation Message 139
Figure 35 - Test 3 - Help Embed 139

Figure 36 - Test 4 - Reload Help Embed 140
Figure 37 - Test 6 - Eval Error Embed 140
Figure 38 - Test 8 - Non-Existant Error Embed 140
Figure 39 - Test 15 - Spam Removal Action Embed 140
Figure 40 - Test 37 - Message Deletion Audit Embed 141
Figure 41 - Test 38 - Message Deletion with Attachment Audit Embed 141
Figure 42 - Test 39 - Message Update Audit Embed 141
Figure 43 - System Data structure Diagram 143
Figure 44 - Overlord Message Event Flow 144
Figure 45 - Overlord System Event Flow 144
Figure 46 - Automatic Moderation Process Flow Diagram 145
Figure 47 - Example Module defaultConfig 146
Figure 48 - Example Command defaultConfig 146

LIST OF TABLES

Table 1 - Project Objectives 13
Table 2 - Technical Development Objectives & Criteria 27
Table 3 - Prototype 1 Testing Table 29
Table 4 - Prototype 2 Testing Table 29
Table 5 - Prototype 3 Testing Table 30
Table 6 - Prototype 4 Testing Table 31
Table 7 - Prototype 5 Testing Table 32
Table 8 - Prototype 6 Testing Table 34
Table 9 - Prototype 7 Testing Table 36
Table 10 - Prototype 8 Testing Table 37
Table 11 - Prototype 9 Testing Table 39
Table 12 - Prototype 10 Testing Table 40
Table 13 - Prototype 11 Testing Table 42
Table 14 - Prototype 12 Testing Table 45
Table 15 - Prototype 13 Testing Table 47
Table 16 - Prototype 14 Testing Table 49
Table 17 - Prototype 15 Testing Table 52
Table 18 - Prototype 16 Testing Table 60
Table 19 - Prototype 17 Testing Table 68
Table 20 - Prototype 18 Testing Table 72
Table 21 - Prototype 19 Testing Table 80
Table 22 - Prototype 20 Testing Table 86
Table 23 - Prototype 21 Testing Table 101
Table 24 - Prototype 22 Testing Table 109
Table 25 - Prototype 23.1 Testing Table 114
Table 26 - Prototype 23.2 Testing Table 118
Table 27 - Prototype 23.3 Testing Table 119
Table 28 - Prototype 24.1 Testing Table 120
Table 29 - Prototype 24.2 Testing Table 123
Table 30 - Prototype 24.3 Testing Table 126
Table 31 - Prototype 24.4 Testing Table 128
Table 32 - Prototype 26 Testing Table 134
Table 33 - Copy of the Technical Objectives Table 148

1 ANALYSIS

1.1 PROJECT BACKGROUND

Over the last few years, many people like myself have started using the now massively popular social media platform Discord. Discord is a popular communication app, that allows for 'servers' that contain both text and voice channels – some have likened it to 'slack for gamers'. Users (administrators, moderators and members) can join these servers and are then managed by a complex permissions system to fine-tune what users can do what where, primarily governed by 'tags' called roles (more on these later), as well as human moderators/administrators, who have authority over regular members. Due to the large numbers of members, manual moderation quickly became inefficient – the general trend is the number of moderators per member decreases as the server member count increases. This means that moderation is especially difficult for larger servers (10,000+ members can have as few as eight moderators).

This reliance on purely human moderation changed when Discord made public their REST/WebSocket (WS) based API just over four years ago, 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 primary aim of offloading as much content moderation as possible to an automated system - ideally leveraging Machine Learning (ML). Overlord also aims to be easily self-hosted and modifiable by administrators seeking to add custom functionality.

1.2 PROJECT OUTLINE

My end users for Overlord will be administrators and moderators (admin group) and general guild (Discord server) members. To this end, I want to ensure that the management backend/system is simple enough for first-time set and forget administrators (admins), but also powerful, extensible, and comprehensive enough for those seeking to add custom functionality - whilst remaining robust and easy to use for the other 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 in conjunction, or simply not at all (such as the ML components and integration of end-user created features) whilst being completely open-source (all files placed on a public GitHub repository).

A Discord bot functions as a 'fake user' – using the Discord WS API to receive 'events' – actions that occur – of various types on the guilds they are a part of, such as a message being sent, a new user joining, or a new channel being created. The main way they interact directly with users is via commands, where users can type out a message with specialised prefix character(s), e.g. "!" followed by a command name and then arguments to execute a command on a specific bot. How a bot responds to events is the primary focus of designing a Discord bot. More details of the specifics and limitations on bot functionality will be expanded upon in later sections.

1.3 CLIENT

My client for this project is primarily a group of Discord server moderators/administrators that are looking for a way to manage a large set of servers without requiring a small army of moderators to be present at nearly all times. This group has permitted me to perform controlled tests of the bot in their guild, as well as some key metrics and usage statistics for the initial tests. The secondary group is that of the Discord server members – those that will be managed by Overlord once it is completed. The viewpoints of both sides of the system – administrators/moderators and members – is valuable to shaping the design priorities that will be used to formulate the developmental plan as well as the feature set moving forward.

2 RESEARCH

2.1 CLIENT SURVEY

I created and distributed a simple survey on Google Forms, allowing people who frequently use Discord (members of the client group and others) to answer on a scale of 1 to 5 (1: least important, 5: most important) about what they think are the most important features of a Discord bot. This information will be key for establishing my base development priorities. The survey results are displayed in Figures 1 to 7.

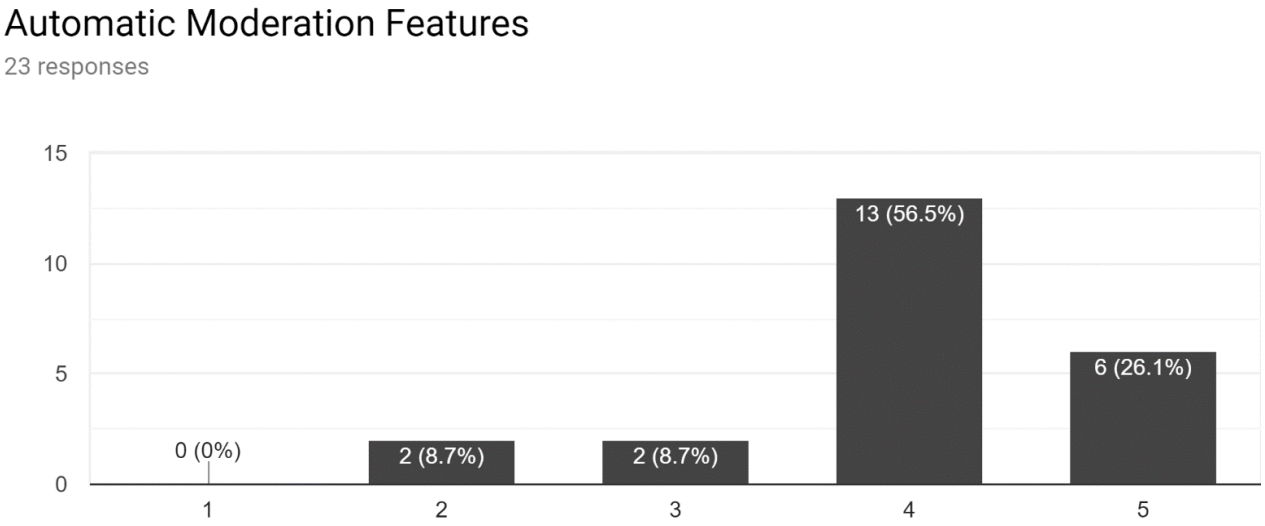

Figure 1 - Automatic Moderation Features Priority Graph

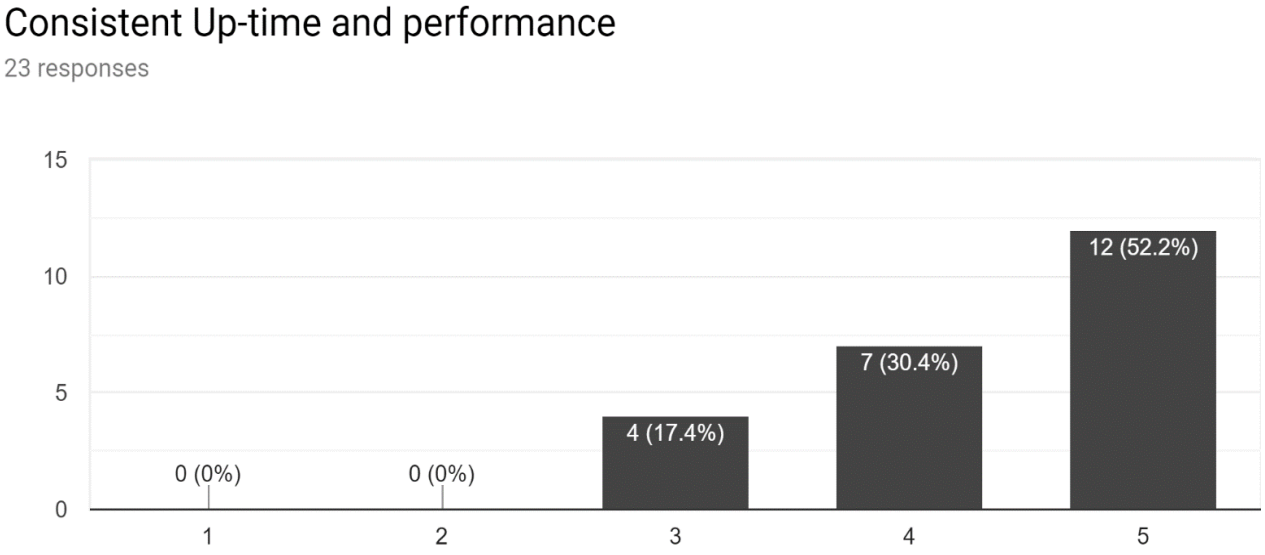

Figure 2 - Uptime and Performance Priority Graph

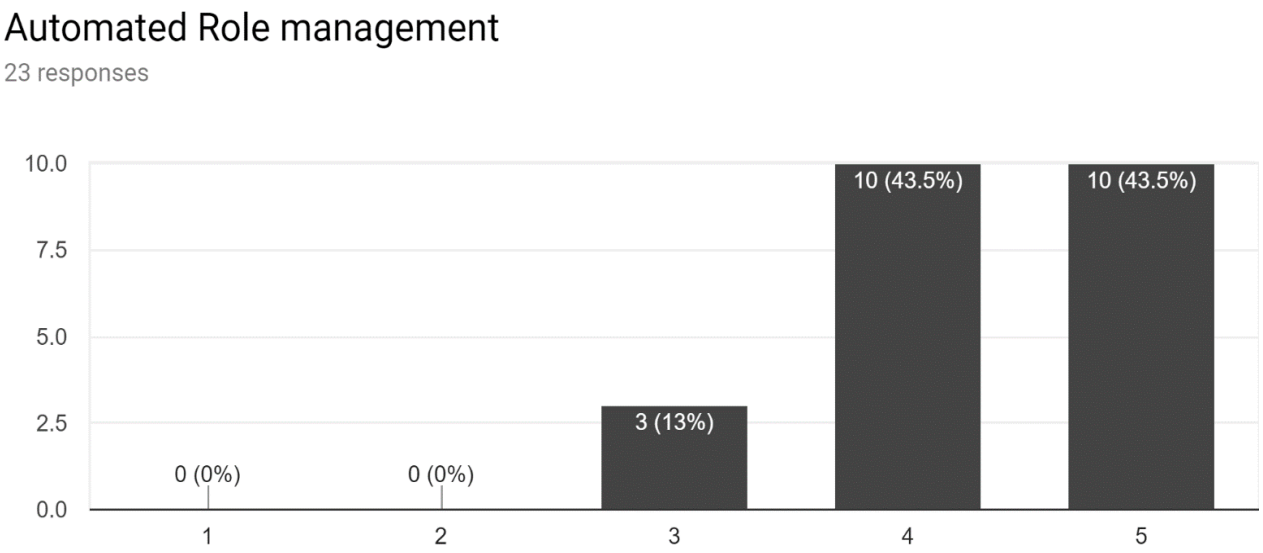

Figure 3 - Automated Role Management Priority Graph

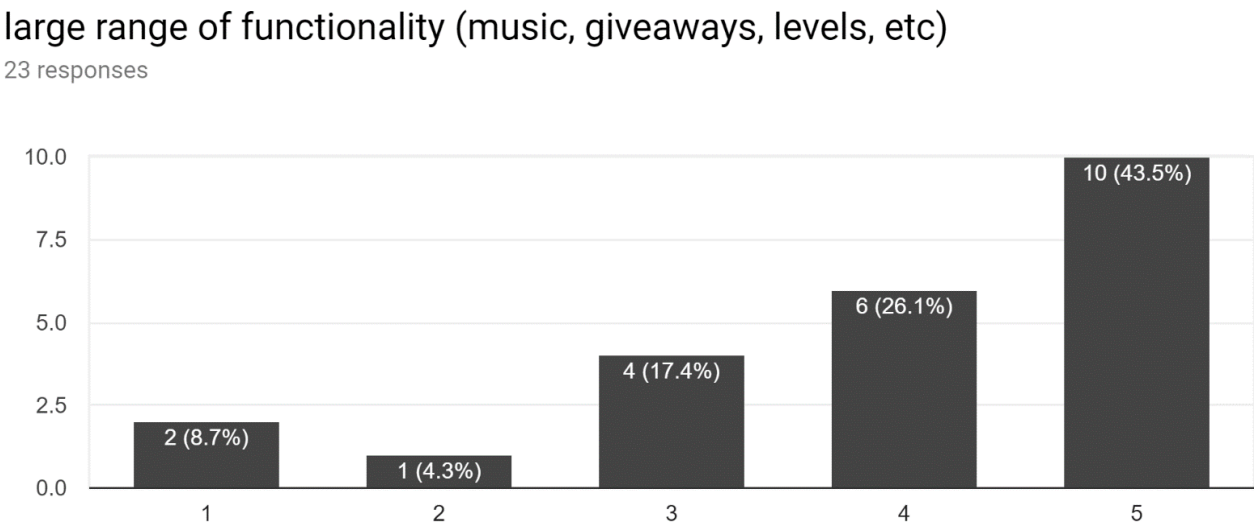

Figure 4 - Wide Range of Functionality Priority Graph

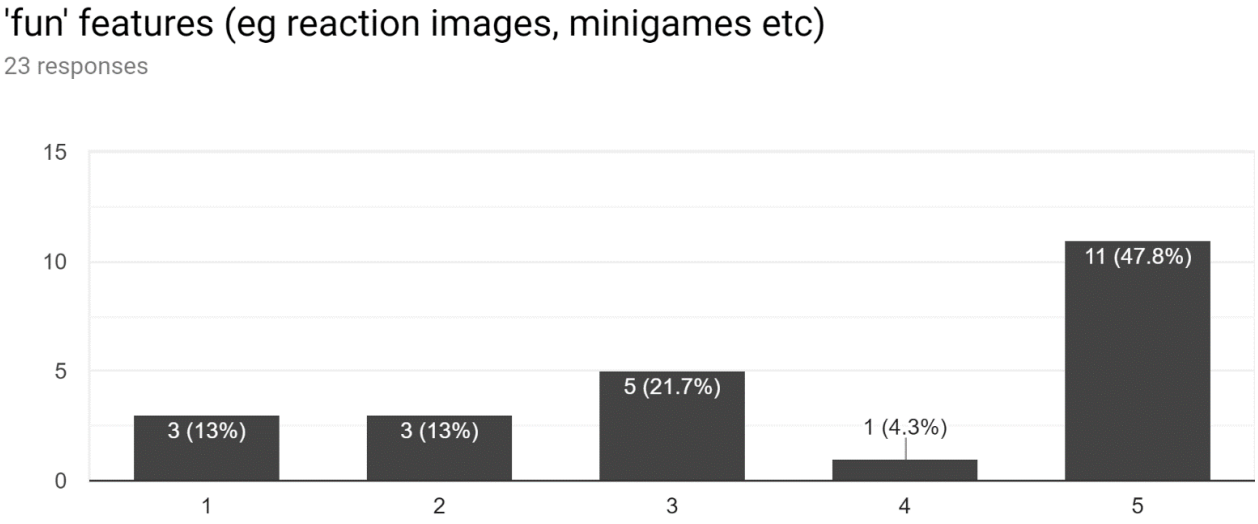

Figure 5 - 'Fun' Features Priority Graph

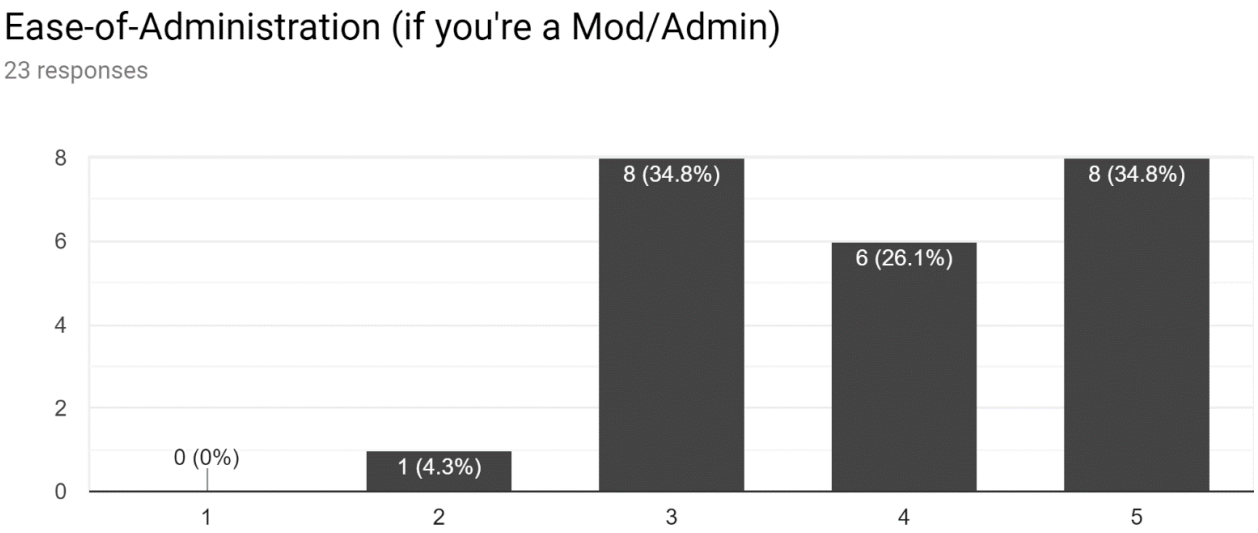

Figure 6 - Ease of Administration Priority Graph

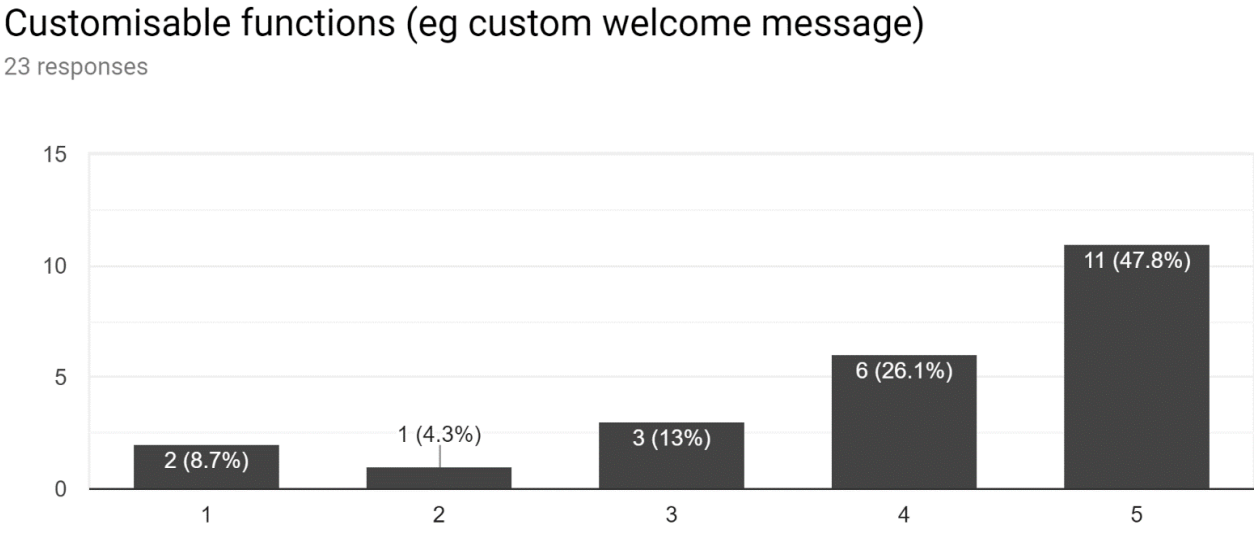

Figure 7 - Customise Functions Priority Graph

Although a limited number of responses (23) were received, the trends were very interesting to examine regarding my developmental priorities. It showed that members place a higher demand on simpler features (such as automatic moderation) than more 'advanced' features (such as the ability to add custom commands) which is behaviour that would be expected given the 1 to 100+ moderator to member split and the differing priorities of each group. After discussing with the administrators/moderators of my client group, their very strong desire for uptime, followed by automatic moderation features, ease of administration, automated role management, customisable functions, and then a large range of functionality and 'fun' features was confirmed. As the split between administrators and users is rather large, this helped me to put the data from the survey into context. As an administrator myself, I decided to place a greater degree of focus on the priorities of the admin client group than the member client group as It was determined Overlord will be far more desirable if it had these features over a more user-focused bot.

After discussing my analysis of the results offered by the Google Forms with the admin client group, they reassured me that the main problem facing their duties was simply the overwhelming volume and unceasing nature of user-generated content (UGC) – after all, members do not *tend* to report content that breaks the rules, leading to admins having a greatly increased workload - having to unconditionally sift through large volumes of content simply to figure out if any rule-breaking has occurred. After suggesting the usage of neural networks (NN) as a way of processing UGC, there was an expectedly mixed reception due to the perceived lack of

accuracy of such systems, but the idea of using more conventional heuristics received a much better reaction due to their widespread usage. Many of the group expressed the idea that if the NN implementation could be proven to be accurate enough, then they agree that the time saving would be immense, and they would very happily invest in the additional costs of having to self-host the bot, as well as the sacrifices to other functionality that may have to be made in the process of development. This confirms my initial idea that the implementation of NN processing will prove invaluable to the bot's appeal and success as a primary differentiating feature not found in any other bots to date, at least as far as I am aware, and that the focus should be on the moderation system and the surrounding components.

2.2 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 at the time of writing:

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 best all-round bots for both advanced and basic users. It includes a somewhat complex web dashboard but offers an inordinate amount of functionality, including highly customisable and advanced auto-moderation/management features, as well as plenty of other functionality, backed up by nigh-impeccable user experience (UX). It is undoubtedly my biggest inspiration for the feature set/execution of Overlord.

3.) Pokécord – Pokécord is a primarily minigame based bot focussing 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 UX element (command design and reporting)

5.) Giveaway bot – 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 (specialised message elements) and reactions (an emoji response to a message). This is an example of a bot achieving the simple goal of doing one thing, but doing it really, *really* well.

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 ten Discord bots, I determined it is ease of use above all else. It doesn't 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 step back and reorganise my developmental priorities for Overlord to 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 than can be done do to make it as easy as possible to use.

2.3 INITIAL DEVELOPMENT OBJECTIVES

After evaluating these existing solutions and the initial technical experimentation, I've created a list of development objectives as shown in Table 1.

Objective	Criteria
Fluent User Experience	The bot needs to be able to fluently communicate information in various forms to users whilst remaining within the Discord app in such a way that no unexpected behaviour on the part of the bot is left unexplained to the user
Consistent uptime (4.5 9's)	The bot needs to be able to rapidly recover from a critical failure state and needs to be mitigating as many failure states as possible to ensure uptime. Under no circumstances should the bot be left in a state wherein it is inoperable due to a failure state within itself. 4.5 9's = 26 minutes of downtime/year (99.995% uptime)
Automatic Moderation	The bot should be able to automatically screen and identify many types of content, penalising it appropriately and enacting punishments if required. It should do so by following a set of heuristic guidelines that it is configured with. The bot should be able to moderate with only optional human intervention, but human intervention should be an option for certain systems.
Ease of administration	The bot should handle automated role assignment, as well as the ability to perform actions to a group of users, as well as implementing other features that streamline the tasks of moderators.
High performance	The bot should make very few unconditional compromises against performance, especially concerning key processing routines. The bot should use a programming language, API wrapper, and database solution geared towards enabling this objective.
Extensibility /modularity	The bot should have almost all functions easily extensible or accessible via client-generated functionality, should the client create additional/new functionality for the bot.
Privacy	The bot should ensure that any crucial information recorded is done so in a secure fashion and users have the option to remove any stored data provided the data does not expire in some fashion.

Table 1 - Project Objectives

Using this table as a means of evaluation, I will be evaluating the systems that will be used for my technical implementation.

2.4 TECHNICAL IMPLEMENTATION

2.4.1 PERFORMANCE CONSIDERATIONS

As Overlord is designed to be a high-performance application with minimal overhead, as well as the assumption of being hosted on a low powered device/virtual private server (VPS) imposes limitations primarily on memory (RAM) usage and CPU cycles. As almost all API wrappers (and by extension Discord itself) have a very efficient network layer, network performance (apart from latency) is less of an issue, as is non-volatile storage on the host machine, as Overlord does not store much data persistently, and will offload media to an external service (transfer.sh). 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 – both on a per-guild and overarching per bot instance.

Overlord also needs to be fully capable of being platform agnostic, as it must be easy to self-host, even without the usage of Docker containers (which are inherently platform agnostic). To this end, Overlord will be using platform-agnostic paths for data files and other resources that it uses, with the rest of the project already planned to be platform agnostic due to the natures of the languages that will be evaluated.

Overlord also needs to have as close to 100% uptime as possible (aiming for the 4.5 9's of enterprise: 99.995%). To ensure this is the case, a process manager that has the functionality of collating logs, errors, and the ability to safely shutdown/restart Overlord in the case of an unexpected crash, as well as for instance monitoring - allowing for the automatic restart of Overlord upon the host rebooting, or a fatal error occurring, will be used. Whilst the process could be configured to not exit upon a fatal error, the usage of the process manager as well as the fact the program will be in an 'unclean' state after this means that this is not a good idea.

This feature set mainly allows me to ensure that even if 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. It is also planned to implement an opt-out log aggregation (telemetry) system to allow for aggregation of error logs across all the instances of the bot running so that I as the author will be able to fix any persistent bugs. Any information will, of course, be anonymised to the greatest extent that can be done. This feature will be one of the last features planned to be implemented, as it is only useful for 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 the 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.

2.4.2 LANGUAGE

The language in which Overlord will be written in is quite possibly the most important decisions, as it will drastically alter the characteristics of the final product. For each language, I've evaluated its suitability for a primarily web-based heavily parallelised system against my proficiency of the language as well as the API wrappers that each language has. (I have decided to use a wrapper due to the enormous – and arguably unnecessary workload creating an API interface would be). I have decided to more thoroughly evaluate the top three languages of my initial evaluation.

The first main language for consideration for Overlord is Python.

Python has one of the more well maintained and robust Discord API wrappers (see https://discordpy.readthedocs.io) – However, as Python is not primarily designed towards heavily parallelised web-based functionality, the asynchronous behaviours that would be needed are not generally considered as polished as some of the other options (notably JS). Python, however, does have more support for running hardware-accelerated subprocesses, this coupled with it being the platform of choice for Machine Learning means that using Python for the planned ML modules may prove invaluable. It is also the language I have the highest proficiency with, and as such my time spent working in Python will be more effective in terms of work done compared to the other languages

The second language being considered for usage in Overlord is JavaScript – specifically deployed in a Node.js environment.

The Discord.js API wrapper (see discord.js.org) boasts the highest API feature parity against the official WS API as well as the official client (as the client is coded in JS), which is good for an application like Overlord - as oftentimes these updates can lead to some significant improvements in application performance. JS is also an object-oriented scripting language, which works very well for my use case, as the scripting nature means that

general clean-up is handled automatically without the need for an aggressive garbage collector (such as that found in Java) once the script is terminated. It also facilitates the ease of implementation of heavily parallelised subsystems that can be used to scale with demand, something that would otherwise have to implement. Due to its nature as a primarily web-based scripting language, it is extremely performant with minimal overhead. I have some JS experience from other projects, so the time to work done trade-off would not be too far off that of Python. It does, however, have some difficulties interfacing with local resources, as many modules are designed for a transient web-only environment (regular JS compared to Node.JS).

The third and final language for consideration is Java.

Java is a very established language, with its retrospectively cutting-edge reliance of virtual machines for application sandboxing/management means if Overlord were to be placed into a Docker container, this would not be necessary due to the Java Virtual Machine effectively accomplishing this for me. This also means more control over resource utilisation, a fact that is reflected in the JDA (Java Discord API – see https://github.com/DV8FromTheWorld/JDA), which is comparatively more low level and thus far more powerful by default in terms of potential functionality. However, I have very little Java experience, this coupled with the relative complexity of some of the systems planned to be implemented means it would take me comparatively more time to get work done using Java.

The language I eventually chose as the optimum to utilise for the development of Overlord was (Node.)JS. This is primarily due to the latent transient property of the scripting nature, allowing for a 'clean slate' for every execution of a script – with many executions being performed in parallel – another concept that JS handles very, very well compared to language with more 'overhead' such as Java, which also means it has stellar networking with, again, very little overhead – also important for the many smaller packets that will be sent/received by the bot. Despite my comparative lack of experience with JS compared to Python, the amount of time I would spend learning the language will be less than the time I need to learn the specific libraries, this coupled with the more mature nature of Discord.JS means that this time is reduced further.

2.4.3 DATABASE

As Overlord is a rather data access-heavy bot, one of the more important design choices to make is the choice of database. 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 is required, so I ended up deciding on a few database solutions that work off key-value pair storage with support for complex objects and with both built-in in-memory cache and on-disk persistency that function with Node.JS. 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 its popularity, it has a large support community around it, with no end of plugins and libraries, some of which would no doubt be use. However, as Redis is not purpose-built for this application, it has a rather large overhead in terms of additional functionality Overlord simply won't use.

2.) MongoDB

Like Redis, MongoDB is a 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 would be of use for Overlord. As with Redis, it has a marginal amount of overhead for my use case

3.) 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 its 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 pretty much a purpose-built system for this use case.

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 transferred and the 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 faster on average for ENMAP, and within what would consider within the margin of error between the other solutions. A key advantage of ENMAP was that it was able to be read/written to synchronously rather than the other two that require asynchronous calls to set/get data. This is what I would consider as a tie-breaker feature that means that implementing the database within functions is significantly easier (mainly for new users who want to add features), 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 but invalid with the design I have chosen for Overlord. If the user needs to 'shard' to continue operating the bot (which is something Discord requires over 2,500 guilds!), they can simply duplicate the existing container or create a new instance via the process manager. In conclusion, although choosing ENMAP over the other solutions does trade robustness and scalability for performance and ease-of-use, this is a trade-in this use case that is worth making.

2.4.4 COMMANDS

Commands are specialised functions that a user can directly invoke through the bot via a message with a specialised prefix – the usage of a prefix allows for the bot to discriminate between regular conversation and a command – the default prefix for Overlord is "$", and is planned to be able to be changed on a per-guild basis.

The prefix is followed by the name of a command – or an alias, which is a planned feature by which commands can have multiple 'names' that resolve to the same command. The command name/alias is then followed by arguments (or 'args') that are optional additional pieces of information that can be passed to a command for additional functionality. An example of this would be the help command, where no arguments can be supplied and it will provide a basic overview of the commands the user can execute. However, if an argument is given, the help command will try to resolve it as a command name/alias to provide more in-depth information about a specific command – providing information such as example usage, etc.

As commands are the main way of interacting with the bot, there had to be a set of systems governing if user x can run command y. after discussing a set of potential 'limiters' with my admin client group, we decided that Overlord should be planned to obey the following set of restrictions, placed in a hierarchy based on the 'scope', with the idea being why perform all these checks only to fail at a 'higher level' scope check (e.g. check if enabled in the guild but the bot can't process commands posted in that channel). This is part of the smaller performance optimisations that have made to the message event – as it is by far the most evoked event out of those Overlord uses.

Guild-wide blacklists: is the user/channel in the guild-wide blacklist?

Command existence: does the command exist?

Command enabled state: is the command enabled for this guild?

Command-wide user Blacklist: is the user in the blacklist?

Permissions: does the user have the required permissions?

Command-wide channel Blacklist: is this channel in the blacklist?

Cooldown: has the user used this command in the last x milliseconds?

If the user 'passes' each of these checks, the command is executed. If they fail, the failure state is returned to the user via reporting (see the UX section) and the command is not executed.

I was planning on initially opting for a different system for permission management for Overlord – that of users having permission "tiers" from 0 to 4, with 0 being no special permissions and 4 being the bot owner (able to execute dangerous commands such as eval). However, It was decided against this implementation as it was inflexible towards specific requirements, with having to add an override system based on user/role to achieve the desired functionality. Instead, the current system was decided on, where an array of discord's permissions flags are used in a command's config to check that a user can execute a command – as generally if a user is executing a command that requires x permission, the user executing it should also have x permission. In this model, the bot is simply an automated extension of the invoker/user – which is an idea that Is generally preferred as it is more intuitive. This model is also inherently a lot more robust and easily tweaked compared to the original model.

On the Discord side, permissions are primarily governed by 'roles' – specialised tags that offer specific permissions to a user as well as a way of grouping users. Overlord uses Roles as a medium by which specific punishments are administered, e.g. the application of a 'muted' role to a user that removes their ability (sic: permission) to post in the server. Overlord uses the Permissions given by roles to determine the role flags used in the aforementioned 'limiters' for commands.

2.4.5 PROJECT ARCHITECTURE

One of the main goals of Overlord is the ease of extensibility via modularity – there are quite a few ways to implement this, but how it has been planned to implement modularity is on a meta contextual level – that is, how the code is split out into other 'contexts' - (into files) rather than a logical split (functions in one larger file).

This means that for each event that is being planned to use, It will use a separate.JS file, the same will happen with commands, with the name of the file correlating with the name of the command or event, for example: ./commands/help.js is executed if the help command needs to be executed by the bot, and ./events/message.js is executed upon the bot receiving a message event from Discord. This also helps not just for debugging and resiliency (as code is inherently isolated, so one buggy/erroring section will not affect the entire system) but also for ease of navigation and addition for code, with a simple drag-and-drop for adding/removing functionality or changing default configuration values that can be done whilst the bot is active as long as the proper reloading commands are also utilised – this reloading is very useful for debugging without having to restart the bot.

In Overlord, as part of the modularity and configurability objectives, each guild has its own Object (keyed by the Guild's Unique ID) in the ENMAP Database that contains configuration and persistency data – all the data relating to a guild that Overlord keeps is stored here. When a guild is created, this object is created from a template and then filled with values from default configuration values from commands and modules. The Object created has a set schema, shown by Figure 8.

Figure 8 - Database entry - guild - schema

Each event and command file share a common base schema, which can be seen in Figure 8 as the instance schema as part of the commands object, as well as for the modules object. Creating a copy of the config means that guilds can have their own discrete configuration – only referencing the default if something goes wrong or if no configuration exists.

This schema is by no means exhaustive, as modules can have specific keys due to specialised functionality but outlines the data types and overall patterns.

Due to the modularity goals as well as usage of Discord's API, understanding and visualising the data structures/architecture of systems I would be using was essential. To facilitate this, I researched the data structures, which can be summarised as follows:

Discord is composed of two main data structures, guilds (aka Discord servers) and users. The basic outline of both of these data structures, as well as some basic properties, can be seen in Figure 9.

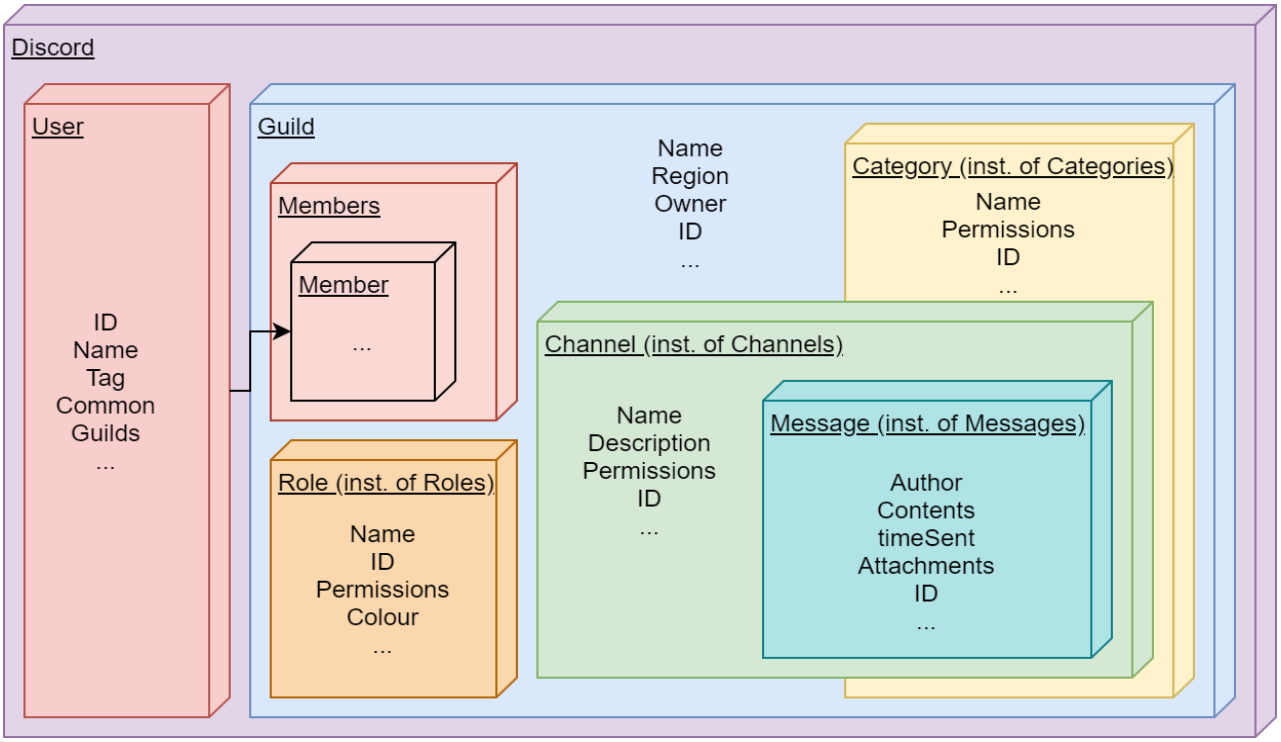

Figure 9 - Parent-child diagram of notable data structures

Note that channels can either be children of categories or the Guild directly. Inst. = instance

I generated this diagram from the Discord.JS documentation, and whilst it only contains shortened versions of object metadata, it helps to visualise the structures I will be working with throughout the course of development.

Every data structure in Discord has a unique identifier created by Discord and used to reference these data structures. Each ID is a string of numbers that can be used to retrieve/reference data structures. These generated IDs solve a problem that was anticipated with creating suitable primary keys for data storage and referencing.

An example of a reference to a message by ID could be guildID:channelID:messageID, so 558283449234:7793292273:104482238424. Using this string, Overlord would be able to determine the exact guild, channel, and message being referenced. This can be done with many other objects also.

The main data structure that Overlord will interact with is undoubtedly the message object. This object is what the client receives whenever a message is sent by a user and contains a few key attributes – notably references to the content of the message, the guild, channel, its author, any attachments, any reactions it has, a history of edits - among other attributes. Using these attributes, Overlord can easily contextualise a message to within a specific guild/channel for the purposes of per-guild configuration.

As previously stated, Discord bots work as fake users – receiving all the information that a real user would indirectly receive. Every time something happens, it triggers an 'event' in the Discord.JS client, where the contents of the event packet(s) are passed to the bot to process as required. Events can be triggered due to any changes to any of the elements outlined in Figure 9 – new messages, deleted messages, leaving and joining users, role changes, etc. The main events Overlord will be processing (followed by the module using the information) are:

- guildBanAdd – emitted whenever a user is banned in a guild - audit
- guildBanRemove – emitted whenever a user is unbanned in a guild - audit
- guildCreate – whenever the bot joins a new guild - core
- guildMemberAdd – whenever a member joins a guild - core

- guildMemberRemove – whenever a member leaves a guild – core
- guildUpdate – whenever a guild updates- e.g. settings change – audit
- message – whenever a message is sent by a member in a guild - core
- messageDelete – whenever a message the bot has cached is deleted by a user - audit
- messageReactionAdd – whenever a message the bot has cached has a reaction added to it
- messageUpdate – whenever a message is modified by the author – e.g. contents edited - audit
- roleCreate – whenever a role is created in a guild - audit
- roleDelete – whenever a role is removed in a guild - audit
- roleUpdate – whenever a role is modified in a guild - audit
- error – emitted when a WS/API error occurs, e.g. a disconnect – core

In Overlord, any code that directly handles an event is called a ‘module’.

Some of these events may not end up being utilised and may just be left as ‘stubs’ for future extensions depending on the developmental priorities.

The aforementioned Discord.JS client is the primary way the bot interfaces with the Discord API, and for the purposes of Overlord, has become a ‘global’ variable - as it is required for almost every function of the bot. This client contains all the information, functions and interfaces needed to create a Discord bot.

2.4.6 AUTOMATED CONTENT MODERATION

In order to produce a set of systems able to successfully moderate and filter the User Generated Content (UGC) produced in Discord servers, you first have to look at what the content is, as well as what elements of a piece of content that makes the difference between a rule infringement and acceptable.

With my preliminary studies, done with data compiled from the Admin Client group, the general common factors a piece of content has to have one or more of to be prohibited in most servers are:

Be NSFW (Not Safe For Work) in nature (provided the content is not in a marked NSFW channel)

Contain unwarranted levels of ‘toxicity’ - e.g. excessive swearing, racial slurs, etc.

Be part of a set of messages that have been sent in very rapid succession (“spam”)

Excessive use of capital letters (“Caps spam”)

Excessive use or repetition of specific characters (“Character spam”)

There are some more sets of identified criteria, but these are the general ones that apply to almost every server that I had access to via the admin client group, as well as personal experience with rules in other servers. These are the types of patterns that Overlord is planned to be able to recognise, with the first two (NSFW + Toxicity) being much harder, in theory, to implement via more traditional heuristics, unlike the last three.

2.4.7 MACHINE LEARNING SUBSYSTEM

One of the main ways in which It has been planned to offload as much of the administrative workload from humans to something automated (such as Overlord) was by implementing Machine Learning, specifically in regards to content recognition. The two areas of UGC that needs to be focused on is the content and context of the messages themselves, as well as any attached media, e.g. pictures and videos. With the recent titanic advances in making ML much more accessible and implementable over the past few years, I decided to see what I could do to implement this technology into Overlord.

Initially, I had planned to train my own NNs, one of each of the content “types” (text and media) while leaving context to a more simplistic heuristics system. I initially thought I had to train my own NNs since, for text specifically, identifying the specific types of content to test for as well as acquiring large amounts of normalised

test data for something as audience-specific as Discord would almost certainly not have been compiled into an existing Neural Network already. Thankfully, I was wrong, and the developers of TensorFlow, my ML library of choice, had just recently released a pre-trained NN designed for the specific purpose of detecting 'toxicity' in online chats. This almost perfectly fit my requirements for a NN capable of detecting the type of content in text that would, for the most part, be against the rules list of most Discord guilds. I was initially rather sceptical, as the performance of a NN is essentially directly related to its training time as well as the quality and variety of its training data, but in this case, the 2 million + messages, a summary of the methods used to collate this data set, and many, many days of reported training time put those worries to bed.

Similarly, for media, the primary types of content that were planned to be identified were the types most banned by Discord servers: NSFW (Not Safe For Work) content. This, naturally, gave me quite a headache as I did not want to have to train a model dealing with such content. Thankfully, the team over at Infinite Red trained a model into a module aptly called NSFW.js, that does what I was planning to make my model do – it takes input media and classifies it against a range of classes of types of content. I was, again, sceptical about the accuracy of this NN, but as demonstrated by Figure 10, the accuracy of the NN is very high – seemingly much higher than the system used by Discord itself as part of its content scanning system; whilst also being far more configurable.

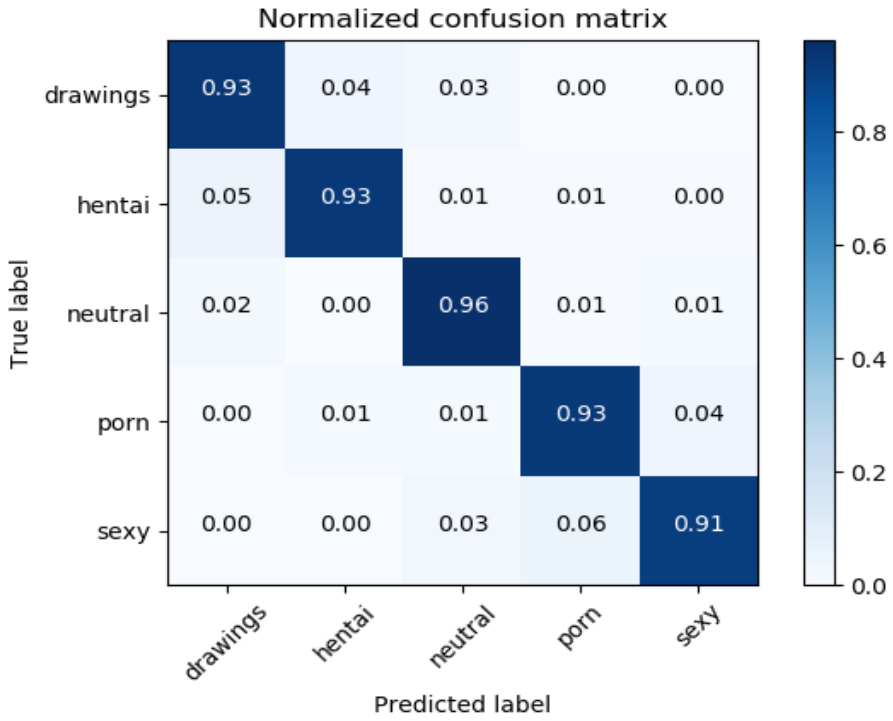

Figure 10 - Neural Network Confusion Matrix

Graphical representation of the confusion (classification overlap/certainty) in the NSFW Neural Network.

Initial testing of the toxicity NN also indicates a high level of classification accuracy, after being provided with data by the admin portion of the client group. Provided that the resource overhead incurred via the usage of these NN's is not too extreme, especially for the Toxicity Classifier, then the usage of both the NSFW

Classifier, as well as the Toxicity Classifier in Overlord, would be a major load reduction on moderators, who would otherwise have to manually review all this content.

Initially, I was also evaluating a system where content that broke a rule that was not detected by the NN, was passed through to the NN in a training cycle to adjust it so it would be able to detect such content in the future. This leads to a problem, however, with each guild needing a unique variant of the NN, as otherwise, it would

get ‘contaminated’ by the training data from other guilds. This would become problematic rather quickly, as each model needs to remain in memory at all times to minimise classification delay and having a unique NN for each guild would ramp up memory demands very quickly (each model is 200+MB). As such, It was decided to use a single NN for all the guilds, but after doing some testing with a standalone system, have discovered that the ‘weighting’ for each classification type can be altered to change how the system classifies content. This should, in theory, resolve this issue whilst still providing some customisability to the behaviour of the NN.

2.4.8 HEURISTIC MODULE

For the other section of automatic content moderation, I looked into some more conventional heuristics methods used either by other Discord bots, or other systems entirely. These methods cover the last three required factors - Spam, Character and Caps flood.

Spam detection could be implemented in a variety of ways, but the method I’m probably going to use is that of an array of ID’s, with an ID correlating with a user being added to the array every time they send a message in the channel that the array is keyed to (present in a larger object representing the guild). Then, after a certain amount of time, the added ID is removed from the array. If an ID is added and the number of instances of that ID in a channel-specific array exceeds a specified count, the user is ‘spamming’.

Similarly, for Character and Caps, a key:value system for character occurrences could be used, e.g. the word “Hello” => {“H”:1,”e”:1,”l”:2,”o”:1}, much like something similar to Run Length Encoding. If the counts of a character exceed a specified threshold, e.g. if 50% of a message is just one character, they could be classified as Character/Caps flooding.

These were the approaches that were decided to be used after evaluating the source code of some existing solutions, as well as the results from my own preliminary testing/investigations.

2.4.9 PUNISHMENTS

One of the areas that is often overlooked when developing an automatic moderation system is the algorithm that governs the punishments that are meted out upon a rule break occurring. After performing some evaluation of the systems commonly used, most of these are ‘unreactive’, e.g. you get a pre-set punishment x for breaking rule y. However, after talking to my admin client group about any particularly good implementations of punishment systems they have seen, one of them recommended the use of a reactive points-based system. After refining the general idea, the approach that has been chosen to go with is such that each punishment ‘tier’ would have a threshold value of points, once that value is exceeded, the punishment is enacted until the points ‘decay’ to below that specific threshold (or another specialised threshold, e.g. until points go to 0). The behaviour of this can be modelled as shown in Figure 11:

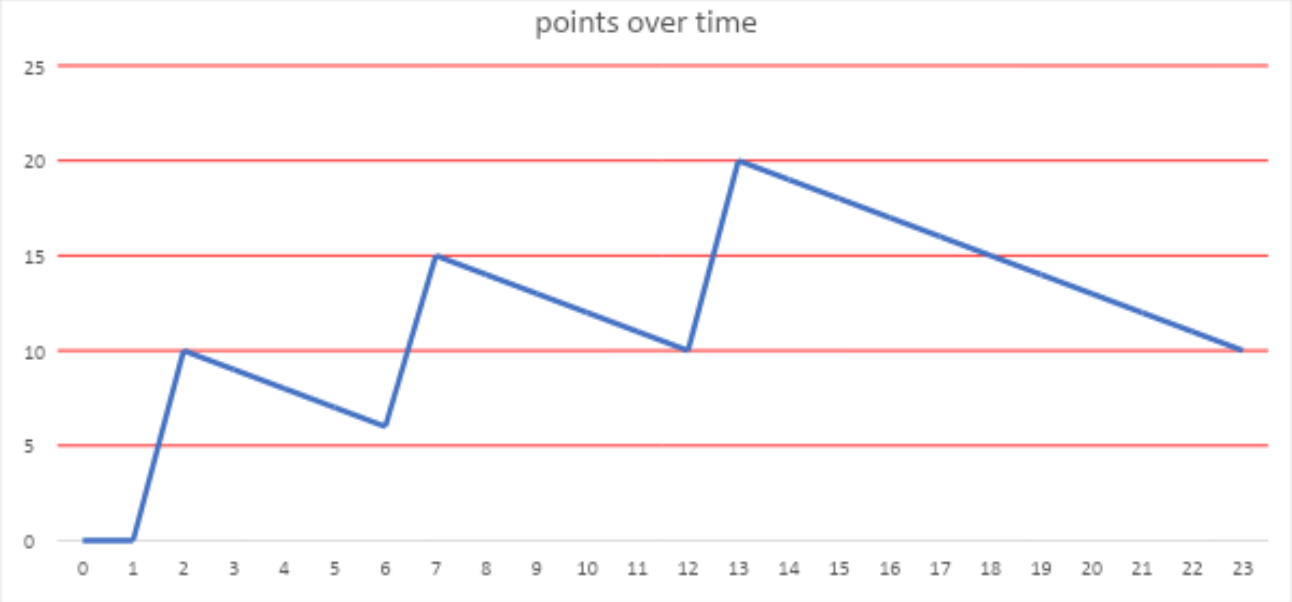

Figure 11 - Graph representing a model of the points system

In the above graph, the decay of 1 point/hour can be seen – each of the 'spikes' at 1 hour, 6 hours and 12 hours are simulated rule breaks of differing severity. If the threshold for a 'mute' is 10 points, the user is muted for an hour (hour 2 to hour 3) and then again at 7 hours until just after 23 hours. if a 'temporary ban' were to be a punishment for 20 points, with an 'end' boundary at 15, the user would be temporarily banned from 13 hours onwards until their points decayed below 15 (at 18 hours), at which point they would still be under the effects of the muted punishment.

This system would let me implement a lot of fine-tuneable behaviours in the future, such as a multiplier of points for repeat punishments, as well as allowing for the Admin client group to also fine-tune how punishments are handled in their guild, and as such, despite being more complex than a traditional punishment x for infraction y, it is a lot more versatile as well as extensible, and it has been determined that the additional effort towards developing this system would be worth it, as it would better achieve the performance criteria for this system.

As for the technical specifics of how this would be implemented, the current idea is to have a points and timestamp value attached to each user in the database, and after a configurable interval, the points decay and the count on the user is reduced and any changes in punishment is calculated. This will require a custom scheduler system to enable persistency.

2.4.10 AUDIT SUBSYSTEM

Another key way Discord servers are moderated is tied to a system known as the 'audit log' – it logs certain administrative actions that occur within the server, so that administrators can know who changed what and where, in case a change needs to be reversed, for example. The audit log only logs certain actions, however – such as it will only log that a message has been deleted if the user deleting the message is not the message's author, as well as not logging message edits at all – this poses a problem wherein users can post content and then remove it, dodging the potential ramifications of a punishment whilst still giving it limited exposure. Not to mention message edits, with no way to see the original on the client-side, message edits can also muddy the water so to speak about what was *actually* said. Although these measures are intentional on the part of Discord as part of their privacy-focused mandate, this system is not conducive towards moderation efforts. Fortunately, Discord provides events that signal the editing or deletion of a message, and Discord.js provides both the old and new (for edits) or just the old message (for deletions), which can be used to log any message 'changes' (edits/deletion). Using these, a system could be made where when a change occurs, the changes are posted to a specialised audit-log channel (different from the built-in audit log!) where it is available for moderator reference if required. This way, all changes can be logged.

In addition, the bot will also log any administrative actions it performs (as these are mostly not recorded in Discord's audit log), and adding a system by which other added modules will be able to access this logging interface to report the bot's actions is also planned.

2.4.11 HUMAN INTEGRATION

One of the main aspects I do not want Overlord to have is a lack of human input on moderation – primarily on the results from the automatic moderation systems. As such, I've been evaluating various ways in which Overlord can allow moderators to interact with itself as part of its moderation process. So far, the best way I have theorised to do this is by having Moderators accept/reject automatic moderation actions – e.g. if a user posts content that the automatic system (Overlord) deems outside acceptable parameters, the Moderators can determine if the suggested automated action should be executed, or not. This could be done through specialised/generalised actions that are sent to a moderator-only channel, that moderators then react to signal approval/disapproval. The actions will be in the forms of embeds and will contain links to detected content, as well as important information about the action – enough for the moderator to make a confident decision at ideally just a glance.

2.4.12 USER EXPERIENCE (UX)

User experience is one of the core ways in which a Discord bot can be made or broken, with my research showing that the ease of use factor (which is heavily used to UX) is *the* major factor between a successful and unsuccessful Discord bot. As such, It was decided to put more time into developing the best UX I can devise given the constraints of interaction due to Discord as a platform. To do this, I first looked at other existing solutions, to see if there are any particularly notable methods/examples of good UX design - whilst remaining entirely within the Discord app. One of the more notable systems as part of this investigation that I developed was centred around the idea of 'optional reporting' concerning commands. In this example, a user would attempt to execute a command, which would fail for some reason. The bot would then 'react' to the command message (a reaction is a specialised 'response' to a message in the form of a specific emoji – see the below figure) and provide feedback to the user via direct messaging (DM), so as to not clutter the server channel the command was sent from. By doing this, it shows the user that there's an issue, without simply ignoring the incorrect command, and provides optional and discrete assistance to the user without cluttering the channel the command was sent in.

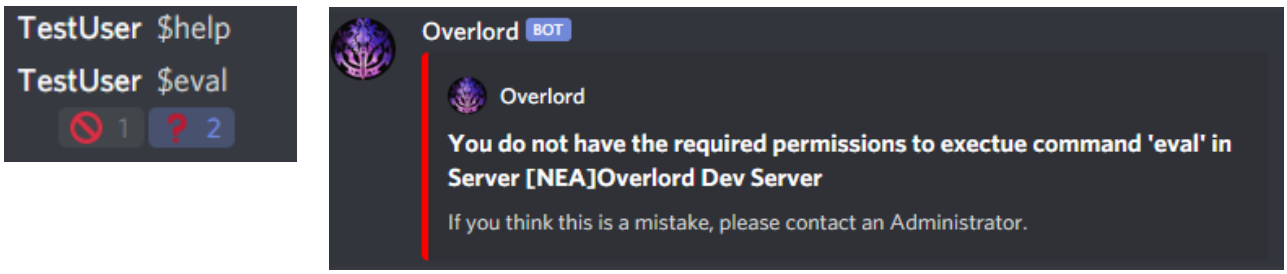

Figure 12 - Demonstration of Error Reporting UX

Figure 12 shows a user using two commands – one they are allowed to use(help), the other they are not allowed to(eval) – you can also see the intended usage of a prefix (in this case a"$") to discriminate between normal conversation and wanting to execute a command. When the user executes the command they do not have the permissions for, the bot reacts with two emojis: 🚫, which signifies the execution of the command has 'stopped' and ❔, which when also reacted to by the user with a simple click/tap, is detected by the bot and interpreted as a query for 'why the failure occurred' – which prompts the bot to send a message correlating to the failure state of the command to the user via DMs.

Another core feature of UX was to develop a system that allows for information to be presented clearly, even where there is a lot of information that could be present. The primary way to accomplish this in Discord is via 'embeds' – specialised messages that can contain multiple media elements and fields. For example:

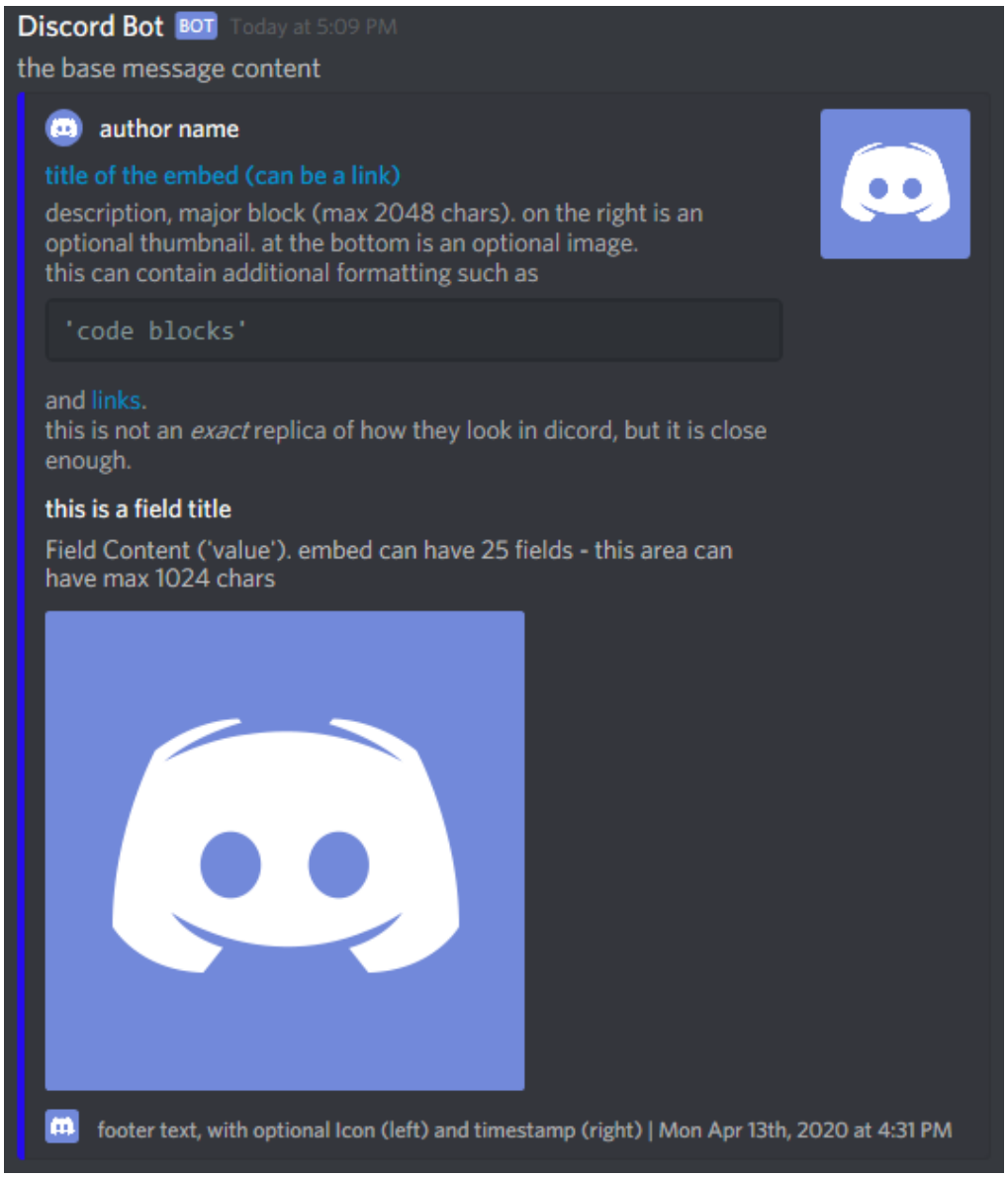

Figure 13 - Demonstration of Possible Embed Attributes

Embeds can contain a lot of data (25 fields of up to 1024 chars each) and are very flexible (no set layout – all controllable by the composer) – this means that they are ideal information delivery mechanisms for data within the context of discord. An example of this is the aforementioned planned 'actions' system that would use embeds to allow moderators to make streamlined administrative decisions.

Another theorised application of this UX strategy was for Overlord to detect mentions of itself, and respond to them as if the user had used the help command – this would give users who had no idea how the bot worked a way of figuring it out. This can be coupled with the use of a 'status' – a small string of text present in a user's profile – that could read something like "@ me for help!" or something equivalent as a means of prompting this behaviour.

So far, the chosen systems appear to be optimal for their respective tasks per the development objectives for Overlord and will be the systems implemented moving forwards in my technical implementation of Overlord. With the Conceptual foundations now in place, I began by creating a model of the execution flow both in

Overlord as a whole as well as through the most important of the myriad of events: the message event. The results of this can be seen in the following figures, showing visual diagrammatic representations of each of the flows in these two core systems (overall, then message):

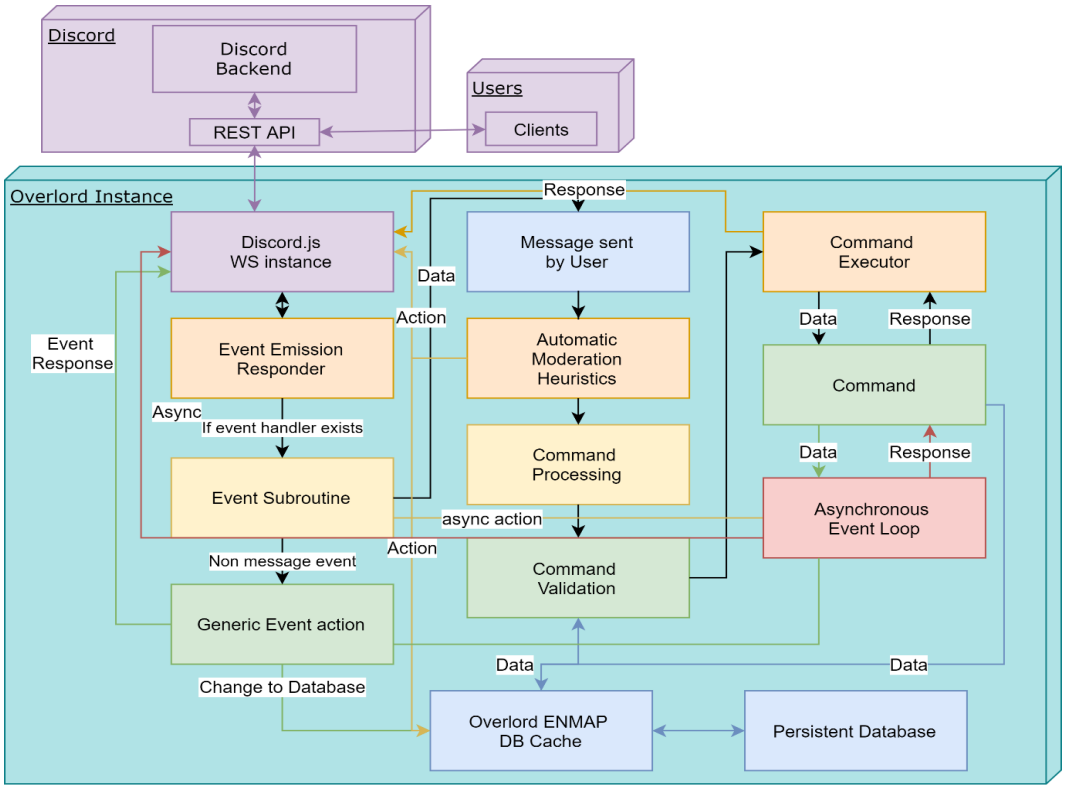

Figure 14 - Overlord Overarching Process flow

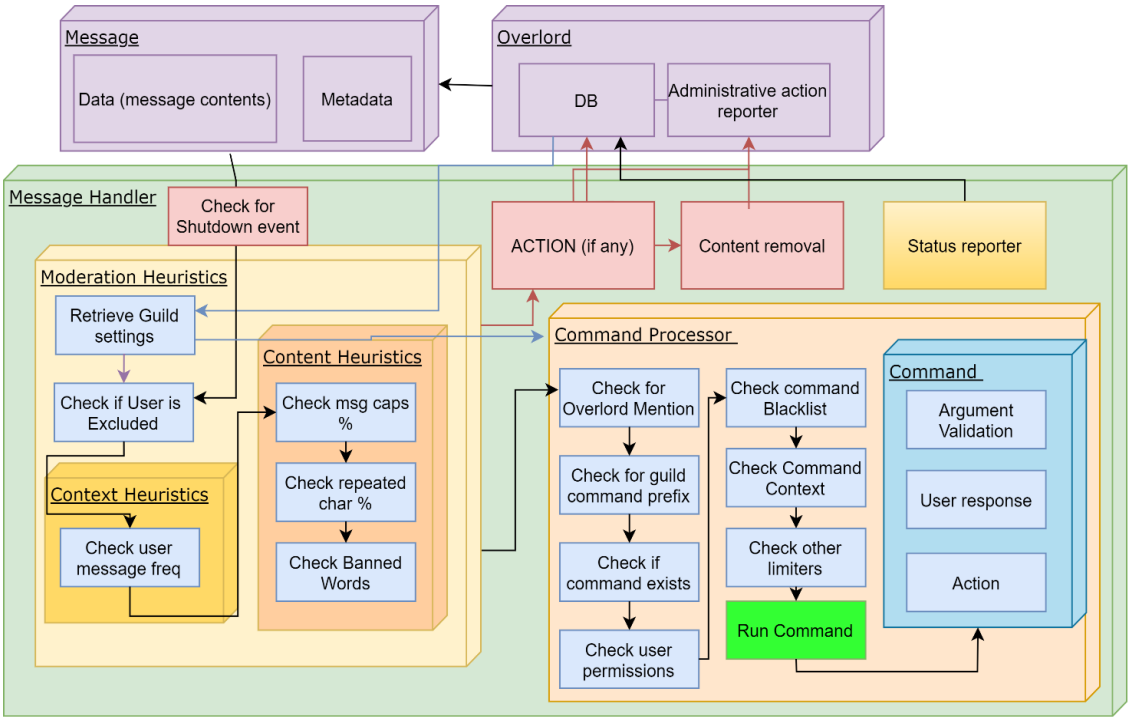

Figure 15 - Message event Overarching Process Flow

2.5 TECHNICAL DEVELOPMENT CRITERIA

These objectives are the overall goals each prototype will, in part, be fulfilling - with the goal of all these requirements being satisfied by the end of development.

Objective	Performance Criteria
Authentication and API interaction with Discord	Overlord can login using a specified token, and can send and receive events to/from Discord to accomplish actions (e.g. sending messages)
Modularity	Overlord should, upon a configurable trigger, execute a specified file in a specified directory with relevant data – both for client events and commands. the bot should be able to allow for and respect the instanced configuration of these files within the database, as well as being able to deal with the addition/removal of files between restarts. This allows the bot to create as many asynchronous threads as required to scale with processing demands.
Uptime/performance	Overlord should, with use of a 3rd party application manager, restart itself if (and only if) the application is in an unstable state. The bot should ensure that all critical data is stored within a persistent solution and that the bot is otherwise resilient towards errors, with error handling techniques such as try..catch being used throughout. Overlord should prioritise performance heavily, especially for more commonly utilised execution paths – and as such have toggles for more resource-intensive operations.
UX	Usage of embeds as opposed to regular messages wherever possible due to the clarity of conveying larger amounts of information. Integration of theorised UX techniques laid out in 2.3.12, as well as techniques to better help unfamiliar users understand how to use Overlord
Automatic Moderation	Overlord should, on a configurable basis, discriminate and classify content based on content (media -NSFW and text – Toxicity, char, caps) as well as context (spam). Once the content is classified, a recommended action should be suggested and, on a configurable basis, this action should be processed automatically. Otherwise, a moderator needs to accept this action. Once the action has been approved, the action needs to handled depending on its type and other requirements. Any actions taken by the bot should be reported and affected users notified. Overlord should also offer tertiary functionality to better aid in moderation efforts
Configurability	Overlord should, at start up, ensure that all present commands, modules, guilds and users have the proper configurations present for operation, creating new entries as/when required form default values. Under no circumstances should existing/modified configuration be overwritten by default values. Upon command/module execution, functionality should be beholden to the configuration for this functionality contained within the database entry for the guild this command/module.

Table 2 - Technical Development Objectives & Criteria

.

3 PROGRAMMING

The main 'index' file of Overlord is aptly named Overlord.js. generally, unless otherwise stated, code is being written to this file.

Development log:

This log is a record of my development of Overlord, split into different prototypes, each focusing on implementing a specific function or functionality outlined above. Each prototype builds upon its predecessors to create the final product. Each prototype is individually tested, with the entire bot being tested after further refinements later on – the finalised codebase can be found in Section 4 – Appendix.

The chosen testing methodology for this project is primarily grey box testing, as well as some testing via end-users (client group). grey box (a mix of white and black box) was chosen as both the internal functionality (white box) as well as it's overall functionality (black box) are oftentimes tested at the same time as most of this code directly feeds back to a user in a way that is far less common in larger applications due to its nature as a Discord bot – with black box being done primarily by my end-user group. The decision to not use module tests is due to what Is perceived as limited usefulness for a system that has to deal with a wide range of input data from users, testing it on a set of fixed inputs does not align well with its intended input data. It is for this reason that It was also decided to, for specific features as well as a 'final product' test, use end-user testing to expose the bot to the uncontrolled environment it will be operating within to find any bugs that escaped my testing within a controlled environment. It is believed that with these testing strategies – as long as the testing is thorough – the number of bugs within the production version should be minimal.

Testing will be reported using the 'minimalist' testing table, denoting the input, output, expected output, as well as any bugs/comments. This is because all tests are for the same purpose – ensure whatever function/procedure being implemented complies with the requirements set out for it. For the full solution tests however, a more extensive table will be used due to the greatly increased variety and number of tests.

Note: For functions processing data from Discord directly, any form of input validation can be said to be unnecessary due to the already validated status of the input (both by Discord and the client itself) – this is why only functions with 'novel' inputs have input data tests.

3.1 PROTOTYPE 1 – BASIC FUNCTIONALITY

Goals: Basic message response functionality and client initialisation

This prototype was to get to grips with the API and data structures – very basic in terms of functionality – logs into the client before responding to any messages it receives with "Hello, World!"

Relevant code:

```
const discord = require("discord.js")
let client = new discord.client()
client.login("TOKEN") //login via token
client.on("ready", () => { //once the client has logged in
 console.log("ready!")
})
client.on("message", (message) => { //respond to a message
 if (message.author.bot) return
 message.channel.send("Hello, World!")
})
```

Input	Output	Expected	Bug
Generic message	Infinite loop of “Hello, world!” outputs	A single response of “Hello, World!”	Doesn’t ignore posts from bots (and thus itself)
Generic message	“Hello, World!”	“Hello, World!”	none

Table 3 - Prototype 1 Testing Table

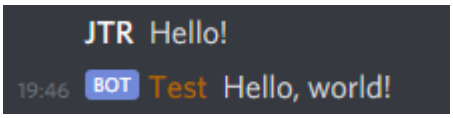

Figure 16 - Picture of Call-Response Behaviour

Comments:

Successfully logged in to the Discord API but would respond to itself. Made it ignore messages by bots to fix this issue.

3.2 PROTOTYPE 2 – EMBED FUNCTIONALITY

Goals: test embed functionality

This prototype was used for testing of embed functionality, a key part of my planned UX approach.

Relevant code:

```
client.on("message",(message)=>{ //respond to a message
 let embed = new discord.RichEmbed() //create a new embed
 .setTitle("test") //add elements
 .addField("field test")
 .setFooter("test")
 .setDescription("description")
  message.channel.send("Hello, World!",{embed: embed}) //send the embed
})
```

Input	Output	Expected	Bug
Generic message	Message with embed	Message with embed	none

Table 4 - Prototype 2 Testing Table

Comments:

No issues with this prototype, worked as expected, creating an embed with all the relevant fields before sending it.

3.3 PROTOTYPE 3 – MODULAR EVENT CODE

Goals: modular code for events

Create functionality so that when a client event happens, an event file is executed to process that event.

Pseudocode:

Iterate over every file in ./events

If the file doesn't end in .js, return

Set name as the name of the file, removing .js.

Whenever the event "name" is triggered, run file.

Relevant code:

```
const fs = require("fs")
const eventFiles = fs.readdirSync("./events/"); //load the files
eventFiles.forEach(eventFile => {
 //skip anything that's not a .js file
 if (!eventFile.endsWith(".js")) return;
 //strip the file extension
 const eventName = eventFile.split(".")[0];
 const eventObj = require(`./events/${eventFile}`); //load it
 //bind it to the event as well as the client variable
 client.on(eventName, eventObj.bind(null, client));
})
message.js:
module.exports = (client, message) => {
 //code from previous prototype here
}
```

Input	Output	Expected	Bug
bot run with message.js file in events	Message.js file run	Message.js file run	none

Table 5 - Prototype 3 Testing Table

Comments:

Required a lot of iteration and research due to the apparent difficulties with ensuring all the data of an event is passed to the file - eventually discovering the bind method that allows for functions to be invoked within a specific context therefore resolving this issue. Tested by moving the message into a separate file. This worked as expected, with moving the client.on 'message' code to a separate file functioning correctly.

3.4 PROTOTYPE 4 – ENMAP IMPLEMENTATION

Goals: implement ENMAP database

Implement the primary persistent ENMAP database for the bot. test functionality and implementation. This is the primary persistent datastore for the bot for all guilds.

```
const enmap = require("enmap")
```

```
client.DB = new enmap({
 name: "DB"
})
client.DB.defer.then(
 console.log("ready!")
 client.DB.set("12345", "test")
 client.DB.get("12345") // returns stored value : "test"
)
```

Input	Output	Expected	Bug
Set “1234” to “test”	“test” stored at key “1234”	“test” stored at key “1234”	none
Get the value of key “1234”	“test”	“test”	none

Table 6 - Prototype 4 Testing Table

Comments:

This prototype worked as expected, saving the data into the default save directory (./data) after the enmap was given a name, as well as setting and getting values working as expected. there were some initial problems however with the database not being ‘ready’. This meant the defer statement had to be used to wait for the database to be ready before performing any operations on it. It also became clear that client was going to be an object required by almost everything in the bot, so It was decided to append the enmap database as a new property, as it too was going to be required throughout the bot.

3.5 PROTOTYPE 5 – DATABASE INITIALISATION

Goals: add required data to the database for users and guilds

Pseudocode:

For every guild

In client.DB, ensure guild.id is set to defaultConfig

For every member of guild

In client.DB.(guild.id) ensure member.id is set to {xp:0}

for every file in ./commands

in client.DB ensure file.defaultConfig in commands

Relevant code:

```
let defaultConfig = {
 prefix: "$",
 adminRoles: [],
 modRoles: [],
 serverOwnerID: 0,
 mutedRole: 0,
```

```
 welcomeMsg: "Welcome {{user}} to {{guildName}}!",
 welcomeChan: 0,
 blockedChannels: [],
 modActionChan: 0,
 modReportingChan: 0,
 auditLogChan: 0,
 modules: {},
 commands: {},
 users: {},
 persistence: {
  messages: {},
  attachments: {},
  time: [],
 },
 blacklist: {}
};
//iterates over every guild
client.guilds.forEach(guild => {
 //if a value exists, do nothing, if a value doesn't set it to the default
 client.DB.ensure(guild.id, defaultConfig)
 //iterate over all the guild members
 guild.members.forEach(member => {
  //if they exist, do nothing else crate them with xp set to 0.
  client.DB.ensure(member.id, { xp: 0 }, guild.id)
 })
})
```

Input	Output	Expected	Bug
Ensure guild as default with no data	Guild set to default	Guild set to default	none
Ensure guild as default with data	Guild data unchanged	Guild data unchanged	None
Ensure member with no data	Member set to provided object	Member set to provided object	None
Ensure member with data	Member data unchanged	Member data unchanged	None

Table 7 - Prototype 5 Testing Table

Comments:

I was initially planning to just use set to setup the guilds, but when I did some testing it shows the ‘set’ method overwrote any values already at the specified ‘key’ (in this case guild.id, e.g. 139456273450). So I had to come up with a solution that only set the value if the value didn’t exist. Thankfully, after reading the documentation

for ENMAP a little more, a function known as ensure implemented this kind of behaviour. ‘Ensure’ returns the existing value, but if there is no value, sets it to a provided value. In this case, if the guild existed in the database, nothing would change, but if it didn’t exist, the guild would be set as the defaultConfig Object. ‘Ensure’ is going to be very useful as it is a very common pattern within Overlord. Command configs are also ensured for each guild, as these configs are per-guild. I checked that the initialisation was operating correctly via debug queries of the database.

3.6 PROTOTYPE 6 – REQUIRED PERMISSIONS

Goals: Determine the Permissions the bot requires from module configs. Also, initialise some companion non-persistent ENMAPs.

Pseudocode:

Set client.cooldown, client.trecent = new enmap

For every guild,

Set reqPermissions to ["SEND_MESSAGES", "READ_MESSAGES", "VIEW_CHANNEL"]

For every module in Client.DB.(guild.id).modules;

If module has requiredPermissions

For every permission in module.requiredPermissions,

If reqPermissions does not have permission, add permission to reqPermissions

set missingPerms to Filter permissions the bot has in guild against reqPermissions

Relevant code:

```
let client.cooldown = new enmap()
let client.trecent = new enmap()
//check module permission requirements to determine the permissions required by the bot. starts with a basic set:
let reqPermissions = ["SEND_MESSAGES", "READ_MESSAGES", "VIEW_CHANNEL"]
Object.keys(guildData.modules).forEach(key => { //iterates over every module, checking the permissions declared as required.
 let Module = guildData.modules[key]
 if (!Module.requiredPermissions) return;
 Module.requiredPermissions.forEach(perm => { //append missing perms to the array.
  if (!reqPermissions.includes(perm)) { reqPermissions.push(perm) }
 })
});
let missingPerms = reqPermissions.filter(perm => !(guild.members.get(client.user.id).permissions.toArray()).includes(perm))
```

Input	Output	Expected	Bug

Event File with no permissions set	reqPermissions does not change	reqPermissions does not change	None
Event File with permissions set	reqPermissions changed with new permissions	reqPermissions changed with new permissions	None
No permissions missing	None	None	None
Permissions missing	Missing permissions	Missing permissions	None

Table 8 - Prototype 6 Testing Table

Comments:

This algorithm works at expected, producing an array of permissions that the bot is missing in a server. This would allow for admins to give the bot the permissions it needs before any issues arise due to a lack of permissions. Tested this with a dummy event that had permissions the bot did not have. Logged the missing permissions as expected.

Once the reporting system is done, I will integrate this functionality to report the missing permissions.

3.7 PROTOTYPE 7 – CUSTOM LOGGING SOLUTION

Goals: implement custom logging solution, as well as debug flag.

This logging system means I can still have debugging information when required, but otherwise keep it out of the logs unless it's tagged otherwise. Higher 'urgency' level also get differing console formatting, e.g. error gets the ominous red text.

Pseudocode

Set client.debug to true unless the process started with environment variable "Production"

Define function log (message, type)

set caller to stack Trace[indexOfLogFunctionCaller]

if no type exists, set type as "DEBUG"

set message to type+ stringify(message) + caller

switch type

case "ERROR" console.error

case "WARN" console.warn

case "INFO" console.log

case "DEBUG"

if client.debug is false, return

else console.log

if no case matches, console.log error.

Relevant code:

```
//ternary operator for determining debug state
```

```
client.debug = (process.env.NODE_ENV === "production" ? false : true)
/**
  * custom logging system - uses optional tags as well as stack tracing to determine caller locations for ease of access.
  * 4 'levels' - untagged - ignored if not in debug mode
  * INFO - information, always displayed
  * WARN - warning, always displayed
  * ERROR - Error - something has gone wrong. always displayed.
  * helpful as it allows the owner to differentiate types of information easily.
  */
 client.log = (message, type) => {

//using stacktrace to locate caller code/function. useful for determining exactly where information came from.

let caller = ((new Error).stack).split(" at ")[2].trim().replace(client.basedir, ".")
)
  //message - comprised of the type and stringified message, as well as the caller.
  //each type/flag gets different 'treatment'
  if (!type) type = "DEBUG"
  //formats the message with some addition info.
  let msg = `[${type}] ${(JSON.stringify(message)).replace(/"/g, "")} 〖${caller}〗 `
  switch (type) {
   case "ERROR":
    console.error(msg);
    break;
   case "WARN":
    console.warn(msg);
    break;
   case "INFO":
    console.log(msg)
    break;
   case "DEBUG":
    if (!client.debug) break
    console.log(msg)
    break
   default:
    console.log(`Invalid logging type! ${msg}`)
  }
 }
```

Input	Output	Expected	Bug

Message with type debug with debug set to true	Message logged with debug tag	Message logged with debug tag	None
Message with type debug with debug set to false	No logging	No logging	None
Message with type error set	Message logged with error tag	Message logged with error tag	None
Message with type warn set	Message logged with warn tag	Message logged with warn tag	None
Message set with type info	Message logged with log tag	Message logged with log tag	None

Table 9 - Prototype 7 Testing Table

Comments:

This system works very well, providing debugging information whilst running the bot in production mode, where a proper debugging environment is not available. The code properly identifies caller function/line and formats message into string-only format, replacing extraneous quotation marks as these are injected into the string with usage of JSON.stringify. Current and future logging statements have been updated to use this new code.

3.8 PROTOTYPE 8 – CONFIG.JS IMPLEMENTATION

Goals: implement config file

This file implements configuration options for the entire bot.

Pseudocode:

Config.js

Exports:{

ownerID: id of the owner/operator of the bot's Discord account.

Token: production and development token, selected inline

Datadir: the path of the directory where the database should be stored

Status: the string used for the bot's status – has placeholders for data.

Variable for enabling Neural Networks/models

Preload: variable for toggling preload functionality of messages and database keys.

Relevant code:

Config.js:

```
module.exports = {
 ownerID: "150693679500099584", //discord ID of the  bot's Owner.
 token:
  process.env.NODE_ENV === "production" //ternary operator for token selection
```

```
   ? "NTc2MzM3ODI2MTU3NjkwOAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA " //bot's
Production Token (true)
   : "NjQ4OTU5OTU5NDg4MAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
", //dev token (false)
 datadir: "./data", //data storage location for the ENMAP-SQLite backend.

status: "@ me for help – version {{version}} - now on {{guilds}} guilds!",//status m
essage of the bot.
};


//Overlord.js
client.config = require('./config.js')
```

Input	Output	Expected	Bug
Config file created	Config file read and ternary functioning	Config file read and ternary functioning	None

Table 10 - Prototype 8 Testing Table

Comments:

Using the 'require' terminology, the configuration file is able to be loaded and referenced. Due to the inline ternary operator, the token switches seamlessly between deployment environments. These configurations are for global configuration only, and so are comparatively sparse.

3.9 PROTOTYPE 9 – SHUTDOWN PROCEDURES & FATAL ERROR HANDLING

Goals: implement shutdown procedures as well as error handling.

Pseudocode:

Define function graceful shutdown: exit the process after 5 seconds of I/O inactivity – set client flag (isShuttingDown) to true. Log the type of shutdown.

upon process error, sigint, and shutdown message start a graceful shutdown. Message and sigint allows for PM2 integration,

upon client error, log the error. Upon client disconnection, log and then trigger graceful Shutdown.

Relevant code:

```
client.isShuttingDown = false
client.on("gracefulShutdown", (reason) => {

client.log(`Successfully Received Shutdown Request - Reason: ${reason} -  bot Proces
s commencing shutdown.`, "WARN");
 client.isShuttingDown = true

setTimeout(() => { setImmediate(() => { process.exit(0); }); }, 5500); //after 5.5 s
econds, and after all I/O activity has finished, quit the application.
})
```

```
/** PM2 SIGINT and Message handling for invoking a graceful shutdown through PM2 on both UNIX and windows systems */
process
 .on("SIGINT", () => {//unix SIGINT graceful PM2 app shutdown.
  client.emit("gracefulShutdown", "PM2")
 })
 .on("message", (msg) => {//Windows "message" graceful PM2 app shutdown.
  if (msg === "shutdown") {
   client.emit("gracefulShutdown", "PM2")
  }
 })
 .on("uncaughtException", (err) => {

console.dir(err.stack.replace(new RegExp(`${__dirname} / `, "g"), "./")); /** process error catching with custom stacktrace formatting for ease of reading */
  client.emit("gracefulShutdown", "Exception")
 });


/** catches and logs any Discord.js Client errors */
client
 .on("error", error => { client.log(error, "ERROR"); })
 /** if the client disconnects, report the disconnection */
 .on("disconnect", (event) => {
  client.log("Client disconnected! restarting...\n" + event, "ERROR")

client.emit("gracefulShutdown", "Disconnect"); /**this event signifies that the connection to discord cannot be re-established and will no longer be re-
attempted. so we restart the bot process to (hopefully) fix this (note: requires PM2 to restart the process).*/
 });
```

Input	Output	Expected	Bug
Emit gracefulShutdown event	GracefulShutdown executed successfully	GracefulShutdown executed successfully	None
Process SIGINT	GracefulShutdown executed successfully	GracefulShutdown executed successfully	None
Process Message	GracefulShutdown executed successfully	GracefulShutdown executed successfully	None
Uncaught exception	Exception logged, GracefulShutdown executed successfully	Exception logged, GracefulShutdown executed successfully	None

Client error	Exception logged, GracefulShutdown executed successfully	Exception logged, GracefulShutdown executed successfully	None
Client Disconnect	Exception logged, GracefulShutdown executed successfully	Exception logged, GracefulShutdown executed successfully	None

Table 11 - Prototype 9 Testing Table

Comments:

Initial testing of these systems shows that they are working as intended – both PM2 integration on UNIX-based and windows function fully, and the timeout with set immediate and the locking allows for a truly 'clean' shutdown. Initially, uncaught exceptions were not going to restart the bot, however, as explained earlier in this document, leaving the application in an 'unclean' state may prove problematic.

3.10 PROTOTYPE 10 – DISCORD RICH PRESENCE

Goals: Discord presence and initialisation notification

This is the implementation of a Discord status (or presence) which can be seen in Figure 17.

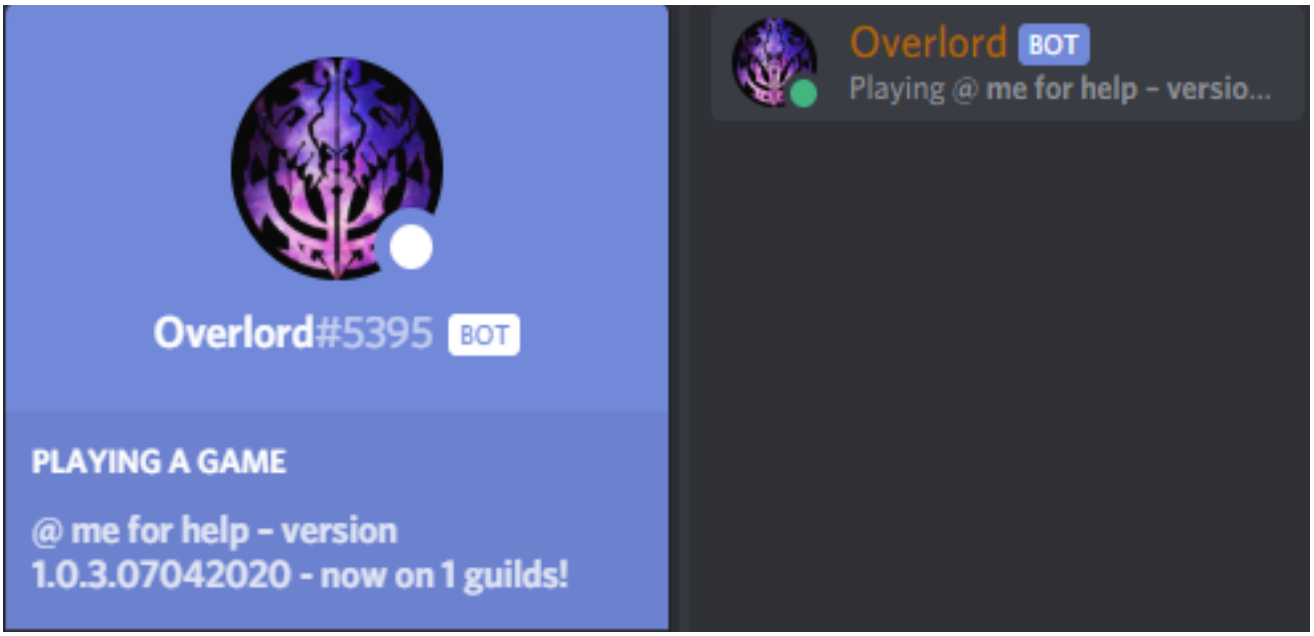

Figure 17 - Rich Presence within Discord

The initialisation notification is a message the bot sends to the bot owner via DM's upon starting up fully to notifying them that the bot has started up, as well as if the bot has restarted at all.

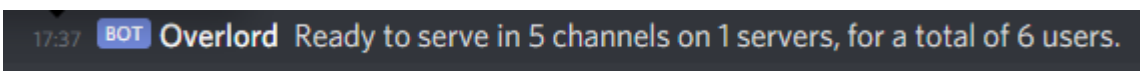

Figure 18 - Initialisation Message

Note: client.version is manually defined when I do updates. Present in the main file of the bot (Overlord.js)

Pseudocode:

Set client.status to

Client.config.status.replace “{{version}}” client.version, “{{guilds}}” client.guilds.size

Send client.config.ownerID “ready to serve in {client.channels} channels on {client.guilds} servers, for a total of {client.users} users”

Relevant code:

```
client.version = "0.0.3.25042019"; //release.major.minor.date
//status - small message that shows up under the profile of the bot. customisable in config.js.
var status = client.config.status.replace("{{guilds}}", client.guilds.size).replace("{{version}}", client.version);
client.user.setPresence({ activity: { name: status }, status: "active" });
```

Input	Output	Expected	Bug
bot loaded with config string	String placeholders replaced and status displayed	String placeholders replaced and status displayed	None

Table 12 - Prototype 10 Testing Table

Comments:

This code works as expected, replacing the placeholders as required for the status message.

3.11 PROTOTYPE 11 – IMPLEMENT CLIENT EVENTS

Goals: implement other events, e.g. messageDelete, Update, etc.

Implement the following events in a basic form (basic .js file with some logging code):

guildCreate

guildMemberAdd

guildMemberRemove

guildMemberUpdate

guildUpdate

messageDelete

messageReactionAdd

messageUpdate

Relevant Code:

guildCreate.js

```
module.exports = (client, guild) => {

client.log(`Client has joined guild ${guild.name}!`, "INFO");//log that a guild has been joined.
```

```
client.validateGuild(guild) //invokes a check for the  bot's DB to initialise the gu
ild in the database for instant usage.
}
```

guildMemberAdd.js

```
module.exports = async (client, member) => {
 let guild = member.guild;

client.log(`new member with Name ${member.displayName} has joined ${guild.name}`, "I
NFO"); //logs

let data = client.DB.ensure(guild.id, { xp: 0 }, `users.${member.id}`); //initialisi
es user to DB
}
```

guildMemberRemove.js

```
module.exports = (client, member) => {

client.log(`member ${member.displayname} has left guild ${member.guild.name}`, "INFO
")
}
```

guildMemberUpdate.js

```
module.exports = (client, oldMem, newMem) => {
 client.log(`member update: ${client.diff(oldMem, newMem)}`)
};
```

guildUpdate.js

```
module.exports = (client, oldSvr, newSvr) => {
 client.log(`Guild has been updated: ${client.diff(oldSvr, newSvr)}`);
};
```

messageDelete.js

```
module.exports = async (client, message) => {

client.log(`message with contents ${message.contents} deleted in ${message.guild.nam
e} ${message.channel} attachments: ${message.attachments} `)
}
```

messageReactionAdd.js

```
module.exports = async (client, messageReaction, user) => {

client.log(`reaction "${messageReaction.emoji.name}:${messageReaction.emoji.identifi
er}" added to message with ID ${messageReaction.message.id} by user ${user.id}`);
}
```

Overlord.js

```
client.diff = require("deep-object-diff").detailedDiff;
```

messageUpdate.js

```
module.exports = (client, Message, nMessage) => {
```

```
 if (Message.content === nMessage.content) {
  client.log("messageEdit invoked - message content identical - assume autoembed");
  //if contents are identical, autoembed was most likely triggered.
  //autoembeds are created by Discord to show media content in a message.
  return
 } else {

client.log(`message updated in ${Message.guild} ${Message.channel}. diff: ${client.diff(Message, nMessage)} `, "INFO")
 }
}
```

Input	Output	Expected	Bug
bot joins a new guild	Guild ensured and logged	Guild ensured and logged	None
Member joins a guild	Member is ensured and join is logged	Member is ensured and join is logged	None
Member leaves a guild	Member leave is logged	Member leave is logged	None
Member is updated	Member change is logged	Member change is logged	None
Message is deleted	Message is logged, attachments unavailable	Message is logged, attachments unavailable	Attachments are locked when message is deleted – no current resolution

Table 13 - Prototype 11 Testing Table

Comments:

Most of these events are barebones at the moment just to make sure they work correctly. I did notice a quirk that throws a spanner in the works with how Discord handles deleted message attachments. Any attachments on a deleted message become completely inaccessible once the message is deleted. This means that for attachment persistency, the bot has to have access to an external copy. This is something that I will have to keep in mind when I implement the full version of the message delete functionality.

3.12 PROTOTYPE 12 – COMMAND PROCESSING DISCRIMINATOR

Goals: command processing limiter backend

Create the functional backend for the limiters for command processing. Needs to adhere to the specification outlines earlier within this document.

Pseudocode:

Define function client.canExecute(client,message,cmdName) as;

Set cmdCfg to client.DB.get(message.guild).commands[cmdName]

If message.authour.id is equal to client.config.ownerID

Return “Passed”

Else if cmdname is undefined,

Return “nonexistant”

Else if cmdCfg.enabled is false,

Return “disabled”

Else if client.DB.get(guild.id).blacklist.cmdName has message.author.id

Return “blacklist”

Else if message.author.permissions doesn’t have cmdCfg.reqPermissions

Return “perms”

Else if cmdCfg.allowedChannels doesn’t have message.channel.id

Return “channel”

Else if client.cooldown has message.author.id

Return “cooldown”

Else

Return “passed”

Related code:

```
/**
  * given a command name as well as context (eg guild/channel) from the message obje
ct, determines
  * whether or not the given user can execute the command they are requesting to exe
cute.
  */

client.canExecute = (client, message, cmdName) => { //check if a user can execute a
command or not
  //gets the current guilds configuration from the message via the non-
standard settings attribute - added in message.js
  let cmdCfg = message.settings.commands[cmdName]

//checks lots of conditions to determine if the user can execute a command - reports
 what 'stage' they fail at, if any.
  //if the user is the owner, pass.
  if (message.author.id === client.config.ownerID) {
   return "passed"
  }
  //if the command doesn't exist, fail with error 'nonexistant'
  if (!cmdName) {
   return "nonexistant"
```

```
  }
  //if the command is disabled on a guild-wide basis, fail with 'disabled'
  else if (!cmdCfg.enabled) {
   return "disabled"
  }
  //if the user is in the guild's blacklist for the command, fail with 'blacklist'

else if (Object.keys(client.DB.ensure(message.guild.id, [], `blacklist.${cmdName}`))
.includes(message.member.id)) {
   return "blacklist"
  }

//if the user isn't the owner and doesn;t have the correct permissions required by t
he command - fail with 'perms'

else if (cmdCfg.permReq.includes("BOT_OWNER") || !message.member.permissions.has(cmd
Cfg.permReq, true)) {
   return "perms"
  }

//if the array allowedChannels has 1+ entries, and it doesn't include the current ch
annel, fail with 'channel'

else if (!((cmdCfg.allowedChannels.length === 0) ? true : (!cmdCfg.allowedChannels.i
ncludes(message.channel.id)))) {
   return "channel"
  }
  //if the user is in the commands cooldown period, fail with 'cooldown'

else if (client.cooldown.get(message.guild.id, cmdName).filter(u => u === message.me
mber.id).length) {
   return "cooldown"
  } else {
   //if they didn't fail any checks, pass.
   return "passed"
  }
 }
```

Input	Output	Expected	Bug
User with no limiters	pass	pass	None
User with all limiters	Fail – command nonexistant	Fail – command nonexistant	None
User with nonexistant limiter	Fail – command nonexistant	Fail – command nonexistant	None

User with disabled limiter	Fail – command is disabled	Fail – command is disabled	None
User present in the blacklist	pass	Fail – blacklist	Blacklist check had invalid Boolean logic
User present in the backlist	Fail -blacklist	Fail -blacklist	None
User lacking permissions	Error: invalid perm “BOT_OWNER”	Fail – lack of perms	Did not add an external handler for the non-standard permission
User lacking Permissions	Fail – Lack of Perms	Fail – Lack of Perms	None
Channel not in allowed channels	pass	Fail – channel not allowed	Lacking check of array length before processing. should be if there are no allowed channels, assume all are allowed.
Channel not in allowed channels	Fail - channel not allowed	Fail - channel not allowed	None
User in command cooldown	Fail - cooldown	Fail - cooldown	None

Table 14 - Prototype 12 Testing Table

Comments:

Lots of minor tweaks and additions to deal with edge cases as well as for resiliency. Lots of testing and iterating using the specification to achieve the desired results. Points of note were the override for BOT_OWNER permissions, which are custom and not native to discord. It also developed alongside various data structures, which I finalised their implementation with the development of this function.

3.13 PROTOTYPE 13 – BASIC MODULAR COMMANDS

Goals: basic commands with command files

Pseudocode

Module.exports (client, message,…){

Command code

}

Module.exports.defaultConfig {

Aliases: [“alias1”,”alias2”],

Info: “this command does x”

Usage: “commandName arg1 arg2”

Enabled: true

permReq: [“MANAGE_MESSAGES”]

allowedChannels: []

}

Relevant code:

./commands/Help.js

```
exports.run = (client, message, args) => {
 message.channel.send(`help command requested with args ${args}`)
}


exports.defaultConfig = {
 aliases: ["help", "commands"],
 info: "lists the commands the user is able to execute in the current server",
 usage: "$help <command name/alias>",
 enabled: true,
 permReq: [],
 cooldown: 10000,
 allowedChannels: [],
};
```

./commands/Eval.js

```
exports.run = async (client, message, args) => {

const code = args.slice(1).join(" "); //removes the commandname from args, and then
executes the following statements/code.
 try {
  var evaled = eval(code); //evaluate (run) the code - VERY DANGEROUS!!

const clean = await client.evalClean(client, evaled); //small function to ensure tha
t the output doesn't contain valuable info, primarily the bot's token.

message.channel.send(`\`\`\`js\n${clean}\n\`\`\``); //send evaluated code as a code
block.
 } catch (err) {
  //send any errors to the channel.

message.channel.send(`\`\` \`\`\`js\n${await client.evalClean(client, err)}\n\`\`\``
);
 }
};
```

./functions.js

```
client.evalClean = async (client, text) => {

//checks if the evaled code is that of a promise, if so, awaits for the promise to r
esolve before continuing.
```

```
  if (text && text.constructor.name == "Promise")
   text = await text;
  if (typeof evaled !== "string")

//inspects the content so we don't just get [object Object] or the full object via s
tringify.

//this provides a single layer of informantion, eg test:{a:{"b"}} => test:{a:{object
}}.
   text = require("util").inspect(text, { depth: 0 });
  //replaces the token with "[BOT_TOKEN]"

text = text.replace(/@/g, "@").replace(/`/g, "`").replace(client.config.token, "[BOT
_TOKEN]");
  return text;
 }; //full disclosure: this code was copied off Etiket2 (another Discord
bot) as it is undoubtedly the best way to do this.
```

./commands/restart.js

```
exports.run = (client, message, args) => {
 message.react("✅") //reacts to the message as feedback.
 //invokes a restart with the provided reason(s)
 client.emit("gracefulShutdown", `Manual - ${args[1]}`)
};
```

Input	Output	Expected	Bug
Help command run	Help command has been run	Help command has been run	None
Eval command run	Code successfully evaluated	Code successfully evaluated	None
Restart command run	GracefulShutdown emitted	GracefulShutdown emitted	None

Table 15 - Prototype 13 Testing Table

Comments:

Basic features for each command – some smaller commands (e.g. eval, reboot) are what I would consider finished. Other commands (e.g. help) have a lot of functionality that needs to be added.

Functions such as ‘evalClean’, the guild validation code, and command loading/reloading have been moved to a separate file known as functions.js. this is where functions bound to the client will be organised instead of residing in Overlord.js

3.14 PROTOTYPE 14 – COMMAND RELOADING

Goals: command reloading

Reloading of a command from disk to apply latest changes/edits to a still running bot instance. This is primarily for debugging purposes.

Pseudocode:

From require cache delete ./commands/commandName.js

Client.loadCommand(commandName)

Relevant code:

./commands/Reload.js

```
exports.run = (client, message, args) => {
 //reloads the provided command
 client.reloadCommand(args[1], message.guild.id)
};
```

./functions.js

```
client.reloadCommand = (commandName) => {
  try {
   //deletes the cached version of the command, forcing the next execution to re-
load the file into memory.
   delete require.cache[require.resolve(`./commands/${commandName}.js`)];
   //load the command back into memory from disk.
   client.loadCommand(commandName);
  } catch (err) {
   //catch and log any errors
   client.log(`Error in reloading command ${commandName} - \n${err}`, "ERROR");
  }
 };
```

Change to client.loadCommand:

./functions.js

```
client.loadCommand = (command, guildid) => {
  if (!guildid) {
   //if no guildid is specified, loads the command for all guilds.
   //useful for initialisation of the bot, and for reloading (as these are bot-wide)
   client.guilds.forEach(guild => { client.loadCommand(command, guild.id); });
  }
  //try:catch for any errors.
  try {
   //loads contents of 'command'.js from disk
   var cmdObj = require(`${client.basedir}/commands/${command}.js`);
   //ensures each guild has the configuration data required for the command command.
```

```
    client.DB.ensure(guildid, cmdObj.defaultConfig, `commands.${command}`);
    //ensures command's presence in client.cooldown
    client.cooldown.ensure(guildid, [], command)
    //ensures that the aliases for the command are present.
    //for every alias...
    client.DB.get(guildid).commands[command].aliases.forEach(alias => {
     //ensure the alias maps to 'command' in commandsTable
     client.DB.ensure(guildid, command, `commandsTable.${alias}`)
     //log the sucessful binding of each alias.

client.log(`bound alias ${alias} to command ${command} in guild ${client.guilds.get(
guildid).name}`);
    });
   } catch (err) {
    //catch and log any errors
    client.log(`Failed to load command ${command}! : ${err}`, "ERROR")
   }
 };
```

Input	Output	Expected	Bug
Reload Command with no args	Caught error	Caught error	None
Reload Command with malformed arg	Caught error	Caught error	None
Reload Command with valid arg	Command reloaded	Command reloaded	None

Table 16 - Prototype 14 Testing Table

Comments:

Over the course of the development of this feature, it became apparent that I would need to alter the load Command section created in prototype 5 to its own separate function, as well as altering some of its behaviours such as the self-recursion for loading a command in all guilds (which is the primary use case) as well as logging and integration with other data structures (e.g. 'client.cooldown'). A sizeable amount of research was required before stumbling upon the 'delete require.cache…' snippet that did what I wanted in terms of functionality. It is a snippet that I plan to use as a way of reducing memory usage for certain single-require operations. Testing this function by altering a command file on disc and then reloading it. Changes were applied as command behaviour changed. This will be very useful for debugging, and the optimisations made to loadCommand will help bot-wide at boot-time. Another change of note is the usage of a mapping table (referenced above as commandsTable) to provide a way of resolving aliases with 'true' command names. I chose to do it this way instead of simply resolving through the aliases part of the command's configuration as by using the table it's 1). a very simple (and quick) lookup to resolve an alias and 2). allows for control over which commands are available by adding/removing them from the table without removing their configurations. This may sound somewhat pointless, but it proves invaluable if a 'commandFile' is removed between restarts, as it gives the bot a way to keep the configuration data whilst preventing the command from being available. Although this admittedly is an edge-case and more of a side effect of pursuing reason 1, it is still very useful.

3.15 PROTOTYPE 15 – SCHEDULER AND PERSISTENCE

Goals: Implementation of data persistency with scheduler

The scheduler is a key part of the bot – as many of the actions related to the automatic moderation system are planned to occur “some time” into the future. Although the timer implementation in JS is stellar, I decided to make my own version on top of it to slightly reduce load as well as allow me to customise some functionality, as well as add persistency.

This scheduler needs to be able to schedule actions to be ‘done at some point’, as well as storing these actions within the persistent database and catching back up if a restart occurs. It also needs to be able to react to new scheduler events occurring, as well as process actions of various types.

Psuedocode:

Define async function check(client, guildID)

Set timeouts to client.timeouts

If timeouts for this guildID does not exist, create it.

Else, set timeout to timeouts.guildID

Clear the current timeout (timeout)

Set now to the current date

Set data to all entries of client.DB in guildID.persistence.time

If there is no data, return

Set closest to the lowest timestamp value of the actions contained in data, that are scheduled to happen after now.

For every action with a timestamp that should have occurred already, execute the action

Set guildID.persistence.time in client.DB to data with all actions scheduled before now removed.

Set timeTo equal to closest – now

Log that scheduler will rerun in timeTo milliseconds

Set timeouts.guildID to a new timeout that will run check() with identical args.

Define async function actionProcessor (client,guildID,action)

Let guild equal to client.guilds.get(guildID)

Switch case with action.type

Case (if action.type == value then)

placeholderActionRelevantToType

break

Default: report invalid type as warn

Relevant code:

./functions.js

```
client.schedule = async (guildID, action) => {
 client.DB.push(guildID, action, "persistence.time")
 client.emit("scheduler", guildID)
}
```

./events/Scheduler.js

```
let actionProcessor = async (client, guildID, action) => {
 let guild = client.guilds.get(guildID)
 //switch:case discrimnator for each action type
 switch (action.type) {
  case "type1": //if action.type is equal to the case then..
   actionForType1
   break
  default:
   client.log(`Unknown action type/action ${JSON.stringify(action)}`, "WARN")
   break
 }
}

module.exports = async function check(client, guildID) {
 /** Scheduler - uses setTimeout to call itself
  * array of objects (
  * "action:"{end:TS,type:typeInst,...data}
  */
 let timeouts = client.timeouts //alias for timeouts.

if (!timeouts.has(guildID)) timeouts.set(guildID, null) //ensure the guild exists in
 timeouts.
 let timeout = timeouts.get(guildID) //get timeout for this guild

client.log(`Checking... (timeout = ${timeout}) ${new Date()} `); //notification that
 the scheduler is operating
 // clears previous check refresher
 clearTimeout(timeout);
 const now = new Date()
 //gets persistence data from the DB
 let data = client.DB.get(guildID, "persistence.time")
 if (data.length == 0) return
 //finds the shortest delta between now and the end of all the actions.

const closest = Math.min(...data.filter(action => action.end >= now).map(action => a
ction.end));
 //executes all the actions that should've been executed
- e.g. if the bot crashes. this allows it to catch back up, so to speak.
 data.filter(action => action.end <= now).forEach(action => {
  actionProcessor(client, guildID, action)
 })
```

```
client.DB.set(guildID, data.filter(action => action.end >= now), "persistence.time")

if (closest === Infinity) return; //if the number is infinity then there are no pend
ing actions.
 const timeTo = closest - now;
 client.log(`checking timeout in ${timeTo} ms`)
 // will only wait a max of 2**31 - 1 because setTimeout breaks after that

timeouts.set(guildID, setTimeout(check, Math.min(timeTo, 2 ** 31 - 1), client, guild
ID))
};
```

Input	Output	Expected	Bug
Valid action object with timestamp x ms in the future	Scheduler rescheduled for check in x milliseconds	Scheduler rescheduled for check in x milliseconds	None
Valid action with timestamp in the past	Scheduler immediately schedules action – action executed successfully	Scheduler immediately schedules action – action executed successfully	None
Valid action with no timestamp	Scheduler set to infinity – errors as greater than 2 ** 31 – 1	Scheduler is not set	Cannot handle values greater than 2 ** 31 – 1. Add checks for this and infinity.
Valid action with no timestamp	Scheduler is not set	Scheduler is not set	None
Invalid action with timestamp x ms in the future	Scheduler rescheduled for check in x milliseconds	Scheduler rescheduled for check in x milliseconds	None
Invalid action with timestamp in the past	Scheduler executed action – invalid action is caught and logged	Scheduler executed action – invalid action is caught and logged	None

Table 17 - Prototype 15 Testing Table

Comments:

During testing of this function, I found a few edge cases whilst testing with extreme data – 1, that if there are no closest values (e.g. all actions were supposed to be executed in the past) closest gets set to infinity – which causes setTimeout to break. After reading the documentation for setTimeout, it has a maximum timeout value of 2 ** 31 – 1. I included this value in a math.min function in the final call to ensure that this value is not exceeded, as well as halting execution if closest is equal to infinity after actions have been processed.

Currently, no systems use the scheduler but they will eventually. These systems will extend the actionProcessor's case statements.

3.16 PROTOTYPE 16 – FINALISE MESSAGE EVENT

Goals: finish ./events/message.js

Need to finish up message to be able to develop and test new features – by far one of the larger planned prototype stages. It will be implementing a lot of features that are not currently present.

The specification for this prototype is as follows:

Messages sent by a bot must be ignored

Messages sent whilst the client flag isShuttingDown should be ignored, and feedback given to the user.

Messages not sent in a guild by any user other than the bot owner should be ignored

If the author is the bot owner, process any input as commands. Else;

Assign the guild configuration to the message object for ease of referencing

If the member is not loaded into cache, load the member into cache.

Using regEx, determine if the bot (and just the bot) has been mentioned. If so, send the user a DM giving them the prefix in the current guild.

Send the message off for processing by other heuristics (not yet implemented)

Check if the message is a command. If not, do nothing. Else;

Split the message's content into an array of arguments.

Resolve the command name from an alias using the commandsTable.

Load the configuration of the command.

Check the state of the command via client.canExecute (implemented in a previous prototype stage)

Create a new embed to be populated and sent to the user as a form of feedback

Configure a reaction collector, to wait for the author reacting to the message with a ❔ emoji. If a failure state occurs, the bot reacts with the ❔ emoji, which allows the user to also react with it to prompt feedback. More information on this can be found in the UX section.

If the user passes the checks, they are added to the command's cooldown

The command as well as the arguments and invoker are logged.

The command object is loaded into memory and then executed. Any errors result in the same procedure as a failure state, just with different contents for the embed. The error is also logged.

Relevant code:

./events/message.js

```
/**
 * primary event - triggered whenever any channel in any guild the bot can 'see' get
s sent a new message.
 * almost all the processing in the bot is based off data from this event - it is th
e main event, so to speak.
 * @param {object} client
 * @param {object} message - Object of the message that has been sent by a user.
 */
module.exports = async (client, message) => {
 //partials are uncached data that can then be resolved into 'full' data structures.
```

```
//not currently used due to lower project version (v11 vs V12). once I update this w
ill become very useful.
 if (message.partial) {
  await message.fetch().catch(err => {
   client.log(err, "ERROR")
   return
  })
 }
 //ignores all messages from other bots or from non-
text channels, EG custom 'news' channels in some servers, or storefront pages, etc.
 if (message.author.bot || !message.channel.type == "text") return;

//checks if the bot is currently undergoing a shutdown. if so, interdicts all furthe
r processing.
 if (client.isShuttingDown) {

//reactions as a form of feedback to the user. these indicate 'stop' and 'wait' for
the shutdown.
  message.react("🚫").then(() => { message.react("⌛"); });
  return;
 }
 //guild-only due to increased technical complexity to account for non-
guild scope'd interactions.
 if (message.guild) {

//binds the guild's settings and the level of the user to the message object, for ea
se-of-access for later operations (eg commands)
  message.settings = client.DB.get(message.guild.id)

//fetches the member into cache if they're offline. important to do this as this can
 be used by functions later on.
  if (!message.member) await message.guild.members.fetch(message.author);

//used to check that the message contains *only* the mention of the bot, and nothing
 else.
  let  botMentionRegEx = new RegExp(`^<?@!?${client.user.id}>?$`);
  //checks if the bot, and *only* the bot, is mentioned.
  if (message.isMentioned(client.user.id) && message.uncleanContent.match(
botMentionRegEx)) {

//sends (DM's) the user the Command Prefix for the guild, or the default prefix if a
nything out of scope happens.

message.author.send(`Hi there,${message.member.displayName}, My prefix in guild ${me
```

```
ssage.guild.name} is ${message.settings.prefix || "$"}.\n for help, use the command:
 ${message.settings.prefix || "$"}help `);
   return
  }

//built in method for cleaning message input (eg converting user mentions into a str
ing to prevent issues when returning message content)
  message.content = message.cleanContent;
  //emit events for message heuristics
  client.emit("attachmentRecorder", message)
  client.emit("toxicClassifier", message)
  client.emit("autoMod", message)
  //command processing
  //checks the message starts with the prefix.
  if (message.content.startsWith(message.settings.prefix)) {
   //splits the messages content into an array (space delimited)

let args = (message.content.slice(message.settings.prefix.length).trim().split(/ +/g
))
   //resolve 'true' command name from an alias.

let cmdName = client.DB.get(message.guild.id, `commandsTable.${args[0].toLowerCase()
}`)
   //resolves the commands config using the 'true' name
   let cmdCfg = message.settings.commandsTable[cmdName]

//checks that the user can execute the command (see Functions.js). stores resultant
state as state
   let state = client.canExecute(client, message, cmdName)
   //imports discord.js to construct embeds
   let Discord = require("Discord.js")
   //creates a new embed instance
   let embed = new Discord.RichEmbed()
    .setAuthor(client.user.username, client.user.avatarURL)
    .setTimestamp(new Date())

//default content that should always be replaced. if not, it's very clear something
is not functioning correctly.

.setTitle("if you are seeing this, something has gone wrong. please inform the bot o
wner.")

.setDescription(`If you think this is a mistake, please contact an Administrator.`)
    .setColor("#FF0000") //red
```

```
message.awaitReactions((reaction, user) => { return reaction.emoji.name === "❓" &&
user.id === message.author.id }, { max: 1, time: 6000 })

//restrictions are indicated by reactions - the ❓ reaction can be used to make the
bot send the user the 'key' as to what each reaction means.
    //this code waits for a reaction, checks the reaction 'name' and react-
er before sending the embed to the user.
    //this is the best way I found to provide the user with non-
intrusive feedback for commands, etc.
    .then(collected => {

//can proceed even if no reactions occur in the time frame. added a check to ensure
one is added.
     if (collected.size) {
      message.author.send({ embed: embed })
     }
     //remove the reactions from the message - reduces visual clutter
     message.clearReactions()
    })

//switch:case for modifying the reporting embed's contents according to what the fai
lure is.
   switch (state) {
    //if the command doesn't exist
    case "nonexistant":
     embed.setTitle(`Command/Command Alias ${args[0]} Does not exist.`)

.setDescription(`Please use the \`\`\`\n${message.settings.prefix}Help\n\`\`\` Comma
nd to see all Available Commands.`)
     break
    //if the command is disabled (guild-wide)
    case "disabled":

embed.setTitle(`Command ${args[0]} has been disabled for guild ${message.guild}.`)
     break
    //if the user lacks the perms required
    case "perms":

embed.setTitle(`You do not have the required permissions to execute command '${args[
0]}' in Server ${message.guild}`)
     break
    //if the channel is not in the allowed array
    case "channel":
```

```
embed.setTitle(`Command '${args[0]}' has been disabled in channel ${message.channel}
 in Server ${message.guild}`)
     break
    //if the user is still in the command's cooldown period
    case "cooldown":

embed.setTitle(`Woah There! Please slow down with the usage of Command '${args[0]}'
in Server ${message.guild}`)
     break
    //if the user is in the blacklist for this command - guild-
wide ban of specific command usage
    case "blacklist":

embed.setTitle(`Oh Dear. Looks like you've been blacklisted from using command '${ar
gs[0]}' in Server ${message.guild}`)
     break
    //if the user passes all the checks...
    case "passed":
     //if the user is not an admin, add them to the command's cooldown.
     if (!message.member.permissions.has("ADMINISTRATOR")) {
      client.cooldown.push(message.guild.id, message.member.id, cmdName, true)

//after a set time, remove them from the cooldown, allowing them to use the command
again.

setTimeout(() => { client.cooldown.remove(message.guild.id, message.member.id, cmdNa
me); }, cmdCfg.cooldown)
     }
     //log the fact that a command has been executed.

client.log(`User ${message.author} executed command ${cmdName} with arguements ${arg
s.join(", ")} in ${message.guild}:${message.channel}`, "INFO")
     //load the command from disk as an object
     let cmdObj = require(`${client.basedir}\\commands\\${cmdName}.js`)
     try {
      //run the command - passing args and the message invoking the command.
      cmdObj.run(client, message, args)
     } catch (err) {
      //on an error, log it, then provide 'feedback' to the user

embed.setTitle(`Oh Dear. It seems that an error occurred whilst trying to execute th
is command.`)

.setDescription(`If this continues to occur, please notify ${client.users.get(client
.config.ownerID)}`)
```

```
//react to the message to allow the user to aquire feedback - optional but clear eno
ugh.
      message.react("❌").then(() => { message.react("❓") })
      client.log(`Error with command ${cmdName}`, "ERROR")
      console.log(err)
     }
     break
   }
   //this is the trigger that allows for the embed to be sent to the user.
   if (state != "passed") {
    message.react("🚫").then(() => { message.react("❓") })
    return
   }
  }
 } else {

//Owner universal restriction escape - as they control the bot anyway - useful for d
ebugging.
  //treat everything they send as a command.
  if (message.author.id === client.config.ownerID) {
   let args = (message.content.trim().split(/ +/g)) //split args into an array
   let cmdObj = require(`${client.basedir}\\commands\\${args[0]}.js`)
   try {
    cmdObj.run(client, message, args) //run the command
   } catch (err) {
    console.log(err)
   }
   return
  } else { //log any messages sent to the bot Via DM's.

client.log(`Message with contents ${message.content} sent to the  bot Via DM's by us
er ${message.author}`, "INFO")
  }
 }
}
```

Input	Output	Expected	Bug
Message from non-owner via Direct Messaging	No processing	No processing	None
Message from owner via Direct messaging	Message Processed via Owner override	Message Processed via Owner override	None

Message from a bot user	No processing	No processing	None
Message from user in a guild	Processed just by heuristics – error due to unknown member	Processed just by heuristics	Add code to ensure member is in bot member cache
Message from a user in a guild with just a command prefix	Processed by heuristics and then command processing system – lack of command error caught. Reporting embed sent unprompted	Processed by heuristics and then command processing system – lack of command error caught Reporting embed sent when prompted	reactionCollector finishes either when the conditions have been met or if the timelimit is surpassed. Added check for data to prevent this.
Message from a user in a guild with command prefix	Processed by heuristics and then command processing system – lack of command error caught Reporting embed sent when prompted	Processed by heuristics and then command processing system – lack of command error caught Reporting embed sent when prompted	None
Message sent with command prefix and valid command with no limiters	Heuristics processing and then command processing – no errors, data passed to executed commandFile	Heuristics processing and then command processing – no errors, data passed to executed commandFile	None
Message sent with limiter(s)	Heuristics processing – execution stops at earliest limiter with correct error message in reporting embed.	Heuristics processing – execution stops at earliest limiter with correct error message in reporting embed.	None Note: no need for exhaustive testing as this was done for client.canExecute in a previous prototype.
Message sent with malformed arguments/command data	Heuristics and Command processing completed, commandFile execution errored. Error caught and report produced.	Heuristics and Command processing completed, commandFile execution errored. Error caught and report produced.	None
Multiple Messages sent with the same command under 1000ms apart	Processing by Heuristics, command processed by stopped at cooldown check	Processing by Heuristics, command processed	Administrators are not exempt from cooldown.
Message sent in specialised channel type	Fatal error in code due to lack of required metadata	No processing	Content from non-text channels was not being ignored
Message sent in specialised channel type	No processing	No processing	None
Message with command content as well as mentions	Heuristics and command processing - error due to unknown command	Heuristics and command processing	Command alias resolver was caps sensitive

Message with malformed command content as well as mentions	Heuristics and command Processing – report embed kept mention	Heuristics and command Processing - error for malformed caught	Input not sanitised for Discord (mentions and other special strings not sanitised/removed)
Message with malformed command content as well as mentions	Heuristics and command Processing - error for malformed caught	Heuristics and command Processing - error for malformed caught	None
Message with bot mentioned	bot responded with prefix help message	No response due to non-exclusive mention	System does not screen for just the bot alone being mentioned. Implement exclusive regEx.
Message with bot mentioned	bot responded with prefix help message	No response due to message containing other content	Non-exclusive regEx for content – modified regEx to ensure that it only responds if the message is just a bot mention and nothing else.
Message with bot mentioned	No response due to message containing other content	No response due to message containing other content	None
Message with just bot mention	bot responded with prefix help message	bot responded with prefix help message	None

Table 18 - Prototype 16 Testing Table

Comments:

This version of the message event handler is, in my opinion, complete for all intents and purposes, implementing all the features and behaviours required, as well as some additional features that were added during testing. Testing involved messages of varying contents, as well as doctored environments to trigger specific failure states – every failure state was induced and tested, with the reporting system also being tested:

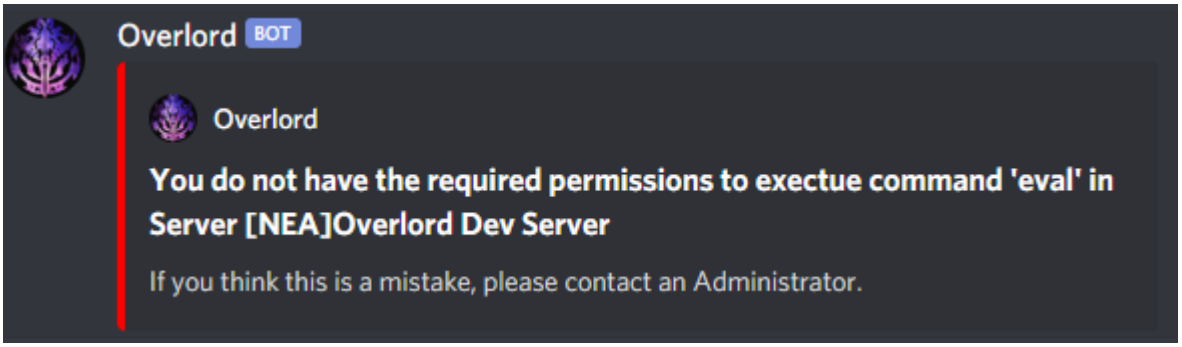

Figure 19 - Lack Of Permissions Error Message

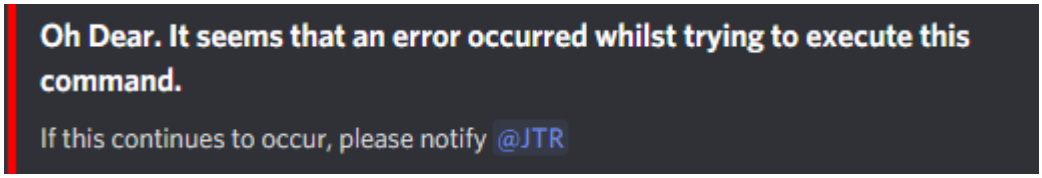

Figure 20 - General Error Message

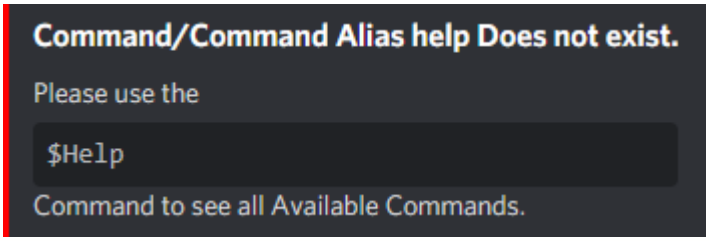

Figure 21 - Unknown Command/Alias Error Message

A fair few bugs were uncovered over the course of testing – most notably an issue with the reactionCollector wherein it would trigger even if the author hadn't reacted once the time limit expired. Adding a check for this resolved this issue. Another issue was that channels that weren't just text channels could post messages. An example of this is a store channel/page and a news channel/page. This caused issues as these messages do not have the typical metadata that the bot uses. Another bug was that when the bot re-used input text as part of the error messages, these could contain mentions as the content was not sanitised. Luckily, a method of a message object means that I can replace the contents with a sanitised version, after running the code that requires the unaltered version (the 'mention detection' code). Another bug was in this aforementioned 'mention detection' code, with the 'regEx' initially triggering if there was other message content apart from the mention. I fixed this by specifying that no other characters are to be before or after the mention to be matched in the regex sequence. Another bug can be seen in the last snippet of the reporting embeds; the alias/command resolution system was initially caps-sensitive. This has quickly been resolved by changing the command name to its lowercase version strictly. Yet another change was the implementation of a cache fetch for the member if they were not in cache already, as 'message.member' is used in quite a few commands later on, mainly in heuristics, as well as the addition of a 'partials' handler – not currently used but planned to be when I update Overlord to the latest API version (which was released amid development).

3.17 PROTOTYPE 17 – MODERATION HEURISTICS

Heuristics engines

Goals: implement Heuristics engines according to the specifications laid out earlier in this document.

Heuristics algorithms include char%, caps% and spam detection. The previously created client.trecent will be used for this.

Pseudocode:

Define async function (client, message)

Set config to message.settings

Set modConfig to config.modules.autoMod

Set ASconfig to modConfig.antiSpam

Set member to message.member

Set trecent to client.trecent

If member has a role that is 'protected' (ID in the module config as an excluded role) return

Else;

If the author is the bot owner, also return

Else;

Execute function antispam

If the module config (modConfig) heuristics.ignoreBelowLength is greater or equal to the length of the message's content, return.

Set chars to a new map instance

For every character in the message's contents,

Set characters[character] to the number of instances of this character in message.content

Set chars to an array from chars.entries().

Execute function caps(client, message)

Execute function charFlood(client, message)

Define function antispam (client, message)

If the module is disabled return

Ensure that an entry for this channel exists within client.trecent.guildID

Push the authorID to the array at client.trecent.guildID.channelID

Create a timeout operation that will remove a single instance of the authorID from the array after a number of milliseconds

Set userCooldown to this array

Filter this array to all instances of the authorID

If the number of instances exceeds the threshold, report as spamming.

Define function caps(client, message)

If the function is not enabled, return

Else;

If the %age of a character in chars which is uppercase exceeds the percentage limit

Report as caps flooding

Define function charFlood (client, message)

If the function is not enabled, return

Else;

If the percentage of a character in the message exceeds a threshold

Report as character flooding

Relevant code:

./events/autoMod.js

```
/**
 * runs optional heuristics on messages, such as antispam, charFlood, and CapsFlood.
 * @param {object} client
 * @param {object} message
 */
module.exports = async (client, message) => {
 //declare some aliases for commonly use pieces of data.
 let config = message.settings
```

```
 let modConfig = config.modules.autoMod
 let ASconfig = modConfig.antiSpam
 let member = message.member
 let trecent = client.trecent
 //if the user has one of the 'excluded' roles, ignore them.

if (Array.from(member.roles).filter(role => modConfig.excludedRoles.includes(role)).
size >= 1) { return }
 //if the user is the bot's owner, ignore them.
 if (message.author.id === config.ownerID) { return }
 //prepare action object with common data.
 let action = {
  memberID: message.author.id,
  guildID: message.guild.id,
  type: "action",
  autoRemove: false,
  //autoRemove: ASconfig.autoRemove,
  penalty: ASconfig.penalty
 }
 /**
  * runs antispam heurisitics on a given message object.
  * @param {object} client
  * @param {object} message
  */
 let antiSpam = (client, message) => {
  //check state - if not enabled, do nothing.
  if (!ASconfig.enabled) return

//ensure entry in DB exists for the specified channel. if it exists, return the curr
ent val. if not, set to provided val ([])
  let userCooldown = trecent.ensure(message.guild.id, [], message.channel.id)

//push the member ID to the relevant subObject array to indicate who has been sendin
g messages where.
  //true is used to allow for duplicate pieces of data in the array.
  trecent.push(message.guild.id, member.id, message.channel.id, true)

//after ASconfig.interval Milliseconds, remove a instance of member.id from the trec
ent array for this channel.

setTimeout(() => { trecent.remove(message.guild.id, member.id, message.channel.id) }
, ASconfig.interval)
```

```
//if the number of instances of member.id for this channel exceeds the threshold, th
e user is classified as 'spamming.'

if ((userCooldown.filter((user) => user === member.id)).length >= ASconfig.count) {
   //remove all other instances of member.id from trecent

trecent.set(message.guild.id, userCooldown.filter((user) => user != member.id), mess
age.channel.id)
   //populate action object
   Action = {
    title: "Message spam Violation",

src: `Spam limit broken by user ${message.author} in channel ${message.channel} : [J
ump to message](${message.url})`,
    trigger: {
     type: "Automatic",
     data: `Spam Limit Broken by ${message.author}`
    },
    request: `Removal of content (last ${ASconfig.count} messages)`,
    requestedAction: {
     type: "bulkDelete",
     target: `${message.guild.id}.${message.channel.id}.${message.author.id}`,
     count: ASconfig.count
    },
   }
   client.emit("modActions", { ...action, ...Action })
  } else {

//add XP, using the antispam delay as a restriction/timeout. will be it's own module
 eventually.

client.DB.math(message.guild.id, "+", client.getRandomInt(1, 3), `users.${message.au
thor.id}.xp`)
  }
 }
 /**
  *
checks the %age of a message's contents being the same character vs a configured thr
eshold
  * @param {object} client
  * @param {object} message
  */
 let charFlood = (client, message) => {
```

```
  //check if enabled - if not, do nothing
  if (!modConfig.heuristics.charFlood.enabled) return

//if the perctange of the message being one character is above the threshold, take a
ction

if (chars.filter(char => (char[1] / message.content.length) * 100 >= modConfig.heuri
stics.charFlood.percentLimit).length != 0) {
   //flesh out the action object with properties
   Action = {
    title: "Message Flood Violation",

src: `Flood % limit broken by user ${message.author} in channel ${message.channel} :
 [Jump to message](${message.url})`,
    trigger: {
     type: "Automatic",
     data: `Flood Limit Broken by ${message.author}`
    },
    request: `Removal of content`,
    requestedAction: {
     type: "delete",
     target: `${message.guild.id}.${message.channel.id}.${message.id}`,
    },
   }

client.emit("modActions", { ...action, ...Action }) //pass action to modActions for
processing.
  }
 }
 /**
  * checks the message contents for characters that are in upper case (caps) and det
ermines the message %age that is comprised of capitals
  * @param {object} client
  * @param {object} message
  */
 let caps = (client, message) => {
  //check if enabled - if not, do nothing.
  if (!modConfig.heuristics.caps.enabled) return

//if the total percentage of the message as caps is greater than the threshold, take
 action.

if (((chars.filter(char => char[0] === char[0].toUpperCase()).length) / message.cont
ent.length) * 100 >= modConfig.heuristics.caps.percentLimit) {
```

```
   //if caps% exceeds or equals configured limit, take action.
   Action = {
    title: "Message Caps Flood Violation",

src: `Caps % limit broken by user ${message.author} in channel ${message.channel} :
[Jump to message](${message.url})`,
    trigger: {
     type: "Automatic",
     data: `Caps % limit Broken by ${message.author}`
    },
    request: `Removal of content`,
    requestedAction: {
     type: "delete",
     target: `${message.guild.id}.${message.channel.id}.${message.id}`,
    },
   }

client.emit("modActions", { ...action, ...Action }) //pass action to modActions for
processing.
  }
 }


 antiSpam(client, message)

//if below the ignore threshold, ignore (as smaller messages are much more likely to
 be a false +ve.)
 if (modConfig.heuristics.ignoreBelowLength >= message.content.length) return
 //below is used to check how many instances of a character exist in a string.
 let chars = new Map();

//iterates over every character as a spread string array (spreads the string into in
dividual characters, then makes each char it's own entry in the array)
 [...message.content].forEach(char => {
  //escape control chars for the RegEx to prevent them from messing anythng up
  if (["."", "{", "}", "[", "]", "-
", "/", "\\", "(", ")", "*", "+", "?", "^", "$", "|"].includes(char)) char = "\\" +
char
  //sets the value of 'char' to however many chars there are, or simply none.
  chars.set(char, (((message.content).match(new RegExp(char, "g")) || []).length))
 })
 //convert to an array of entries : [...,[char, count],...].
 chars = Array.from(chars.entries())
 //execute heuristics (caps)
 caps(client, message)
```

```
 //execute heuristics (chars)
 charFlood(client, message)
}

//default configuration - allows for editing of thresholds and punishments
module.exports.defaultConfig = {
 enabled: true,
 bannedWords: [], //stump
 bannedURLs: [],
 excludedRoles: [],
 punishments: {
  mute: {
   start: 5,
   end: 0
  },
  tempBan: {
   start: 10,
   end: 5
  },
  ban: {
   start: 15,
   end: -1 //-1 = infinite duration
  }
 },
 decay: 3,
 heuristics: {
  ignoreBelowLength: 20,
  charFlood: {
   enabled: true,
   percentLimit: 40,
  },
  caps: {
   enabled: true,
   percentLimit: 40,
  }
 },
 antiSpam: {
  enabled: true,
  interval: 3000,
  count: 5,
  penalty: 1,
```

```
  autoRemove: true
 },
 requiredPermissions: ["MANAGE_MESSAGES"]
}
```

Comments:

Much like the message command, I planned to almost fully complete this section – the only bit left to do is fully integrate it into the yet-to-be-developed moderation backend. Other than this, this module does everything I required for a heuristics system. I also added a temporary XP statement to the antispam section, as this is a feature I am planning to implement in the future and wanted to test out here as it seemed suitable.

Testing:

Input	Output	Expected	Bug
<40% of a message as caps	Reported for caps spam	No action	regEx control chars
<40% of a message as caps	No action	No action	None
>40% of a message caps	Reported for caps spam	Reported for caps spam	None
<5 messages in 3 seconds	No action	No action	None
5 messages in 3 seconds	Reported for spam	Reported for spam	None
5 messages in 3 seconds, followed by another 2	3 spam reports	1 spam report	Trecent entries not wiped after report
5 messages in 3 seconds, followed by another 2	1 spam report	1 spam report	None
<40% of a message a single char	No action	No action	None
>40% of a message a single character	Reported for char flood	Reported for char flood	None
Content below cut-off length	No processing	No processing	None
Content above cut-off length	Processing done	Processing done	None

Table 19 - Prototype 17 Testing Table

Changes and bugs:

Bug: after a user was reported for spam, sending another message would re-report them. This is not intended behaviour. Resolution: wipe all instances of the user for spam once they have been reported for spam.

Bug: specialised characters (regEx control characters) lead to issues within the char counting subroutine regEx. Resolution: implementation of a character escape for these control characters.

3.18 PROTOTYPE 18 – ATTACHMENT RECORDING

Goals: attachment Recording for message deletion as well as NSFW classification

As I recently found out, attachments are deleted when the message they are attached to is. Thus, I need to create a system to download and then optionally upload content to an external (secure) service. The content needs to first be downloaded not only for the upload process (as some attachments require being authenticated to Discord to download so I can't just use a redirect) but also for classification by the planned NSFWClassifier. The external service I have chosen is transfer.sh for its large file limits, existing NPM module, automatic deletion after 14 days, with unlimited downloads during this time and the option for password locking.

Specification:

Check for attachments and embeds for valid media types (pictures/videos) within a message

(embeds are created for detected media URLs automatically by discord)

For every attachment/piece of media;

Download the media to a local cache

Process it via NSFWclassifier or uploading depending on config

If NSFWclassifier is disabled, delete the copy from cache

Else if NSFWclassifier is enabled, do not delete the copy from cache (and NSFWclassifier will handle deletions if it is enabled)

Once all attachments have been processed,

Write a new entry in client.DB.guildID.persistence.attachments under key messageID with the array of transfer.sh URLs as well as a timestamp 14 days in the future for cleaning in case of expiry.

Relevant code:

./events/attachmentRecorder.js

```
/**
 * detects and downloads any attachments present in a message - detects links via th
e automated embed system Discord uses.
 * uploads and pipes data to NSFWClassifier (optional). attachments used for when a
message is deleted.
 * @param {object} client
 * @param {object} Message
 */
module.exports = async (client, Message) => {
 var modConfig = Message.settings.modules.attachmentRecorder //gets module config
 //checks state
 if (!modConfig.enabled) return
 //''accumulator'' for attachment links
 var atts = []
 //sets the cache path for downloaded files.
 var path = modConfig.storageDir
 /**
  * downloads all attachments from a given message object - optional piping and stor
ing.
  * @param {object} client
```

```
  * @param {object} message
  */
 let getAttachments = (client, message) => {
  //array of URL's
  let toProcess = []
  //add any attachments URLs
  toProcess.push(...message.attachments.array().map(attachment => attachment.url))
  //add any URLs from media Embeds.

toProcess.push(...(message.embeds.filter(embed => embed.type === "image" || embed.ty
pe === "video")).map(embeds => embeds.url))
  //iterate over each URL
  toProcess.forEach(att => {
   //sets the file name = message.id + a counter

var filename = message.id + "(" + (toProcess.indexOf(att)) + ")" + "." + att.split("
/").pop().split(".")[1];
   //full file path, eg D:/Overlord/cache/445743395557322(0).jpg
   var filePath = path + filename
   //invoke download with almost unlimited timeout.

client.download(att, { directory: path, filename: filename, timeout: 99999999 }, fun
ction (err) {
    if (err) client.log(`download of attachment ${att} failed!`, "ERROR");
    else {
     client.log("download successful!");
     //check if attachments are kept and if NSFWClassifier is enabled.
     if (message.settings.modules.NSFWClassifier.enabled && !modConfig.keep) {
      //pipe data to NSFWClassifier
      client.emit("NSFWClassifier", message, filename);
     }
     //if attachments are configued to be kept..
     if (modConfig.keep) {
      client.log("starting upload...")
      //upload the file to transfer.sh.
      new client.transfer(filePath)
       .upload().then(function (link) {

//only unlink without NSFWClassifier - as NSFWC deletes the files it classifies anyw
ay.
        if (!message.settings.modules.NSFWClassifier.enabled) {
         client.fs.unlink(filePath, (err) => {
```

```
if (err) { client.log(err, "ERROR"); } else { client.log("Unlink successful!"); }
         })
        }
        //add the link to atts
        atts.push(link)
        client.log(`Upload Successful! ${link} `)

//check that the attachment is the last one to be processed before writing entry
        if (modConfig.keep && atts.length === toProcess.length) {

//sets expiry of the attachments to 14 days, which is the expiry of data on transfer
.sh.

client.DB.set(message.guild.id, { attachments: atts, expiry: (new Date()).setDate((n
ew Date()).getDate() + 14) }, `persistence.attachments.${message.id}`)
        }
        if (message.settings.modules.NSFWClassifier.enabled) {
         //run NSFW classifier if attachments are kept.
         client.emit("NSFWClassifier", message, filename);
        }
       })
     }
    }
   });
  });

 }
 //delay to wait for embeds to be automatically generated.
 setTimeout(getAttachments, 500, client, Message)

};
//default configuration for the module.
module.exports.defaultConfig = {
 enabled: true,
 storageDir: "./cache/",
 keep: true,
 requiredPermissions: ["MANAGE_MESSAGES", "READ_MESSAGE_HISTORY", "VIEW_CHANNEL"]
};
```

Input	Output	Expected	Bug

Message with no attachments	No Processing	No Processing	None
Message with a single attachment, no keep no NSFWc	Attachment downloaded – no upload to transfer.sh or piping to NSFWc	Attachment downloaded – no upload to transfer.sh or piping to NSFWc	None
Message with a single image url sent	No processing due to lack of URLs due to no embeds	Download of media from embed URL, setting of persistence object	Delay between message posting and generation of embeds means that at execution time they do not yet exist. Solution is to add a small delay to account for this.
Message with a single image url sent	Download of media from embed URL, setting of persistence object. Passed to keep and NSFWc	Download of media from embed URL, setting of persistence object. Passed to keep and NSFWc	None
Message with attachment and media link sent (multiple pieces of media)	both downloaded, but one was lost due to the same filename	both downloaded and passed to keep/NSFWc	Lack of discriminator apart from message ID for multiple child media elements. Added ‘counter’ to resolve this.
Message with attachment and media link sent (multiple pieces of media)	both downloaded and passed to keep/NSFWc according to config	both downloaded and passed to keep/NSFWc according to config	None
Message with media and just keep enabled	Media downloaded and then uploaded then deleted, persistence entry created	Media downloaded and then uploaded then deleted, persistence entry created	None
Message with media and just NSFWc enabled	Media downloaded and then saved. No upload or persistency, data sent to NSFWc	Media downloaded and then saved. No upload or persistency, data sent to NSFWc	None

Table 20 - Prototype 18 Testing Table

Comments:

Having a dedicated module to handle the downloading and ‘keeping’ of attachments was not in my initial requirements set, but had to be added due to the aforementioned issue with Discord deleting media when the message is deleted. Although a downloader would’ve been required for NSFWclassifier to work, this would have been contained just within that module. The set of bugs I encountered whilst developing this module were unexpected, especially the embed one – as it also explains the seemingly random ‘messageUpdate’ events wherein no message content changes. It seems that when the embed is added, it triggers a ‘messageUpdate’ event. I will need to keep this in mind when I develop this feature.

Transfer.sh is the optimal solution out of the set I evaluated as free file sharing hosts, and although I didn’t choose to use the password protected option for this version, it is a feature I am eventually planning to implement.

As far as the potential concerns this raises over data security, transfer.sh uses randomised links to ensure that unless someone were to perform a focused brute force attack (some variant of war dialling), the links are known only to Overlord. This coupled by the fact they are only ever exposed outside Overlord in the guild that the content was posted, as well as only to administrators, as well as automatically expiring after 14 days, the security concerns over using an external service are minimal.

3.19 PROTOTYPE 19 – NSFW UGC CLASSIFICATION

NSFW Neural Network implementation

Due to various legal and moral reasons, I outsourced this testing process to willing volunteers of my Admin Client Group.

To do so, I had to develop a standalone application that implements the classifier as well as reporting functions. This is NSFWtest.js

The NSFWClassifier is based on NSFW.js, a module that allows for me to load a Neural Network Layers model, and process images through it for a classification certainty from 0 to 1 for how confident the NN thinks an image is part of this classification class. The classes are Hentai (NSFW), Porn (NSFW), sexy (NSFW) and Drawing (SFW), as well as a neutral (SFW) class. The three classes we are interested in for this Project are just the NSFW classes.

The requirements for this prototype are:

Load the Neural network at initialisation according to the global configuration flag (config.js/enableModels) to client.NSFWmodel

Within the module file:

Check the status – global and guild scope enabled status

To take an input of a filename and message object

Read the file specified by the filename

Process and reformat it into a format that can be used for tensor processing (e.g. conversion to a tensor)

Classify the processed image/tensor using the loaded Neural Network

Take the classification data from the Neural Network and process it using configurable weightings to determine if a piece of content is suitably NSFW or not.

If it is classed as NSFW by these weightings, report it

If it is not, don’t report it

Delete the file from the ./cache/ folder

In addition to this, once the module has been created, I will need to package it into a standalone bot to be used by my Admin Client group volunteers for testing and feedback.

Pseudocode:

Define async function (client,message,filename)

If the NSFWmodel is not bound to client, return

Set cacheDir to message.settings.modules.attachmentRecorder.storageDir

Set modConfig to message.setting.modules.NSFWClassifier

If this module is disabled, return.

Define async function classifier (client,img)

Return new promise resolve

Set imageBuffer to fs.readFileSync cacheDir + img

Set image to tf.node.decodeImage imageBuffer, 3

Execute client.NSFWModel.classify image then setting output as predictions;

Resolve promise with predictions

Delete the file from cacheDir

Return

Execute classifier(client,filename) then setting output as predictions;

Set predictions to predictions filtered by filter className != “Neutral” or “Drawing”

If the number of predictions above their specified weight exceeds the threshold count,

Create array classi

Push the classname and certainty of every prediction to array classi

Report the content as NSFW.

Relevant code:

Overlord.js

```
/**
 * Loads the neural net model for the NSFWClassifier.
 * @param {object} client
 */
let modelLoad = async (client) => {
 //checks that the models aren't disabled in the main bot config.
 if (!client.config.enableModels) return;
 client.tf = require("@tensorflow/tfjs-node");

//flag to tell TF that this is a production app. nets some major performance improve
ments.
 client.tf.enableProdMode()
 //loads the NSFWModel via NSFWjs Using local model files.

client.NSFWModel = await require("nsfwjs").load("file://./models/NSFW/", { size: 299
 });
 //creates a new instance of the ToxicityClassifier
 client.log("Models loaded!");
 }
 //triggers the loading of the NeuralNet Models. for some reason a self-
invoking anonymous function refused to work.
 modelLoad(client)


 ./events/NSFWClassifier.js
 /**
 * given a filename, runs it through the NSFWModel to determine if it
 * is above the allowed thresholds for each classification of content.
 * e.g. above 90% certainty of being Pornographic, before triggering
```

```
 * moderation intervention.
 * @param {object} client
 * @param {object} message
 * @param {string} filename - name of the file within the cache directory.
 */
 module.exports = async (client, message, filename) => {
  //checks that the models are enabled bot-wide.
  if (!client.NSFWModel) return
  //if the channel has the NSFW tag, disable the classification.
  if (message.channel.nsfw) return
  let cacheDir = message.settings.modules.attachmentRecorder.storageDir
  let modConfig = message.settings.modules.NSFWClassifier
  //if this module is disabled in the guild, return.
  if (!modConfig.enabled) return
  var classifier = async (client, img) => {
   return new Promise(resolve => {
    try {
     //read the image into a buffer
     const imageBuffer = client.fs.readFileSync(cacheDir + img);

//process the imageBuffer as a 3d tensor (x pixels, y pixels, z pixel depth (e.g. co
lours))
     const image = client.tf.node.decodeImage(imageBuffer, 3, undefined, false);
     //pass the processed data to the neural network for classification.
     client.NSFWModel.classify(image).then(predictions => {
      //resolve the promise once classification completes.
      resolve(predictions)
      //unlink (delete) the processed file from the cacheDir
      client.fs.unlink(cacheDir + filename, (err) => {

if (err) { client.log(err, "ERROR"); } else { client.log("Unlink successful!"); }
      })
      return
     });
    } catch (err) {
     console.error(err);
    }
   });
  };
  /**
```

```
* framework built around the classifier to pipe and parse results into useable infor
mation.
  */
  classifier(client, filename).then(predictions => {
   //logs predictions - debug

client.log(`file ${filename} has results ${JSON.stringify(predictions.join(","))}`)
   //remove SFW classes as they're not helpful.

predictions = predictions.filter(p => p.className != "Neutral" || p.className != "Dr
awing")

if (predictions.filter(p => p.probability >= modConfig.classificationWeights[p.class
Name]).length >= modConfig.thresholdExceeders) {
    let classi = [];
    predictions.forEach(p => {
     //push and format classes
     classi.push(`${p.className} Certainty: ${Math.round(p.probability * 100)}%\n`);
    });
   }
  })
 };

 module.exports.defaultConfig = {
  enabled: true,
  classificationWeights: {
   Hentai: 0.65,
   Porn: 0.7,
   Sexy: 0.8,
  },
  thresholdExceeders: 1,
  autoRemove: false,
  penalty: 5,
  requiredPermissions: ["MANAGE_MESSAGES"],
 };

 NSFWtest.js

 /**Dependancy Import and initialisation */
 const Discord = require("discord.js");
 const tf = require("@tensorflow/tfjs-node");
 const load = require("nsfwjs").load;
```

```
const fs = require("fs");
const client = new Discord.Client({ autoReconnect: true });
client.download = require("download-file");
var modConfig = {}
modConfig.classificationWeights = {
hentai: 0.65,
porn: 0.7,
sexy: 0.8,
}

client.on("ready", () => {
console.log("ready!");
});
/**
* Loads TF GraphModels for NSFW detection.
* @param client
*/
var initmodel = async (client) => {
client.NSFWmodel = await load("file://./models/NSFW/", { size: 299 });
console.log("Model loaded!");
};


var classifier = async (client, img) => {
return new Promise(resolve => {
 try {
 const imageBuffer = fs.readFileSync(`./cache/${img}`);
 const image = tf.node.decodeImage(imageBuffer, 3, undefined, false);
 client.NSFWmodel.classify(image).then(predictions => {
  resolve(predictions);
 });
 } catch (err) {
 console.error(err);
 }
});
};

client.on("message", async (message) => {
if (message.author.bot) return;
message.content = message.cleanContent;
console.log(message.content);
```

```
 message.attachments.array().map(att => att.url).forEach(att => {
  var filename = message.id + "." + att.split("/").pop().split(".")[1];
  var filePath = "./cache/" + filename

client.download(att, { directory: "./cache/", filename: filename, timeout: 999999999
 }, function (err) {
  if (err) {
   client.log(`download of attachment ${att.url} failed!`, "ERROR");
  } else {
   console.log("Download finished:")
   classifier(client, filename).then(predictions => {
   console.log(predictions);

predictions.filter(p => modConfig.classificationWeights[p.class] >= p.probability).l
ength
   let preL = [];
   predictions.forEach(p => {
    preL.push(`${p.className} Certainty: ${Math.round(p.probability * 100)}%\n`);
   });

function Type(p) { if (p[0].className == "Porn" || p[0].className == "Hentai") { ret
urn "NSFW ❌"; } else { return "SFW ✅"; } }
   const exampleEmbed = new Discord.RichEmbed()
    .setColor("#0099ff")
    .setTitle("Image Classification result")
    .setAuthor(`${client.user.username}`)
    .setImage(`${att.url}`)
    .addField(`Classified as ${preL[0]}`, `${Type(predictions)}`)
    .addField("Full classification classes:", `${preL.join(" ")} `)
    .setTimestamp()
   message.channel.send(exampleEmbed);
   }).catch(function (err) {
   console.log(err);
   console.error("Error Parsing Content");
   });
  }

  });
 });
 });

 initmodel(client).then(() => {
```

```
client.login("NjQ4OTU5OTU5NDg4Mzk3MzMy.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ");
});
```

Due to the aforementioned concerns over testing this system, I outsourced it to my Admin Client Group for testing. They provided feedback after conducting tests following my instructions on what types of content to use, and I adjusted the code and parameters of the bot as testing proceeded.

Input	Output	Expected	Bug
Generic SFW image	Error due to incorrect use of then statement for waiting for classifier results	Classified as SFW with very low NSFW class %ages	Lack of promise integration within the classifier function
Generic SFW image	Error due to incorrect tf.node.decodeImage args	Classified as SFW with very low NSFW class %ages	Did not specify channels (4th arg),which is required for the tensor formation
Generic SFW images	Classified as SFW with very low NSFW class %ages	Classified as SFW with very low NSFW class %ages	None
Images considered as SFW with very mild pseudo-NSFW elements	Classified as SFW	Classified as SFW	None
Images considered borderline N/SFW	Generally Classified as NSFW	Generally Classified as NSFW	None
Images considered borderline N/SFW – NSFW threshold weights increased	Generally Classified as SFW	Generally Classified as SFW	None
Images considered NSFW - Sexy	Classified as NSFW – classname (sexy) certainty ~70-90%	Classified as NSFW – classname (sexy) certainty ~80-90%	None – simple weight adjustment
Images considered NSFW - Sexy	Classified as NSFW – classname (sexy) certainty ~80-90%	Classified as NSFW – classname (sexy) certainty ~80-90%	None
Images considered NSFW - Hentai	Classified as NSFW – classname (hentai) certainty of 70-80%	Classified as NSFW – classname (hentai) certainty of 80-90%	None- simple weight adjustments
Images considered NSFW - Hentai	Classified as NSFW – classname (hentai) certainty of 80-90%	Classified as NSFW – classname (hentai) certainty of 80-90%	None – testers noticed more certainty variation in this class compared to the others.

Images considered NSFW - Porn	Classified as NSFW – classname (Porn) certainty of 80-90%	Classified as NSFW – classname (Porn) certainty of 80-90%	None

Table 21 - Prototype 19 Testing Table

Comments:

This system was very difficult to get functioning, with lots of research having to go into how to properly format the data for processing, as NSFWjs was designed as a pure web module and as such lacked the documentation for how to format local files into the correct format. After some research, I discovered that I could skip a few steps and directly convert the image data into a tensor, which allowed me to be able to successfully use this system.

I could in theory further refine the classification weightings by averaging a 'perceived' score (from the Client Admin test group) against the certainty of the system. In the end, I decided not to do this as it would be too time-consuming and if the sensitivity is too high/low, the guild admins can simply change the weightings.

3.20 PROTOTYPE 20 – UGC TOXICITY CLASSIFIER

Goals: implement the ToxicityClassifier

Unlike the NSFWClassifier, I have fewer Qualms about testing this but I decided to outsource the testing as it means a lot more data is checked compared to my own testing - for the calibration of the weightings, this quantity is more important than quality.

Like the NSFWClassifier, this system works by passing text from a message through a Neural Network for Classification into specific classes with a value between 1 and 0 denoting it's certainty that it falls into this class.

The classes for the ToxicityClassifier are more diverse than those for NSFWClassifier, and are as follows:

Identity attack, insult, obscene, severe toxicity, sexual explicit, threat, toxicity – with Toxicity and Severe toxicity being overall evaluations of the other classes. Unlike NSFWc, there are no 'positive' classes present, and the greater class diversity means fine-tuning the overall classification is more granular, although it is arguably much harder to calibrate.

Descriptions of classes (from Training Data Annotation Instructions) are as follows:

Overall toxicity:

Very Toxic - a very hateful, aggressive, disrespectful comment or otherwise very likely to make a user leave a discussion or give up on sharing their perspective.

Toxic - a rude, disrespectful, unreasonable comment or otherwise somewhat likely to make a user leave a discussion or give up on sharing their perspective.

Not toxic – no content that would fit into the above guidelines (note: this is not a class that is present)

Classes:

Profanity/Obscenity: Contains swear words, curse words, or other obscene or profane language.

Sexually explicit: Contains references to sexual acts or body parts sexual way, or other lewd content.

Identity-based attack: A negative, discriminatory or hateful comment about a person or group of people based on criteria including (but not limited to) race or ethnicity, religion, gender, nationality or citizenship, disability, age or sexual orientation.

Insulting: Insulting, inflammatory, or negative comment towards a person or a group of people.

Threatening: describes a wish or intention for pain, injury, or violence against an individual or group.

These were the class definitions used by the annotators to classify the training data for this Neural Net and are thus the definitions I will be using for these classifications.

Pseudocode:

Define async function (client, message)

If client.toxicModel does not exist return

Set modConfig to message.settings.modules.toxicClassifier

If the module is not enabled return

If the message length is not greater or equal to the ignore below threshold, return

Execute client.toxicModel.classify message.content

Then setting predictions as output,

If the number of classes that exceed the certainty threshold is above the threshold,

Define array classi

For every prediction, push the class name and certainty %age to classi

Report as Toxic

Else return

Relevant code

./Overlord.js – this replaces the similar code in the previous prototype

```
/**
 * This function Loads the neural net models for  both the NSFWClassifier as well as
 the ToxicityClassifier.
 * @param {object} client
 */
let modelLoad = async (client) => {
 //checks that the models aren't disabled in the main bot config.
 if (!client.config.enableModels) return;
 client.tf = require("@tensorflow/tfjs-node");

//flag to tell TF that this is a production app. nets some major performance improve
ments.
 client.tf.enableProdMode()
 var toxicity = require("@tensorflow-models/toxicity");
 //loads the NSFWModel via NSFWjs Using local model files.

client.NSFWModel = await require("nsfwjs").load("file://./models/NSFW/", { size: 299
 });
 //creates a new instance of the ToxicityClassifier
 client.toxicModel = new toxicity.ToxicityClassifier;
 //overwrite default LoadModel method due to *hard coded* reliance on web-
based model files.
 // I didn't like this reliance so I made it use local files instead.
 client.toxicModel.loadModel = () => {
  return require("@tensorflow/tfjs-
converter").loadGraphModel("file://./models/toxic/model.json");
 };
```

```
 //wait for the models to load.
 await client.toxicModel.load();
 client.log("Models loaded!");
}
//triggers the loading of the NeuralNet Models. for some reason a self-
invoking anonymous function refused to work.
modelLoad(client)
```

./toxicClassifier.js

```
/**
 * much like NSFWClassifier, classfies text inputs to a set of classes with a %age c
ertainty, but this is used for detecting
 * 'toxicity' in text, eg excessive swearing, racial slurs, and other generally frow
ned upon language.
 * @param {object} client
 * @param {object} message - uses the content of the message.
 */
module.exports = async (client, message) => {
 if (!client.toxicModel) { return } //checks that the Model is loaded.

let modConfig = message.settings.modules.toxicClassifier //gets the modules's config
 if (!modConfig.enabled) { return }

if (!message.content >= modConfig.ignoreBelowLength) { return } //checks ignore leng
th
 let classi = [] //array of classification classes and certainties.
 //classifies the message content using the loaded Model.
 client.toxicModel.classify(message.content).then(predictions => {
  //check the number of threshold exceeders, if above or equal continue

if (predictions.filter(p => p.results[0].probabilities[1] >= modConfig.classificatio
nWeights[p.label]).length >= modConfig.thresholdExceeders) {
   predictions.forEach(result => {
    //format the classifications

classi.push(`${(result.label).replace("_", " ")} Certainty: ${Math.round(result.resu
lts[0].probabilities[1] * 100)}%\n`);
   });
  }
 })
};

module.exports.defaultConfig = {
 enabled: true,
 classificationWeights: {
```

```
  identity_attack: 0.6,
  insult: 0.8,
  obscene: 0.8,
  severe_toxicity: 0.6,
  sexual_explicit: 0.6,
  threat: 0.8,
  toxicity: 0.7,
 },
 thresholdExceeders: 3,
 autoRemove: false,
 ignoreBelowLength: 30,
 penalty: 3,
 requiredPermissions: ["MANAGE_MESSAGES"],
}
```

NSFWTest.js – amended with new toxicity code for testing purposes

```
/**Dependancy Import and initialisation */
const Discord = require("discord.js");
const tf = require("@tensorflow/tfjs-node");
const load = require("nsfwjs").load;
const fs = require("fs");
const client = new Discord.Client({ autoReconnect: true });
client.download = require("download-file");
var modConfig = {}
modConfig.classificationWeights = {
 hentai: 0.7,
 porn: 0.7,
 sexy: 0.9,
 drawing: 0.9,
}

var toxicity = require("@tensorflow-models/toxicity");
client.on("ready", () => {
 console.log("ready!");
});
/**
 * Loads TF GraphModels for NSFW and Toxicity detection.
 * @param client
 */
var initmodel = async (client) => {
 client.NSFWmodel = await load("file://./models/NSFW/", { size: 299 });
```

```
 client.toxicModel = new toxicity.ToxicityClassifier;

client.toxicModel.loadModel = () => { //overwrite default LoadModel method due to *h
ard coded* reliance on web-
based model files. I didn't like this so I made it use local files instead.
 return require("@tensorflow/tfjs-
converter").loadGraphModel("file://./models/toxic/model.json");
 };
 await client.toxicModel.load();
 console.log("Models loaded!");
};


var classifier = async (client, img) => {
 return new Promise(resolve => {
 try {
  const imageBuffer = fs.readFileSync(`./cache/${img}`);
  const image = tf.node.decodeImage(imageBuffer, 3, undefined, false);
  client.NSFWmodel.classify(image).then(predictions => {
  resolve(predictions);
  });
 } catch (err) {
  console.error(err);
 }
 });
};

client.on("message", async (message) => {
 if (message.author.bot) return;
 message.content = message.cleanContent;
 console.log(message.content);
 var classi = [];
 client.toxicModel.classify(message.cleanContent).then(results => {
 console.log(...results);
 results.forEach(result => {

classi.push({ type: result.label, certainty: Math.round(result.results[0].probabilit
ies[1] * 100) });
 });
 console.log(...classi);
 message.reply(JSON.stringify(results));
 });
```

```
 message.attachments.array().map(att => att.url).forEach(att => {
 var filename = message.id + "." + att.split("/").pop().split(".")[1];
 var filePath = "./cache/" + filename

client.download(att, { directory: "./cache/", filename: filename, timeout: 999999999
 }, function (err) {
  if (err) {
  client.log(`download of attachment ${att.url} failed!`, "ERROR");
  } else {
  console.log("Download finished:")
  classifier(client, filename).then(predictions => {
   console.log(predictions);

predictions.filter(p => modConfig.classificationWeights[p.class] >= p.probability).l
ength
   let preL = [];
   predictions.forEach(p => {
   preL.push(`${p.className} Certainty: ${Math.round(p.probability * 100)}%\n`);
   });

function Type(p) { if (p[0].className == "Porn" || p[0].className == "Hentai") { ret
urn "NSFW ❌"; } else { return "SFW ✅"; } }
   const exampleEmbed = new Discord.RichEmbed()
   .setColor("#0099ff")
   .setTitle("Image Classification result")
   .setAuthor(`${client.user.username}`)
   .setImage(`${att.url}`)
   .addField(`Classified as ${preL[0]}`, `${Type(predictions)}`)
   .addField("Full classification classes:", `${preL.join(" ")} `)
   .setTimestamp()
   message.channel.send(exampleEmbed);
  }).catch(function (err) {
   console.log(err);
   console.error("Error Parsing Content");
  });
  }


 });
 });
});

initmodel(client).then(() => {
```

```
 client.login("NjQ4OTU5OTU5NDg4Mzk3MzMy.AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
});
```

Input	Output	Expected	Bug
“Normal” messages with no toxic contents	Processed as non-toxic	Processed as non-toxic	None
Message consisting of a Single-word toxicity	Classification as toxic	Classification as toxic	None – advised by test group that shorter phrases should be ignored. Decided to implement this.
Message consisting of a Single-word toxicity	Not classified as under x chars	Not classified as under x chars	None
Message containing mild overall toxicity	Not classified as toxic	Not classified as toxic	None
Messages containing content that would be considered an identity attack	Classified as toxic – identity attack with a 80+% certainty	Classified as toxic – identity attack with a 80+% certainty	None
Messages containing content that would be considered as insulting	Classified as toxic – identity attack with a 90+% certainty	Classified as toxic – identity attack with a 90+% certainty	None
Messages containing content that would be considered as obscene	Classified as toxic – obscene with a 90+% certainty	Classified as toxic – obscene with a 90+% certainty	None
Messages containing content that would be considered as Sexually explicit	Classified as toxic – Sexually Explicit with a 90+% certainty	Classified as toxic – Sexually Explicit with a 90+% certainty	None
Messages containing content that would be considered as threatening	Classified as toxic – Threatening with a 90+% certainty	Classified as toxic – Threatening with a 90+% certainty	None
Messages containing content that would be considered as Severely Toxic (mix of other categories)	Classified as toxic – Severe Toxicity with a 80+% certainty	Classified as toxic – Severe Toxicity with a 90+% certainty	None – minor weighting tweaks

Table 22 - Prototype 20 Testing Table

Comments:

Due to the much larger data set that this model was trained on, the accuracy and certainty of a prediction were much higher than the NSFWClassifier, leading to only some minor adjustments having to be made over the course of the testing process (via feedback from the testing group). Due to it being purely text based, creating novel content to test it with was possible, with excellent overall results.

3.21 PROTOTYPE 21 – AUTOMATIC MODERATION BACKEND

Goals: Build the automatic moderation backend

So far, all the systems I plan to integrate with the moderation system have been completed, so with these systems in place, it is time to create this system.

My idea for this system is to use specialised objects called ‘actions’. Each action should contain data in the form of properties that allows the moderation system to process the action in a specific fashion. There are a few main processing routes planned for this system, these are outlined in the below diagram:

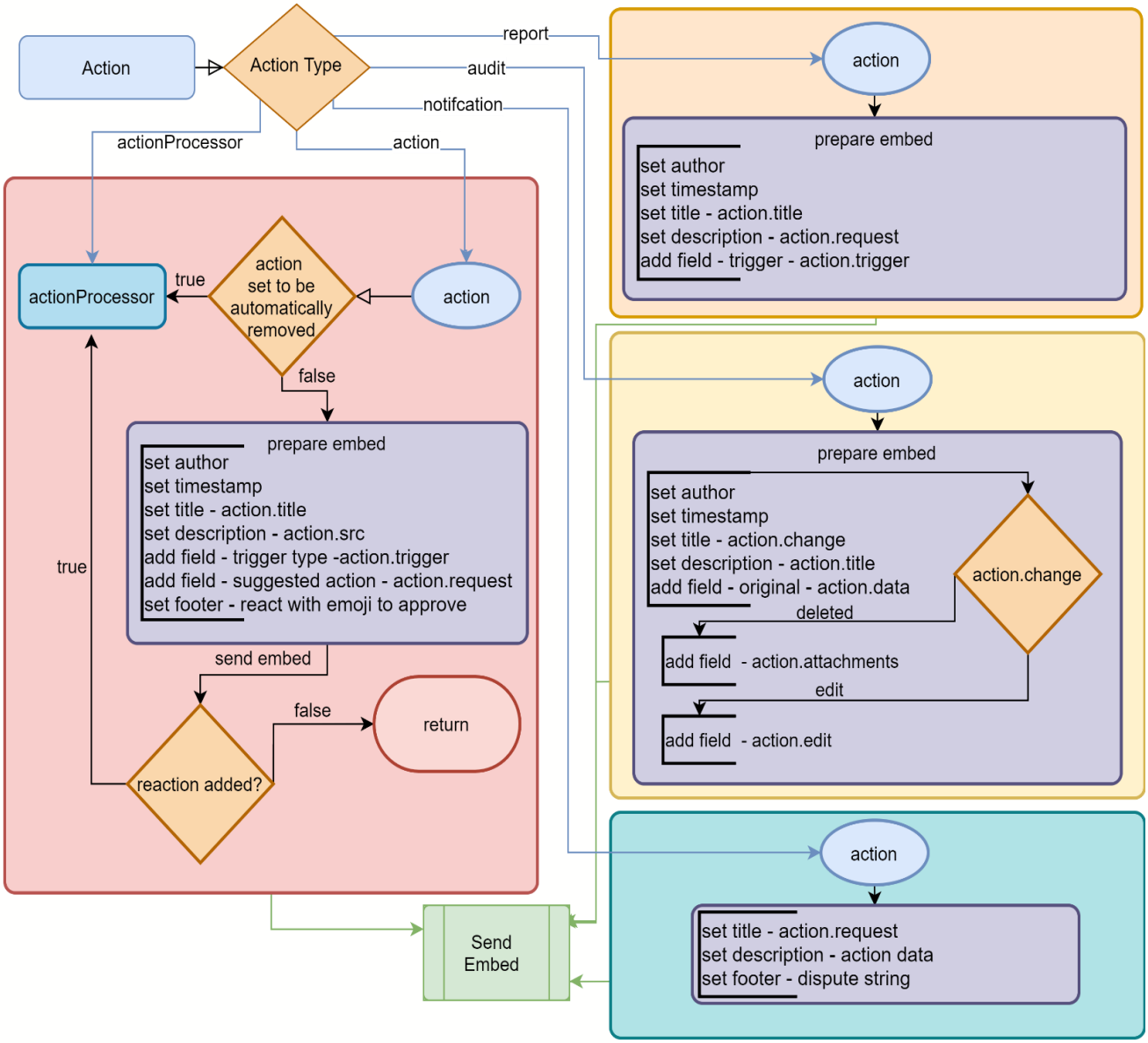

Figure 22 - Automated Moderation Action Flow Diagram

This diagram shows the planned top-level execution paths for the bot – these paths are for various types of action. The type of action simply denoted as ‘action’ is one that another subsystem is ‘advising’ the moderators of the server to take. This system exists due to the desire for human moderator oversight due to the inherent inaccuracies of automated systems, whilst providing them with sufficient information that they are able to make a confident decision just from the information presented by the bot reports are not entirely for moderators but serve more as a transparency mechanism towards any changes that occur. The audit system is like the reporting system, but is for moderators only and contains information that may be sensitive, e.g. message contents from all channels of the server. Its primary use is for logging edits and deletions of messages by users,

and hooks into the attachment recorder system to provide links to attachments upon message deletion. The notification system is a mechanism by which the bot directly (via direct messaging) informs users of any punishments or actions taken, e.g. notification of a mute, kick, tempBan or Ban. This allows the user, in the case of a kick/ban where they can no longer access the server to figure out why they were kicked/banned, to understand the reason why they were kicked/banned directly. Lastly, the action Processor system is the subsystem responsible for processing requested actions, such as those from scheduled actions or from moderator-approved actions. This system links back into the reporting and notification systems as a way of informing users about what it's doing and to whom and why.

Due to the variation within action objects, I have decided on a few baseline properties, with other properties being able to be freely added/removed as systems require them - provided they do not interfere with other existing systems. The current must-have data elements are the type, guildID, memberID, title, trigger :{ type, data} and executor. These requirements or other outlined behaviours may change as the system(s) develop, as this is by far the most interdependent system planned for Overlord.

Pseudocode:

./events/modActions.js

Define async function (client,action)

Set Discord to require module discord.js

Set guild to client.guilds.get action.guildID

Set target to guild.member.get action.memberID

Set config to client.DB.get guild.id

Set modAction to config.modActionChan

Set audit to config.auditChan

Set modReport to config.modReportChan

Define function genModAction (client,action)

If action.autoremove is true, execute actionProcessor (client,action) else;

Set embed to new discord.richEmbed

setAuthor to the bot User

setTimestamp to now

setTitle to action requested + action.title

setDescription to source + action.src

addFields

trigger type : action.trigger.type

trigger description: action.trigger.data

recommended action pending approval : action.request

setFooter react with (emoji) to perform the recommended action

sent embed to channel modActions then

create new ReactionCollector on sent embed

upon a correct emoji reaction being added,

change the executor to the user reacting with the emoji, change the trigger type to automatic

execute actionProcessor (client,action)

define function genAudit (client,action)
set embed to new discord.richEmbed
setAuthor to the bot user
setTimestamp to now
setDescription to action.title
addField original : action.data formatted as a code Block for increased visibility.
Declare switch action.change
Case deleted
Add field attachments : action.attachments
Case edited
Add field edited contents : action.edit
Send embed to channel audit

Define function genNotification
Set notification to new discord.richEmbed
setTitle notification of action + action.request
setDescription in server +guild+ action.src + trigger: +action.trigger.type, executor: +action.executor
setFooter If you wish to dispute this action, please contact an Administrator
send notification to target

define function actionProcessor
if action has a penalty property defined,
set data to client.DB.guildID, ensure data {count:0, TS: now} at users.memberID.demerits
client.DB.guildID set data {count: data.count+penalty, TS: now} at users.memberID.demerits
execute client.schedule {type:demerit, end now + 2000 milliseconds, memberID: memberID} for guild guildID
else;
define switch (action.requestedAction.type)
case delete (client,action)
execute function deleteMessage
case mute (client,action)
execute function mute
case ban (client,action)
execute function ban
case bulkDelete (client,action)
execute function bulkDelete
if none of the cases apply, log the case as log type WARN

define function mute (client,action)
add role config.mutedRole to target

execute genNotification (client,action)

execute genModReport (client,action)

define function deleteMessage (client,action)

set args to action.requestedAction.target split by char “.”

Set message to client.guilds.get(args[0]).channels.get(args[1]).messages.get(args[2])

Execute message.delete

execute genNotification (client,action)

execute genModReport (client,action)

define function ban (client,action)

if the target is not able to be banned,

log the error as log type WARN

else;

try to;

execute genNotification (client,action)

then client.ban (reason: action.title)

execute genModReport(client,action)

if an error occurs,

log the error as log type ERROR

define function bulkDelete (client,action0

set channel to guild.channels.get action.requestedAction.target split by char “.” [1]

set toDelete to execute channel.fetchMessages filtered where message.author is equal to member.id and the number of messages is below action.requestedAction.count

execute channel.bulkDelete toDelete

define switch action.type

case action

execute genModAction (client,action)

case audit

execute genAudit (client,action)

case report

execute genModReport (client,action)

case actionProcessor

execute actionProcessor (client,action)

if none of the cases match, log case with log type ERROR

define defaultConfig.requiredPermissions as managemembers, manageguild

relevant code

./events/modActions.js

```
/**
 * processes an 'action' object containing data to perform a specific action through
 various process control measures (eg optional human intervention)
 * @param {object} client "global scope" for the application. manditory.
 * @param {object} action custom object containg pseudo-
standardised data. see the basic schema below.
*/
module.exports = async (client, action) => {
 //re-imported due to needing embeds (can't access through Client Object).
 let Discord = require("Discord.js")
 let guild = client.guilds.get(action.guildID)
 let target = guild.members.get(action.memberID)
 let config = client.DB.get(action.guildID)
 let modAction = guild.channels.get(config.modActionChan)
 let audit = guild.channels.get(config.auditLogChan)
 let modReport = guild.channels.get(config.modReportingChan)
 //if any of the channels are undefined, resets them to the first valid channel.
 if (!modAction || !audit || !modReport) {

//multiAssign as well as a map iterator 'trick' to get the first valid channel witho
ut having to get the key.

modAction = audit = modReport = guild.channels.filter(chan => chan.type === "text").
values().next().value
  //notify administrators (and bot owner) that the channels are undefined.

client.log(`Undefined Reporting channels for guild ${guild.name} - falling back to f
irst valid channel`, "WARN")

modReport.send("WARNING: Reporting channels Have not been configured properly or I d
on't have access to them!")
 }
 //https://leovoel.github.io/embed-visualizer/ - link used to design embeds.


 /* example of an action:
 {
  guildID: message.guild.id,
  memberID: message.member.id,
  type: "action",
  autoRemove: modConfig.autoRemove,
```

```
  title: "Suspected Toxic Content",

src: `Posted by user ${message.author} in channel ${message.channel} : [Jump to mess
age](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `Toxic content breakdown: \n${classi.join(" ")}`,
  },
  request: "Removal of offending content",
  requestedAction: {
   type: "delete",
   target: `${message.guild.id}.${message.channel.id}.${message.id}`,
  },
  penalty: modConfig.penalty,
 }*/


 /**
  * processes an action object to request approval for an action from a moderator.
  * @param {object} client
  * @param {object} action - the action object to be processed as type modAction
  */
 const genModAction = async (client, action) => {

//if the action doesn't require user input, bypass this system and directly execute.

if (action.autoRemove) { actionProcessor(client, { ...action, ...{ executor: client.
user } }) } else {
   //create a new embed instance
   let embed = new Discord.RichEmbed()
    //add elements to the embed
    //set the author to the bot
    .setAuthor(client.user.username, client.user.avatarURL)
    //set the timestamp to now
    .setTimestamp(new Date())
    //set the title
    .setTitle(`Action requested: ${action.title}`)
    //set the description (main body of text)
    .setDescription(`source: ${action.src}`)
    //add fields for additional information
    .addField("Trigger type", action.trigger.type)
    .addField("Trigger description:", action.trigger.data)
    .addField("Recommended Action Pending Approval:", action.request)
```

```
    .setFooter("react with ✅ to perform the reccomended action.")

modAction.send({ embed: embed }).then(msg => { //wait for the message to be sent and
 reacted to.

msg.react("✅").then(() => { //waits for a ✅ reaction, then executes the requested
 action.
     //checks that only one user (not a bot) reacts with a ✅.

msg.awaitReactions((reaction, user) => { return reaction.emoji.name === "✅" && !us
er.bot }, { max: 1 })

.then(collected => { //passes through all results that passed the above filter funct
ion.
       //gets the userID of the user that reacted.

action.executor = client.guilds.get(action.guildID).members.get(Array.from(collected
.get("✅").users)[1][1].id)

action.trigger.type = "Manual" //changes trigger type as it had moderator/human inpu
t.
       actionProcessor(client, action)
      })
    })
   }).catch(err => {
    //catches and logs any errors
    client.log(err, "ERROR")
   })
  }
 }


 /**
  * processes an action object to generate an audit entry - showing that something h
as changed and what that change is.
  * generates and sends an embed so that moderators can monitor any chnages, eg mess
age deletions/edits, etc.
  * @param {object} client
  * @param {object} action - action object to be processed to produce an auditLog ch
annel entry.
  * (not to be confused with the server's built-in audit log).
  */
 const genAudit = async (client, action) => {
  //create a new embed instance
  let embed = new Discord.RichEmbed() //add attributes
```

```
    .setAuthor(client.user.username, client.user.avatarURL)
    .setTimestamp(new Date())
    .setTitle(`Change: ${action.change}`)
    .setDescription(action.title)

//specialised formatting used to generate 'code blocks' - help with contrast and bre
aking text up.
    .addField("Original:", `\`\`\`\n${action.data}\n\`\`\``)

switch (action.change) { //switch:case for specific types of change needing specific
 fields.
    case "deleted":
     //list attachments that were saved and re-uploaded.
     embed.addField("Attachments:", `${action.attachments.join("\n") || "None"}`)
     break;
    case "edited":
     //add a field showing the edited contents.
     embed.addField("Edited Contents:", `\`\`\`\n${action.edit}\n\`\`\``)
     break
  }
  //send the embed to the audit log channel.
  audit.send({ embed: embed }).catch(err => {
   //catch and log any errors
   client.log(err, "ERROR")
  })
 }


 /**
  * processes an action object to create a report of a moderator action - this is su
pposed to be 'public facing'
  * @param {object} client
  * @param {object} action - action to be processed into a report.
  */
 const genModReport = async (client, action) => {
  //create a new embed - add attributes
  let embed = new Discord.RichEmbed()
    .setAuthor(client.user.username, client.user.avatarURL)
    .setTimestamp(new Date())
    .setTitle(`Action: ${action.title}`)
    .setDescription(`Action description: ${JSON.stringify(action.request)}`)

.addField(`action trigger: ${action.trigger.type}`, `Executor: ${action.executor}`)
```

```
  //send embed to the modReport channel.
  modReport.send({ embed: embed }).catch(err => {
   client.log(err, "ERROR")
  })
 }


 /**
  * processes an action object to generate a nortification for a user to notify them
 of an action having taken place.
  * such as being muted or banned.
  * @param {object} client
  * @param {object} action
  */
 const genNotification = (client, action) => {

//this is a promise to ensure that the ban does not get triggered before the user is
 notified.
  return new Promise(resolve => {
   //create new embed instance, add attributes to it.
   let notification = new Discord.RichEmbed()
   notification
    .setTitle(`Notification of action: ${action.request}`)

.setDescription(`In server ${guild}\n - ${action.src}.\n trigger: ${action.trigger.t
ype}, executor: ${action.executor}.`)

.setFooter("If you wish to dispute this action, please contact an Administrator.")
   target.send({ embed: notification }).then(() => {
    resolve("Sent!")//placeholder value to resolve the promise.
   })
  })
 }


 /**
  * processes an action's requested action after it has been approved.
  * eg mutes the user, bans the user, deletes some messages.
  * @param {object} client
  * @param {object} action
  */

let actionProcessor = async (client, action) => { //proccessing manager/discriminato
r for actions,
```

```
if (action.penalty) { //if the action has a 'penalty' attached to it, fork the objec
t to the scheduler to apply punishments.

//ensure the user has a demerits entry. if they did, return the value, else set to t
he provided object.

let data = client.DB.ensure(guild.id, { count: 0, TS: new Date() }, `users.${action.
memberID}.demerits`)
   //update the user with the new number of demerits and timestamp.

client.DB.set(guild.id, { count: data.count + action.penalty, TS: new Date() }, `use
rs.${action.memberID}.demerits`)
   //schedule the action to happen immeadiatly.

client.schedule(guild.id, { type: "demerit", end: new Date(), memberID: action.membe
rID, })
  }


  /**
   * actions: objects containing data to be done 'at some point', whether via a sche
duler or otherwise.
   * categorised by type, and in this case, by the type of the requested Action.
   */
  switch (action.requestedAction.type) {
   case "delete":
    deleteMessage(client, action)
    break
   case "mute":
    mute(client, action)
    break
   case "ban":
    ban(client, action)
    break
   case "bulkDelete":
    bulkDelete(client, action)
    break
   default:

client.log(`unknown/invalid action type: ${action.requestedAction.type}`, "WARN")


  }


  /**
```

```
   * given a target and a guild, mutes the target in the guild, notifies, and then r
eports.
   * @param {*} client
   * @param {*} action
   */

function mute(client, action) { //given a target, mutes then notifies the target of
the action.
   target.addRole(config.mutedRole) //gives the target the specified muted role.
   genNotification(client, action)
   genModReport(client, action)
  }


  /**
   * given a target message in 'full path' notation, deletes the message, notifies t
hen reports.
   * @param {*} client
   * @param {*} action
   */
  function deleteMessage(client, action) {
   let args = action.requestedAction.target.split(".")

//deletes the specified message using it's "Full Path" (guildID.channelID.messageID)
   client.guilds.get(args[0]).channels.get(args[1]).messages.get(args[2]).delete()
   genNotification(client, action)
   genModReport(client, action)


  }


  /**
   * given a target user, checks they can be banned before notifying, banning and th
en reporting.
   * notifying before it does so as it cannot DM banned users.
   * @param {object} client
   * @param {object} action
   */
  async function ban(client, action) {
   //check the both can actually ban the user
   if (!target.bannable) {
    client.log(`Cannot Ban user ${target} - I do not have the permissions!`, "WARN")
    return
   } else {
```

```
try { //catch in case the user has already been banned or some other error occurs.
     //generate and send the notification of action before banning the user.
     genNotification(client, action).then(() => {
      //ban the user
      target.ban({ reason: action.title })
     })
     //genreate a moderation report
     genModReport(client, action)
    } catch (err) {
     client.log(err, "ERROR")
    }
   }
  }
  /**
   * given a user as well as a channel, and a count, deletes the last count messages
 by that user in the specified channel.
   * where count is action.requestedAction.count, then notifies and reports.
   * @param {object} client
   * @param {object} action
   */
  async function bulkDelete(client, action) {
   //get the channel
   let channel = guild.channels.get(action.requestedAction.target.split(".")[1])
   //wait for the messages to be fetched and filtered by authorID

let toDelete = await ((channel.fetchMessages().then(messages => messages.filter(m =>
 m.author.id === target.id))))
   //deletes all passed messages up to count.
   channel.bulkDelete(toDelete.array().splice(0, action.requestedAction.count))
   //generate a notification and modReport
   genNotification(client, action)
   genModReport(client, action)


  }
 }


switch (action.type) { //switch:case for flow control to descriminate against differ
ent types of actions
  case "action":
   genModAction(client, action)
```

```
    break;
   case "audit":
    genAudit(client, action)
    break;
   case "report":
    genModReport(client, action)
    break;
   case "actionProcessor":
    actionProcessor(client, action)
    break;
   default:

client.log(`incorrect action type ${action.type}! event processing failed.`, "ERROR"
)
    break;
  }

}
module.exports.defaultConfig = {
 requiredPermissions: ["MANAGE_MEMBERS", "MANAGE_GUILD"],
}
```

Input	Output	Expected	Bug
Action Object – type: action – missing key properties (guildID, memberID)	Execution fails – logging channels undefined - error logged	Execution fails, error logged	No fallback in case of lack of config values – implemented code to do this.
Action Object – type: action – missing key properties (guildID, memberID)	Execution fails, error logged	Execution fails, error logged	No fallback in case of lack of config values – implemented code to do this.
Action object – type: action, requestedAction delete	Action embed sent and once reacted to, target message was deleted. Reporting system triggered	Action embed sent and once reacted to, target message was deleted. Reporting system triggered	None
Action Object – type: action requestedAction mute	Action embed sent and once reacted to, target was muted. Reporting system triggered	Action embed sent and once reacted to, target was muted. Reporting system triggered	None

Action Object – type: action requestedAction ban	Action embed sent – once reacted to, ban failed due to target not being bannable	Action embed sent – once reacted to, target was banned. Reporting system triggered	Didi not check if the target can actually be banned by the bot before banning.
Action Object – type: action requestedAction ban	Action embed sent – once reacted to, target was banned. Reporting system triggered	Action embed sent – once reacted to, target was banned. Reporting system triggered	None
Action Object – type: action requestedAction bulkDelete	Action embed sent – Once reacted to the last x messages sent by the target were deleted. Reporting system triggered	Action embed sent – Once reacted to the last x messages sent by the target were deleted. Reporting system triggered	None
Action Object – type: action requestedAction: ThisIsInvalid	Action embed sent – once reacted to invalid RequestedAction was logged.	Action embed sent – once reacted to invalid RequestedAction was logged.	None
Action Object – type: audit, invalid data	Execution failed due to undefined data – error logged	Execution failed due to undefined data – error logged	None
Action Object – type: audit, change: None	Audit embed created and sent. No change fields added.	Audit embed created and sent. No change fields added.	None
Action Object – type: audit, change: Edited	Audit embed created and sent. Edited field and content added	Audit embed created and sent. Edited field and content added	None
Action Object – type: audit, change: Deleted	Audit embed created and sent. Deleted field and content added	Audit embed created and sent. Deleted field and content added	None
Action Object – type: report – invalid data	Execution failed due to undefined data – error logged	Execution failed due to undefined data – error logged	None
Action Object – type: report	Embed sent, correct formatting and fields	Embed sent, correct formatting and fields	None
Action Object – type: actionProcessor – invalid data	Execution failed – error logged.	Execution failed – error logged.	None

Action Object – type: actionProcessor – action x	For x action, correct processing occurred.	For x action, correct processing occurred.	None (note: no need for exhaustive re-testing as these were indirectly tested earlier)

Table 23 - Prototype 21 Testing Table

Embeds:

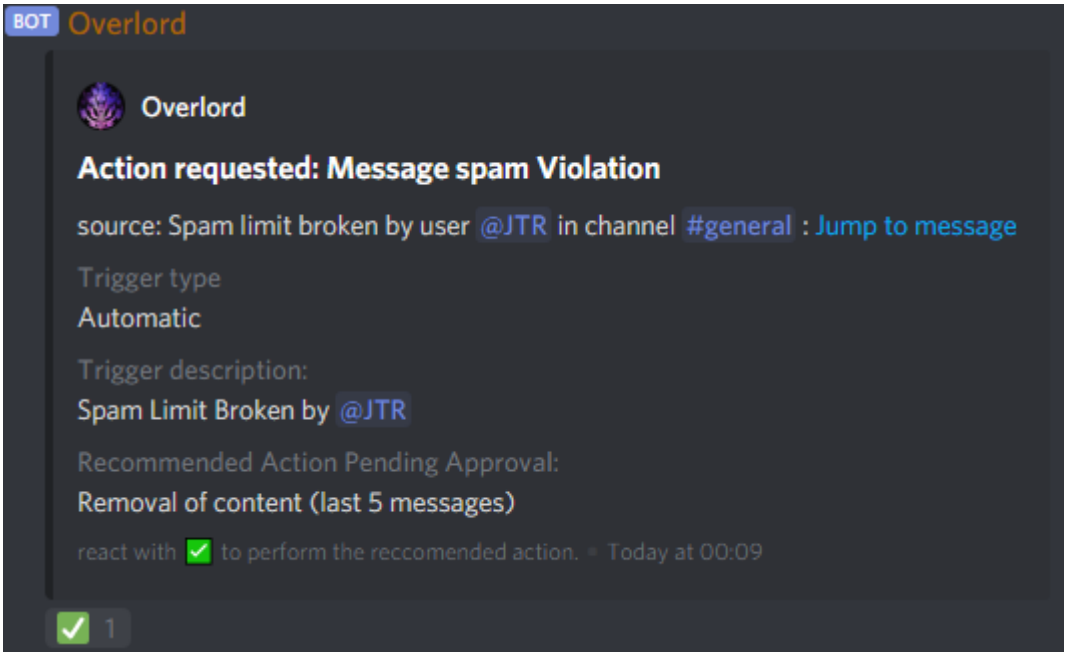

Figure 23 - Example Action Embed

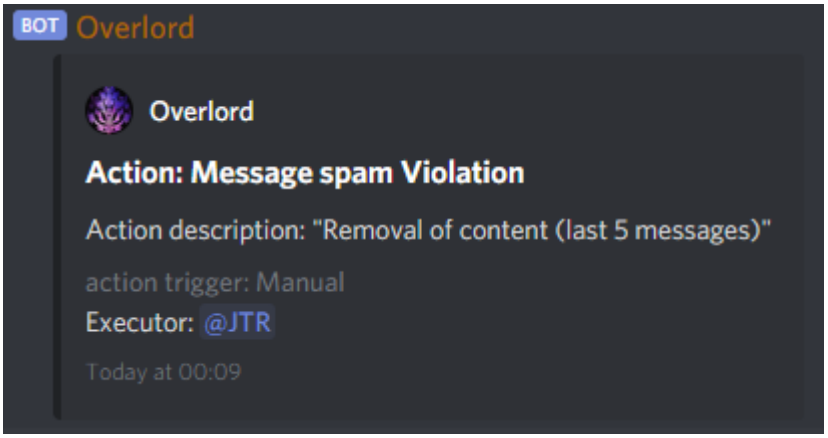

Figure 24 - Example Report Embed

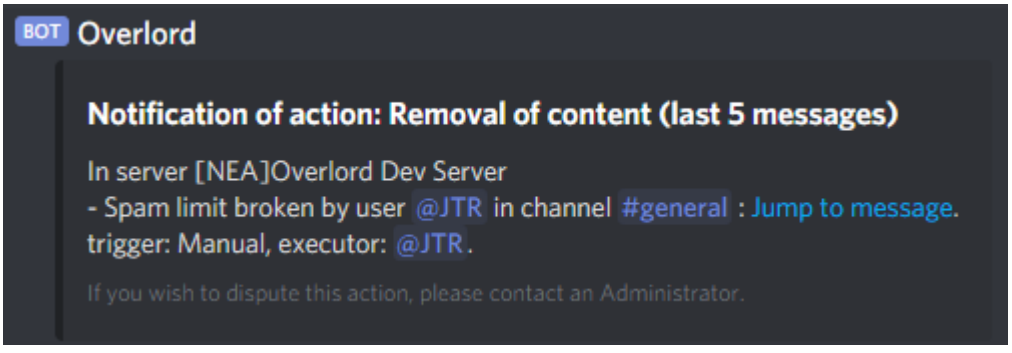

Figure 25 - Example Notification Embed

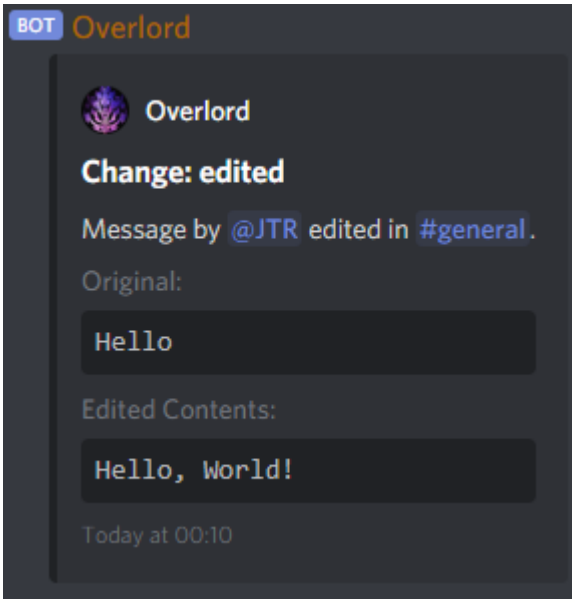

Figure 26 - Example Audit Embed

Comments:

Other than some bugs, the overall process and algorithm works well and as intended, providing a versatile method by which moderation systems can process various events. There could be more room for improvement by, for example, providing an annotated class for the action to act as a template for all the systems requiring it – greatly streamlining the process of creating an action object. However, this system does not implement the system that actually manages punishments, just moderation actions.

3.22 PROTOTYPE 22 - PUNISHMENT

Goals: implement punishment system within scheduler

This prototype seeks to integrate punishment systems within the scheduler submodule to properly integrate the planned algorithms for the demerits/points system.

Requirements

Action object from scheduler passed for processing:

Given the type of action, a specific function needs to be executed, planned functions include:

roleRemove : remove a role from a Member

reminder: remind a user (send them a Direct Message with a predefined message)

nick: change a Members nickname

roleAdd: Add a role to a Member

unban: remove the ban of a Member from a specific server, allowing them to rejoin.

demerit: process the demerits for the current Member, and add/remove punishments as required.

Demerit is the only really complex system out of these, and follows the following process:

Load required punishment data - determine the number of decay cycles between the last decay cycle (timestamp in the user's demeritentry) and now.

(a decay cycle is an interval of x ms, e.g. 3,600,000ms - 1 hour)

For every punishment tier defined for the guild,

If the user was in the range of demerits for the current punishment tier but is no longer, revert the punishment via modActions.

If the user was not in the range of demerits for the current punishment tier, but now is, apply the punishment via modActions.

If the user has reached 0 demerits, delete their demerits entry.

Else; redefine the demerits entry to the count after the number of decay cycles.

Define a new action to occur after 1 decay cycle, that will re-evaluate the current user

Push this new action to the DB, removing any duplicate copies.#

Create a new function that checks if a value in in range of two other values.

Pseudocode:

Define function inRange x, min, max

Return (x-min)*(x-max) <0

Define function actionProcessor (client,guildID,action)

Set guild to guildID from client.guilds

Set target to membderID from guild.members

Define switch action.type

Case roleRemove:

Get target, remove role action.roleID from target

Case reminder:

Get target, send them the message action.message

Case nick:

Get target, set nickname to action.nick

Case roleAdd:

Gat target, add role action.roleID

Case unban

Get guild, unban target

Case demerit:

Set mConf to guildID.modules.autoMod from client.DB

Set data to ensure users.memberID.demerits with client.DB

Set intervals to the max value between(the floor of the current day minus data.ts) divided by (3600000 times mConf.delay) ,and 0.)

Set afterDecay to the max value between data.count-intervals, and 0

Set pAction to new Object: with type, memberID, guildID, trigger, src and executor properties.

For every punishment tier (pt) in mConf;

If inRange(data.count,pt[1].end,pt[1].start) and !inRange(afterDecay,pt[1].end,pt[1].start)

Define switch pt[0]

Case mute:

actionProcessor client, guildID, action

case tempBan:

actionProcessor client,guildID,action

else if !inRange(data.count, pt[1].end, pt[1].start) and afterDecay > pt[1].start;

define case pt[0]

case mute:

client.emit modactions …mute

case tempBan:

client.emit modactions …tempban

case ban:

client.emit modactions …ban

if afterDecay <=0;

remove target.demerits from client.DB.guildID

else;

set target.demerits to count: afterDecay, TS: now in client.DB.guildID

action.end = now + mConf.delay hours

set persistence.time to persistence.time with any actions that have the same memberID as action.memberID being removed.

Execute scheduler.

Relevant code:

./events/scheduler.js

```
/**
 * givens a min, max, and value to test, determines if a value is in range or not.
 * @param {number} x value that you want to check
 * @param {number} min the minimum of the range
 * @param {number} max the maximum of the range
 */
function inRange(x, min, max) {
 return ((x - min) * (x - max) < 0);
}
/**
 * processes actions that need to be executed by the scheduler, such as punishments,
 role removals, reminders, etc.
 * @param {object} client
 * @param {string} guildID
 * @param {object} action - object containing data for processing.
 */
let actionProcessor = async (client, guildID, action) => {
```

```
 let guild = client.guilds.get(guildID)
 let target = guild.members.get(action.memberID)
 //switch:case discrimnator for each action type
 switch (action.type) {
  case "roleRemove": //removes a given role from a given user
   target.removeRole(action.roleID)
   break
  case "reminder": //sends a given user a given string as a reminder.
   target.send(`scheduled reminder: ${action.message}`)
   break
  case "nick": //sets the nickname of a given user to a given string
   target.setNickname(action.nick)
   break
  case "roleAdd": //add a given role to a given user
   target.addRole(action.roleID)
   break;
  case "unban": //unbans a given user
   guild.unban(action.memberID)
   break;
  case "demerit":
   let mConf = client.DB.get(guild.id).modules.autoMod

let data = client.DB.ensure(guildID, { TS: new Date(), count: 0 } `users.${action.me
mberID}.demerits`)

//determine how many decay cycles have occured between this execution and last execu
tion.

let intervals = Math.max(Math.floor((new Date() - data.TS) / (3600000 * mConf.decay)
), 0)

//determine the new demerit count after the number of cycles, bounding it to be >=0.

//this system was made so that perminent punishments (eg permbans) can be given an e
nd of -1 to be infinite.
   let afterDecay = Math.max((data.count - intervals), 0)
   let pAction = {
    type: "actionProcessor",
    memberID: action.memberID,
    guildID: guildID,
    trigger: {
     type: "Automatic"
    },
```

```
    src: "Automated punishment due to reaching demerit threshold.",
    executor: client.user
   } //punishment action - variant of action
   //iterate over all the punishment tiers
   Object.entries(mConf.punishments).forEach(pt => {
    //if it was originally in range (count, pt[1]-start-
end), but is no longer (afterdecay) in range, undo the action/punishment.

if (inRange(data.count, pt[1].end, pt[1].start) && !inRange(afterDecay, pt[1].end, p
t[1].start)) {

client.log(`Removing Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
     switch (pt[0]) {
      case "mute":

actionProcessor(client, guildID, { memberID: action.memberID, type: "roleRemove", ro
leID: guild.mutedRole })
       break;
      case "tempBan":

actionProcessor(client, guildID, { memberID: action.memberID, type: "unban" })
       break;
     }
     //if it wasn't in range and is now above the start, apply the punishment.

} else if (!inRange(data.count, pt[1].end, pt[1].start) && afterDecay > pt[1].start)
 {

client.log(`Applying Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
     //switch:case for the name of the punishment
     switch (pt[0]) {
      case "mute":
       client.emit("modActions", {
        ...pAction, ...{
         title: "Mute of User",
         requestedAction: {
          type: "mute"
         },
         request: `Mute of user ${target.tag}`,
        }
       })
       break;
```

```
      case "tempBan":
       client.emit("modActions", {
        ...pAction, ...{
         title: "Temporary ban of User",
         requestedAction: {
          type: "ban"
         },
         request: `Temporary ban of user ${target.tag}`,
        }
       })
       break;
      case "ban":
       client.emit("modActions", {
        ...pAction, ...{
         title: `Ban of User ${target.tag}`,
         requestedAction: {
          type: "ban"
         },
         request: "Ban of user",
        }
       })
       break;
     }
    }
    if (afterDecay <= 0) { //if the user is out of demerits, delete the property.
     client.DB.remove(guildID, `users.${action.memberID}.demerits`)
    } else {

//set the demerits property of the user to a modified version, with the new counts a
nd Timestamp.

client.DB.set(guildID, { count: afterDecay, TS: new Date() }, `users.${action.member
ID}.demerits`)
    }
   })


   action.end = new Date().setTime(new Date().getTime() + (mConf.decay * 3600001))

//reschedule the action, after filterrng out any duplicate entries (as these contain
 non-essential data)

client.DB.set(guildID, client.DB.get(guildID, "persistence.time").filter(act => act.
memberID !== action.memberID), "persistence.time")
```

```
    //re-schedule the action.
    client.schedule(guildID, action)
    break
   default:
    client.log(`Unknown action type/action ${JSON.stringify(action)}`, "WARN")
    break;
  }
}
```

Input	Output	Expected	Bug
Execution with missing/invalid data (memberID, guildID)	Execution failed, Error Caught and logged	Execution failed, Error Caught and logged	None
Execution with action type roleRemove	Specified user had the specified role removed.	Specified user had the specified role removed.	None
Execution with action type reminder	Specified user was sent predetermined string via Direct messaging	Specified user was sent predetermined string via Direct messaging	None
Execution with action type nick	Specified user's nickname was changed.	Specified user's nickname was changed.	None
Execution with action type roleAdd	Specified user had specified role added.	Specified user had specified role added.	None
Execution with action type unban	Specified user was successfully unbanned.	Specified user was successfully unbanned.	None
Execution with action type demerit – 0 demerits	User had 0 demerits and as such had no punishments added.	no punishments added.	None
Execution with action type demerit – 5 demerits	User was not yet in range of lowest punishment tier - no punishment applied.	no punishment applied.	None
Execution with action type demerit – 6 demerits	User in range for mute tier – applied for 6 decay cycles	Mute applied for 6 decay cycles	None
Execution with action type demerit – 10 demerits	User in range for mute – applied for 10 cycles	Mute applied for 10 cycles	None
Execution with action type demerit – 11 demerits	User in range for mute and tempBan – mute for 11 cycles, tempBan for 6 cycles.	Mute applied for 11 cycles, tempBan applied for 6.	None

Execution with action type demerit – 15 demerits	User in range for mute and tempBan – mute for 15 cycles, tempBan for 10 cycles.	Mute applied for 15 cycles, tempBan applied for 10.	None
Execution with action type demerit – 16 demerits	User in range for mute, tempBan and permBan – mute for 16 cycles, tempBan for 11 cycles, and perm indefinitely.	Mute applied for 16 cycles, tempBan applied for 11 permBan indefinitely.	None – note that the permBan has an 'end' value of -1, denoting an infinite duration. As tempBan is handled through a role and not an actual ban, tempBan will not interfere.
Execution with action type demerit – 16+ demerits	User in range for all default punishments, applied based on ranges until they decay to 10 and 5 for tempBan and mute.	User in range for all default punishments, applied based on ranges until they decay to 10 and 5 for tempBan and mute.	None

Table 24 - Prototype 22 Testing Table

Comments:

This subsystem was one of the more time consuming as far as design went, with multiple iterations of how to implement the points system whilst integrating it with existing systems and making it as versatile as possible. This iteration was the one I decided on due to its intuitive 'range' system for defining a range of values for punishments to start and take effect until, as well as its heavy integration with the scheduler and modActions instead of its own self-contained system for punishments.

3.23 PROTOTYPE 23 – FUNCTION COMPILATION

Goals: compile initialisation routine and other miscellaneous functions into ./Functions.js

Functions.js is a file that contains function code that is bound to the client variable at runtime as methods/properties. As some of these functions are larger and do not fit into other sections or categories of functions, It was decided to compile them into this singular file. Includes some new and some already developed functions/concepts.

Functions.js is planned to include the following:

Client.init – new – includes code from Prototype 5

Client.validateGuild - new - includes code from Prototypes 3 and 6

Client.log – see prototype 7

Client.canExecute – see prototype 12

Client.loadCommand – see prototype 14

Client.reloadCommand – see Prototype 14

Client.evalClean – new – not my own code so will not be fully covered.

Client.getRandomInt – new

As well as these functions, it introduces some more data structures that the bot requires.

Client.init

Requirements:

Check valid client state (e.g. token user)

Set status

Load and bind event files

Trigger guild validation for all guilds

Notify owner of start-up

Pseudocode:

Define function client.init (client)

If the bot is in debug mode, log it with flag WARN

Log the tag of the user the client had logged into

If the bot is not part of any guilds, or logged into a user account, throw an error. This is because operating a bot on a user account is against discord ToS.

If the client is configured to preload in config.js;

For every channel that the bot has access to, fetch the last 100 messages to the message cache.

Set the status of the bot to the configured string with the replacements.

Set eventFiles to every file located in the ./events directory

Log that the bot is loading event files as DEBUG

For every eventFile in eventFiles;

If the file does not end in .js return

Set eventName to eventFile with the extension (.js) removed

Log that the event loading is being attempted as DEBUG – useful if it fails as then you know which event failed.

Set eventObj to ./events/eventFile

When client emits event eventName, execute eventObj, passing it through the relevant data with client as the first arg.

If the eventObj has a defaultConfig,

For every guild the client is a member of,

Ensure that guild has the config as client.DB.guildID.modules.eventname.config

Log that the loading was successful as DEBUG

For every guild the client is a member of;

Execute client.validateGuild (client,guild)

For every file in ./cache,

Delete file – these are files that were left over (probably from a crash) and so need to be removed.

End the initialisation timer

Log that the bot has fully initialised and is ready as INFO

Try to send a notification of readiness to the bot owner, if it fails log as INFO

Relevant code for this section (client.init):

./functions.js:init

```
**
  * Initialisation routine for the client, after the client has been authenticated a
nd connected to discord,
  * as well as after the ENMAP database is ready. handles the initialisation of ever
ything needed for proper function.
  */
 client.init = (client) => {
  //statement to make the owner aware the bot is in debug mode.
  if (client.debug) { client.log("BOT IS IN DEBUG MODE", "WARN") }
  //logging statement to see what user is being logged into.
  client.log(`client logging in as ${client.user.tag}`, "INFO")
  if (client.guilds.size == 0) {
   //aborts if the bot is not part of any guilds
   throw new Error("No Guilds Detected! Please check your token. aborting Init.");
  }
  if (!client.user.bot) {
   //aborts if it detects it has been given a user's token instead of a bot's token.

throw new Error("Warning: Using  bots on a user account is (for the most part) forbi
dden by Discord ToS. Please Verify your token!");
  }

//if preloading is enabled, iterate over every channel and load messages into bot me
ssage cache.
  if (client.config.preLoad) {
   client.channels.forEach(channel => {
    //categories are classed as channels and are thus ignored.
    if (channel.type === "text") {
     channel.fetchMessages({ limit: 100 }).then(c => { return })
    }
   });
  }

//status - small message that shows up under the profile of the bot. customisable in
 config.js.
  //replace placeholders with specified content

var status = client.config.status.replace("{{guilds}}", client.guilds.size).replace(
"{{version}}", client.version);
  client.user.setPresence({ activity: { name: status }, status: "active" });
  //load the files for the events
  const eventFiles = fs.readdirSync("./events/");
```

```
  client.log(`Loading ${eventFiles.length} events from ${client.basedir}/events/`);
  eventFiles.forEach(eventFile => {
   //skip anything that's not a .js file
   if (!eventFile.endsWith(".js")) return;
   //strip the file extension
   const eventName = eventFile.split(".")[0];
   client.log(`attempting to load event ${eventName}`)
   //load it from disk
   const eventObj = require(`./events/${eventFile}`);

//bind it - on eventName, eventName(client, ...other args) as everything needs clien
t

//this means that the event will always be called with client as the first arguement
.
   client.on(eventName, eventObj.bind(null, client));
   //if the event has a default config, load it if it's not already present
   if (eventObj.defaultConfig) {
    client.guilds.forEach(guild => {

//if the value for the config exists, return the existing value, else write the give
n value.
     client.DB.ensure(guild.id, eventObj.defaultConfig, `modules.${eventName}`)
    })
   }
   //log that the loading was successful.
   client.log(`Bound ${eventName} to Client Sucessfully!`);
   //delete the file from the resolve cache as we no longer need it.
   delete require.cache[require.resolve(`./events/${eventFile}`)];
  });

//iterates over each guild that the bot has access to and ensures they are present i
n the database
  client.guilds.forEach(guild => {
   client.validateGuild(client, guild);
  });
  //cleans the cache from any lingering operations that may have been interrupted.
  fs.readdir("./cache/", (err, files) => {
   if (err) throw err;
   client.log(`Deleting ${files.length} files from cache...`)

//message to alert the operator/bot owner that a unexpected shutdown probably occurr
ed.
   if (files.length > 1) {
```

```
client.log("Cache Remnants detected - this should only occur if an unexpected shutdo
wn was invoked!", "WARN")
   }
   for (const file of files) {
    //delete all the remenant files
    fs.unlink(("./cache/" + file), err => {
     if (err) throw err;
    })
   }
  })
  //ends the initialisation timer as the bot is fully operational by this point.
  console.timeEnd("init");
  //

client.log(`Ready to serve in ${client.channels.size} channels on ${client.guilds.si
ze} servers, for ${client.users.size} users.`, "INFO");
  try {

//sends a "I'm alive!" message to the owner so they know when the bot is back online
.
   (client.users.get(client.config.ownerID))

.send(`Ready to serve in ${client.channels.size} channels on ${client.guilds.size} s
ervers, for ${client.users.size} users.`);
  } catch (err) {

client.log("I'm not in the guild with the owner - unable to send bootup notification
!", "INFO")
  }
 }
```

Input	Output	Expected	Bug
Client with no token	Login failed – error logged, init aborted	Login failed – error logged, init aborted	None
Client with no joined guilds	Error thrown and logged. Init aborted	Error thrown and logged. Init aborted	None
Client with user account token	Error thrown and logged. Init aborted	Error thrown and logged. Init aborted	None
Client with valid token	Initialisation routines operated as intended with no errors.	Initialisation routines operated as intended with no errors.	None
Client with valid token and preLoad	Error due to invalid fetchMessages	Initialisation routines operated as	Channel categories are classed as ‘channels’ – added check for

	method for a channel category	intended with no errors – messages loaded into cache.	channel type before attempting message fetch.
Client with valid token and preLoad	Initialisation routines operated as intended with no errors – messages loaded into cache.	Initialisation routines operated as intended with no errors – messages loaded into cache.	None

Table 25 - Prototype 23.1 Testing Table

Comments:

This section is a compilation of previous (already tested) prototype systems as well as some newer integrations. Testing indicates that these systems function as intended.

client.validateGuild

This novel function is the key function that validates a given guild's data in the persistent DB (client.DB). it also handles the loading of commands and guild-specific configurations for the bot, as well as the permissions required by the bot's enabled features in a guild.

Requirements:

Ensure that the guild's entry exists in client.DB, and follows client.defaultConfig as a soft schema.

Initialise the guild in transient data storages – e.g. trecent, cooldown.

Ensure every member is present in the database

Load every command, ensure config is present in the guild's DB entry.

Calculate the requiredPermissions the bot needs in the guild, check it has them. If not, notify relevant users.

Validate and remove any invalid persistence data – attachments and messages alike

Trigger the scheduler for the guild

Pseudocode:

Define function validateGuild (client,guild)

Set guildData to ensure guildID as defaultConfig in client.DB.guildID

Ensure guildID as {} in client.trecent

Ensire guildID as {} in client.cooldown

For every member in the guild:

Ensure users.memberID as {xp:0} in client.DB.guildID

Set reqPermissions to array of common permission flags

For every module in guildData.modules;

If the module has a requiredPermissions entry:

If the values of the requiredPermissions entry are not in reqPermissions;

Add them to reqPermissions

Log the requested permissions as DEBUG

Set missingPerms to filter (where the bot user in the guild does not have an element of reqPermissions)

If the bot has administrative permissions, return.

Else;

Log the missing permissions and send them to the guild's modActionChan, or the first channel in the guild.

For every attachment stored in client.DB.guildID.persistence.attachments;

If the attachments' timestamp is in the past, the attachment is no longer valid.

Remove attachment form attachments

For every message in client.DB.guildID.persistence.messages

If the bot cannot fetch the message from discord, the message is invalid

Delete the message.

Every 300 seconds, emit event "scheduler" ,guildID – this ensures the scheduler will run at least once every 300 seconds/guild.

Emit "scheduler", guildID

Relevant code:

```
/**validates a Guilds's configuration properties and database 'Presence'. called at startup and when a new guild is created.
  * @param {object} client
  * @param {object} guild
  */
 client.validateGuild = (client, guild) => {
  //ensures each server exists within the DB.
  var guildData = client.DB.ensure(guild.id, client.defaultConfig);
  //uses the defaultConfig as the basic framework if the guild really doesn't exist

//quick and easy way of finding roles with common names and assigning them automagically.
  guild.roles.forEach(role => {

client.log(`Testing role with name ${role.name} for Admin/Mod/Muted availability.`);

if (["Admin", "Administrator"].includes(role.name)) { client.DB.push(guild.id, role.id, "adminRoles"); }

if (["Mod", "Moderator"].includes(role.name)) { client.DB.push(guild.id, role.id, "modRoles"); }

if (["Muted", "Mute"].includes(role.name)) { client.DB.set(guild.id, role.id, "mutedRole"); }
  });
  client.DB.set(guild.id, {}, "commandsTable")

//ensures entry for the guild in trecent (used for antispam - stands for TalkedRecent, as in "Who has talked (sent messages) recently?")
  client.trecent.ensure(guild.id, {})
```

```
//ensures entry for the guild for command cooldowns - used to ratelimit command exec
ution.
  client.cooldown.ensure(guild.id, {})

//ensures each server has all it's users initialised correctly - each user is given
an object with just a property for XP (for now)
  guild.members.forEach(member => {
   //this object is extended/pruned as required by various submodules, eg demerits.
   client.DB.ensure(guild.id, { xp: 0 }, `users.${member.id}`);
  });
  //evict all those recently loaded keys from memory cache.
  client.DB.evict(client.DB.keyArray())
  //logs completion of validation.
  client.log(`Sucessfully Verified/initialised Guild ${guild.name} to DB`);
  //finds and automatically sets the ownerID for the server -> automatic admin
  client.DB.set(guild.id, guild.owner.user.id, "serverOwnerID");
  //load commands with default config into guild config (if it doesn't exist)
  const commandFiles = fs.readdirSync("./commands/");

client.log(`Loading ${commandFiles.length} events from ${client.basedir}/commands/`)
;
  commandFiles.forEach(command => {
   //ignore all non-.js files
   if (!command.endsWith(".js")) return;
   //strip file extention
   var command = command.split(".")[0];
   //run another function to fully load the command
   client.loadCommand(command, guild.id);
  });

//check module permission requirements to determine the permissions required by the
bot. starts with a basic set:
  let reqPermissions = ["SEND_MESSAGES", "READ_MESSAGES", "VIEW_CHANNEL"]
  //iterates over every module, checking the permissions declared as required.
  Object.keys(guildData.modules).forEach(key => {
   let Module = guildData.modules[key]
   if (!Module.requiredPermissions) return;
   Module.requiredPermissions.forEach(perm => {
    //append missing perms to the array.
    if (!reqPermissions.includes(perm)) { reqPermissions.push(perm) }
   })
  });
```

```
  //log requested permissisons

client.log(`Requested permissions for server ${guild.name} : ${reqPermissions.toStri
ng()}`)
  //find the permissions that the client is missing.

let missingPerms = reqPermissions.filter(perm => !(guild.members.get(client.user.id)
.permissions.toArray()).includes(perm))
  //override if the user has administrative permissions

if ((guild.members.get(client.user.id).permissions.toArray()).includes("ADMINISTRATO
R")) { missingPerms = [] }
  //if there are any missing perms...
  if (missingPerms.length >= 1) {
   //notify the bot owner as well as guild admins to the missing permissions.

client.log(`bot is missing permisions : ${missingPerms.toString()} in guild ${guild.
name}`, "ERROR")
   //send a message to the guild to request the missing permissions.

guild.channels.get(guildData.modActionChan) || guild.channels.filter(chan => chan.ty
pe === "text").values().next().value
    .send(`I am missing permissions : ${missingPerms.toString()}!`)
  }
  //check attachments, and remove those that have expired.
  let attachments = client.DB.get(guild.id, "persistence.attachments")
  client.log(`Verifying Persistence data for guild ${guild.name}`)
  //for every attachment...
  Object.keys(attachments).forEach(key => {
   //if the attachment has expired
   if (attachments[key].expiry < new Date()) {
    //delete the attachment
    client.DB.delete(guild.id, `persistence.attachments.${key}`)
   }
  })
  //check any persistent messages for expiary
  let messages = client.DB.get(guild.id, "persistence.messages")
  //for every message...
  Object.keys(messages).forEach(key => {
   //get the 'key' of the message - guildID:channelID:messageID
   let messageKey = messages[key].key

client.guilds.get(guild.id).channels.get(messageKey.split(":")[0].toString()).fetchM
essage(messageKey.split(":").toString()[1]).catch(err => {
```

```
//error only if the message is no longer reachable - eg deleted - therefore remove f
rom DB.
    client.DB.remove(guild.id, `persistence.messages.${key}`)
   })
  })
  //manually trigger the scheduler every 300 seconds to check this guild
  setInterval(() => { client.emit("scheduler", guild.id) }, 300000, client, guild)
  // start the scheduler for this guild
  client.emit("scheduler", guild.id)
 };

 };
```

Input	Output	Expected	Bug
guild – no pre-existing presence in database	Guild initialised properly within DB and other data structures	Guild initialised properly within DB and other data structures	None
Guild – existing malformed presence	Missing data entries amended – malformed data is not able to be detected or rectified	Missing data entries amended – malformed data is not able to be detected or rectified	None – this 'high level' integrity checking is intentional, as anything lower-level would rapidly increase complexity.
Guild – existing presence	No persistent records/entries were altered	No persistent records/entries were altered	None

Table 26 - Prototype 23.2 Testing Table

Comments:

Whilst developing and testing this function, Extra useful features were added. These include a subroutine to assign mod and admin roles within the bot based on some common role names, a wiping of the commandsTable to prevent aliases from being persistent (which could potentially cause problems if commands are added/removed between reboots) and the removal of all the loaded keys (as they are loaded due to the ensure operations) from the in-memory cache.

Client.getRandomInt

Very basic function - simply gets a random integer between a min and max value. Only used for basic xp implementation for now.

Define function getRandomInt (min,max)

Set min to math.ceil(min)

Set max to math.floor(max)

Return math.floor(math.random * (max-min)) +min

Relevant code:

./functions.js

```
/**
  * returns a random integer between two numbers (max exclusive, min inclusive.)
  * @param {int} minimum
  * @param {int} maximum
  */
 client.getRandomInt = (min, max) => {
  min = Math.ceil(min);
  max = Math.floor(max);
  //The maximum is exclusive and the minimum is inclusive
  return Math.floor(Math.random() * (max - min)) + min;
 };
```

Input	Output	Expected	Bug
Min: 0 max: 0	0	0	None
Min: -1 max: 100	51	Something in range	None

Table 27 - Prototype 23.3 Testing Table

Comments:

Due to the relative low level of utilisation and thus criticality of this function, exhaustive testing will not be conducted – also this is a very common tried and tested algorithm.

The Completed functions.js code can be seen in the later complete code overview section.

3.24 PROTOTYPE 24 – DEVELOP EVENT STUBS

Goals: integrate event ‘stubs’ into the existing infrastructure.

So far, most events have been left as barely functional ‘stubs’ – this was an intentional choice as developing infrastructure for processing the data from these events would be redundant – due to developmental priorities, some of these will remain as stubs for the foreseeable future.

The current list of events are:

attachmentRecorder – finished

AutoMod – finished

guildCreate – unfinished

guildMemberAdd – unfinished

guildMemberRemove – unfinished

guildMemberUpdate – stub

guildUpdate – stub

message – finished

messageDelete – unfinished

messageReactionAdd – stub

messageUpdate – unfinished

modActions – finished

NSFWClassifier – finished

Scheduler – finished

toxicClassifier – finished

that leaves 5 events to finish: guild Create/member Add/Remove and message Update/Delete

guildCreate:

guildCreate is emitted whenever a guild is 'created' – or for our purposes when the bot joins a new guild.

This event simply needs to invoke the previously implemented client.validateGuild routine to get the guild up-and-running as far as the bot is concerned.

Relevant code:

./functions.js

```
/**
 * Triggered every time a guild is joined by the bot.
 * @param {object} client
 * @param {object} guild - the guildObject for the guild that has just been joined.
 */
module.exports = (client, guild) => {
 //log that a guild has been joined.
 client.log(`Client has joined guild ${guild.name}!`, "INFO");

//invokes a check for the  bot's DB to initialise the guild in the database for inst
ant usage.
 client.validateGuild(guild)
 };
```

Input	Output	Expected	Bug
Join a new guild	Guild fully initialised	Guild fully initialised	None
Join previously joined guild	Guild database records validated by validateGuild – old config kept and reused	Guild database records validated by validateGuild – old config kept and reused	None – this behaviour is intentional.

Table 28 - Prototype 24.1 Testing Table

Comments:

As this almost entirely relies on the previously tested client.validateGuild, exhaustive testing is unnecessary as testing has already taken place for this function.

guildMemberAdd/Remove

This event is triggered whenever a user joins (or re-joins) a guild. Initially all this function had to do was ensure the user into the guild's DB entry, but It was decided to implement a similar set of behaviour to guildCreate, wherein the previous 'state' is restored – for a member, this primarily means restoring their roles and nickname. It was decided to implement the storing and retrieval of this state through guildMemberRemove and

guildMemberAdd respectively. As such, It was decided to develop both features side-by-side. guildMemberRemove is, as the name suggests, invoked whenever a member leaves a guild for any reason – kick, ban, or voluntary.

Objectives:

Add: ensure user in DB, restore state if configured. Log.

Remove: store user state if configured. Log.

Pseudocode: Add

Set modConfig to modules.guildMemberAdd in client.DB.guildID

Log the member and server

Set data to ensure users.memberID as {xp:0} in client.DB.guildID

If data has property savedState;

Set state to data.savedSate

Set the user's nickname to sate.nick

Set the user's roles to sate.roles

Create an action to emit modActions to create a notification of restoration

If the module is enabled,

Send the welcome message with placeholders replaced to the welcome message channel.

Relevant code:

./events/guildMemberAdd.js

```
/**
 * triggered when a user joins (or re-joins) a guild. in the case of a rejoin,
 * the bot will reload a saved user state (roles and nickname) before they left the guild.
 * @param {object} client
 * @param {object} member - member object for the user that just joined the guild.
 */
module.exports = async (client, member) => {
 //gets config
 let modConfig = client.DB.get(member.guild.id, `modules.guildMemberAdd`)
 //alias for guild
 let guild = member.guild;
 //logs memeber join

client.log(`new member with Name ${member.displayName} has joined ${guild.name}`, "INFO");
 //initialisies user to DB
 let data = client.DB.ensure(guild.id, { xp: 0 }, `users.${member.id}`);
 //if a backup of a user state exists, restore the user.
 if (data.savedState) {
  //get state
  let state = data.savedState
```

```
  //set nickname to nickname in state data
  member.setNickname(state.nick)
  //set roles to array of role ID's in state data.
  member.addRoles(state.roles)
  //action object - for notification of action to moderators/admins.
  client.emit("modActions", {
   memberID: member.id,
   guildID: member.guild.id,
   type: "report",
   title: "State restore of re-joining User",
   executor: client.user,
   request: `User ${member} has re-
joined the server, and has had their roles and nickname restored.`,
   trigger: {
    type: "automatic",
   }
  })
 }
 //if the welcome message is enabled
 if (modConfig.enabled) {
  //send it (with some parsing) to the specified Welcome channel

client.guilds.get(guild.id).channels.get(modConfig.welcomeChannel).send((modConfig.w
elcomeMessage)
   .replace("{{user}}", member).replace("{{guildName}}", guild.name))
 }
};
module.exports.defaultConfig = {
 enabled: false,
 welcomeChannel: 0,
 welcomeMessage: "welcome {{user}} to {{guildName}}!"
}
```

Pseudocode: Remove

Log that member has left guild with flag INFO

If this module is enabled;

Store the roles and nickname of the user under users.userID.savedState in client.DB.guildID

Relevant code:

./events/guildMemberRemove.js

```
/**
 * Triggered when a user leaves a guild. the bot will copy their state from just bef
ore they left and save it
```

```
 * (roles and nicknames)
 * @param {object} client
 * @param {object} member - object of the member that just left the guild.
 */
module.exports = (client, member) => {
 //log the fact that a member has left the guild

client.log(`member ${member.displayname} has left guild ${member.guild.name}`, "INFO
")
 //if the module is disabled, do nothing.

if (!client.DB.get(member.guild.id, `modules.guildMemberRemove`).enabled) { return }
 //create state
 let state = {
  nick: member.displayName,
  roles: Array.from(member.roles.keys()),
  TS: new Date()
 }
 //save state
 client.DB.set(member.guild.id, state, `users.${member.id}.savedState`)
}
module.exports.defaultConfig = {
 enabled: true
}
```

Input	Output	Expected	Bug
New member joining guild	Logged, welcomed	Logged, welcomed	None
Member re-joining guild – savedSate present	Logged, state restored, welcomed	Logged, state restored, welcomed	None
Member re-joining guild – savedSate not present	Logged, no state restore, welcomed	Logged, no state restore, welcomed	None
Member leaving guild – savedState disabled	Logged, no savedState stored	Logged, no savedState stored	None
Member leaving guild – savedState enabled	Logged, savedState stored	Logged, savedState stored	None

Table 29 - Prototype 24.2 Testing Table

Comments:

This feature was surprisingly easy to implement due to the flexible nature of the discord.JS API wrapper for adding/removing roles. when developing guildMemberAdd, altering its behaviour was considerd to make it so that a re-joining user is not welcomed like a new user would be. After testing with my client group, we decided to keep this functionality for consistency reasons, and instead implement an action that would, via modActions, alert the moderators of the reset of a re-joining user. This is so that if they want/need to, they can use the embed member reference in the embed (see below) to alter the effects of the restoration.

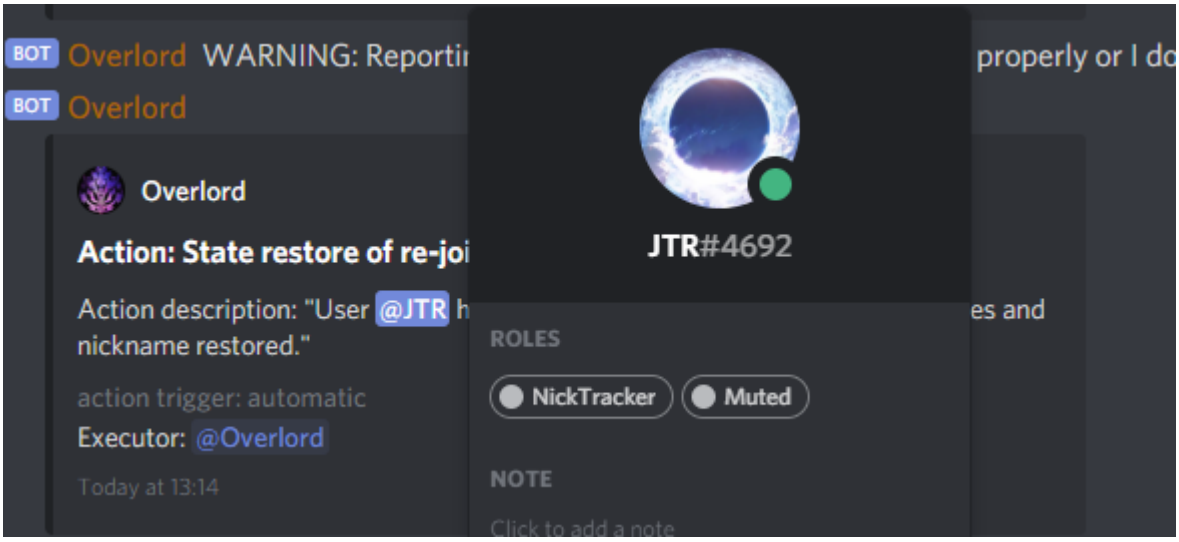

Figure 27 - Embed with Member Reference

Embed with included member reference, through which the state of a member can be directly viewed and altered.

Compiled Design and Process overviews

Message Update/Delete – these events are triggered whenever a message that is in the bot's cache is deleted/edited (delete/update respectively). Each event needs to report differing data – delete needs the original contents as well as any attachments, whilst update requires showing the new and the old contents of a message. Both of these differences have already been programmed into modActions within a previous prototype as I knew I would need to implement this behaviour at some point.

messageUpdate

pseudocode:

if the module is enabled in the current guild;

emit "modActions" via client with action object containing:

the title, type, change, data, edit (novel property), attachments, executor, guildID and messageID.

Relevant code:

```
/**
 * triggered when a message is updated
this (annoyingly) includes the addition of automatic embeds.
 * include other useful changes such as when the content of a message changes.
 * automatically sends data to modActions for reporting.
 * @param {object} client
 * @param {object} Message - original message object
 * @param {object} nMessage - New Message object (changed/updated)
 */
module.exports = (client, Message, nMessage) => {
 if (!Message.guild) return //check that the message is in a guild
 if (!client.DB.get(Message.guild.id).modules.messageUpdate.enabled) return
 //check if reporting edits is enabled.
 nMessage.content = nMessage.cleanContent;
```

```
 //clean the content
 if (Message.content === nMessage.content) {
  client.log("messageEdit invoked - message content identical - assume autoembed");
  //if contents are identical, autoembed was most likely triggered.
  //autoembeds are created by Discord to show media content in a message.
  return
 } else {
  let action = {//prepare action object
   title: `Message by ${Message.author} edited in ${Message.channel}.`,
   type: "audit",
   change: "edited",
   data: Message.content,
   edit: nMessage.content,
   executor: Message.author,
   guildID: Message.guild.id,
   memberID: Message.author.id
  }
  client.emit("modActions", action)
 }
}
module.exports.defaultConfig = {
 enabled: true,
 requiredPermissions: ["MANAGE_MESSAGES"]
}
}
```

Input	Output	Expected	Bug
Message sent with no edits	No processing	No processing	None
Message with media sent with no edits	Triggered as if the message had been edited	No processing	Discord adds automatic embeds for media after they have been posted- this triggers a messageUpdate event. Added check to ensure message content has changed.
Message with media sent with no edits	No processing	No processing	None
Message edited outside of guild	Processing failed – no message.guild property	No Processing	Does not check that the message is sent in a guild – add check to ensure it only pays attention to messages edited within guilds
Message edited outside of guild	No Processing	No Processing	None

Message edited within guild	Embed showing edit sent to defined audit channel	Embed showing edit sent to defined audit channel	None

Table 30 - Prototype 24.3 Testing Table

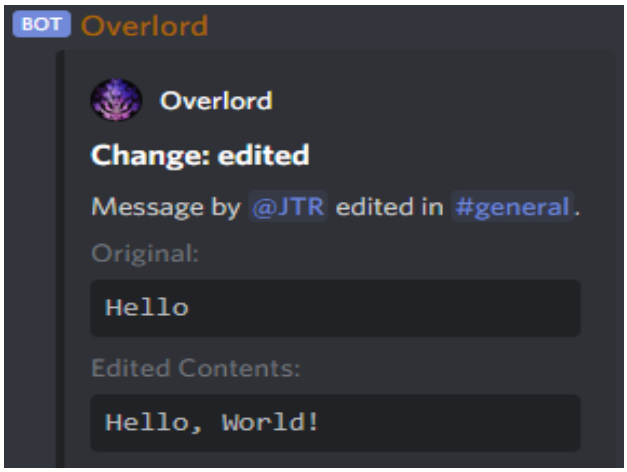

Figure 28 - Example Message Edit Audit Embed

Comments:

Bug testing resulted in some minor alterations to this event handler, such as the check for the message being in a guild – which normally is not an issue for most routines but as this one does not go through the message event, it needed to be implemented as pretty much then entire bot is reliant on guild-only functionality. Another alteration was a simple content comparator to check that the message content (the part we're interested in) has changed rather than anything else – this is primarily to combat automatically added media embeds triggering events.

messageDelete:

The purpose of messageDelete, much like messageUpdate, is to provide transparency to moderators about what was sent, as there is no way to restrict message editing or deletion, nor any way of logging the original contents in any great detail.

Pseudocode:

If the module is enabled in this guild;

Set action to new action object with properties:

Title, type, change, data, attachment,executor,guildID

If the message had attachments:

Fetch them from the database and set action.attachments to attachments.

Emit modActions (action)

Relevant code:

./events/messageDelete.js

```
/**
 * triggered whenever a message still in the bot's cache is deleted by a user. logs the deleted message.
 * @param {object} client
 * @param {object} message
```

```
 */
module.exports = async (client, message) => {
 //check that the author of the deleted message wasn't a bot.
 if (message.author.bot || !message.guild) return;
 //if disabled, stop execution.
 if (!client.DB.get(message.guild.id).modules.messageDelete.enabled) return
 let entry = await message.guild.fetchAuditLogs({ type: "MESSAGE_DELETE" })
  .then(audit => audit.entries.first());
 //get the audit log entries and find the latest MESSAGE_DELETE entry.
 //- executor is the person who deleted the message in this case
 let executor = ""; //eslint-disable-line
 if (entry != undefined
  && (entry.extra.channel.id === message.channel.id)
  && (entry.target.id === message.author.id)
  && (entry.createdTimestamp > (Date.now() - 15000))
  && (entry.extra.count >= 1)) {
  //if this passes, then the person who deleted the message is the executor.
  //check that the executor is not a bot
  if (executor.bot) { return }
  //notification of discrepancy in author and executor.
  executor = `${entry.executor} - Original Author is ${message.author}`
 } else {
  //if not, we have to assume it was the author of the message.
  executor = message.author
 }
 //action object generated for processing.
 let action = {
  title: `Message deleted in ${message.channel} by ${executor}.`,
  type: "audit",
  change: "deleted",
  data: message.content,
  attachments: [],
  executor: "",
  guildID: message.guild.id,
 }
 if (message.attachments) {
  try {
   //get the saved attachments (if they exist) from the DB.
   action.attachments = (
    client.DB.get(message.guild.id, `persistence.attachments.${message.id}`)
     .attachments || [])
```

```
  } catch (err) {
   client.log("message had no attachments")
  }
 }
 //pass the action object to modActions for processing.
 client.emit("modActions", action)
};

module.exports.defaultConfig = {
 enabled: true,
 requiredPermissions: ["VIEW_AUDIT_LOG"],
}
```

Input	Output	Expected	Bug
Message deleted in non-guild channel	No Processing	No Processing	none
Message deleted by a bot	Processed as if a user had deleted the message	No Processing – bots are 'trusted'	Lack of check for executor, added check to ensure executor was not a bot.
Message deleted by a bot	No Processing	No Processing	None
Message by a bot deleted	Processed as if it were a regular user message	No Processing – bots oftentimes delete their own messages or people delete their messages	Lack of check for author bot status – added a check to ensure author was not a bot
Message by a bot deleted	No Processing	No Processing	None
Message by a user deleted by the user	Processed correctly – executor set, and audit embed sent	Processed correctly – executor set, and audit embed sent	None
Message by a user with media deleted by the user	Processed correctly – executor set, and audit embed sent, including the attachments	Processed correctly – executor set, and audit embed sent, including the attachments	None

Table 31 - Prototype 24.4 Testing Table

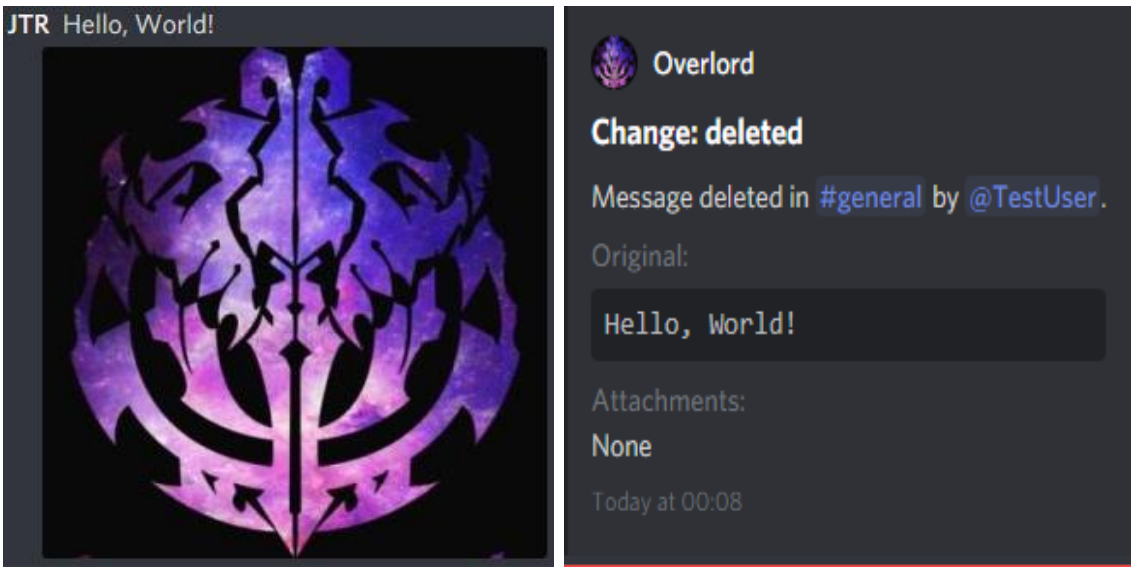

Figure 29 - Message and Audit Embed

Figure 29 shows an example message (with media – left) and audit embed with URL (right).

This event was one of the more complex to implement due to the need to determine the 'executor' - that is, who deleted the message (as users who have the right perms can delete other user's messages). This can be done by exploiting the fact that Discord's built-in audit system logs message delete operations that are done by users on messages that are not authored by them – although this is rather simple in terms of useful information (see below).

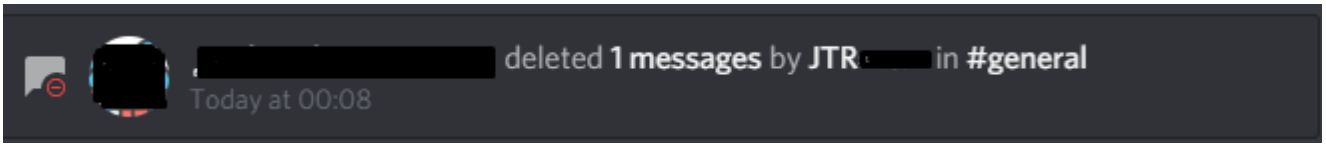

Figure 30 - Audit Log Entry

(In-app visual representation of an audit log entry – Discord usernames and tags removed due to privacy concerns.)

This system means Overlord can use a function to get and process the audit logs to attempt to find one that seems to fits the deletion of the message – if one doesn't exist, we have to assume the executor is the author. The algorithm used for this is not made by me, but to summarise it's functionality, it checks the context and content of the audit log entry to see if it is applicable to the current 'messageDelete' event.

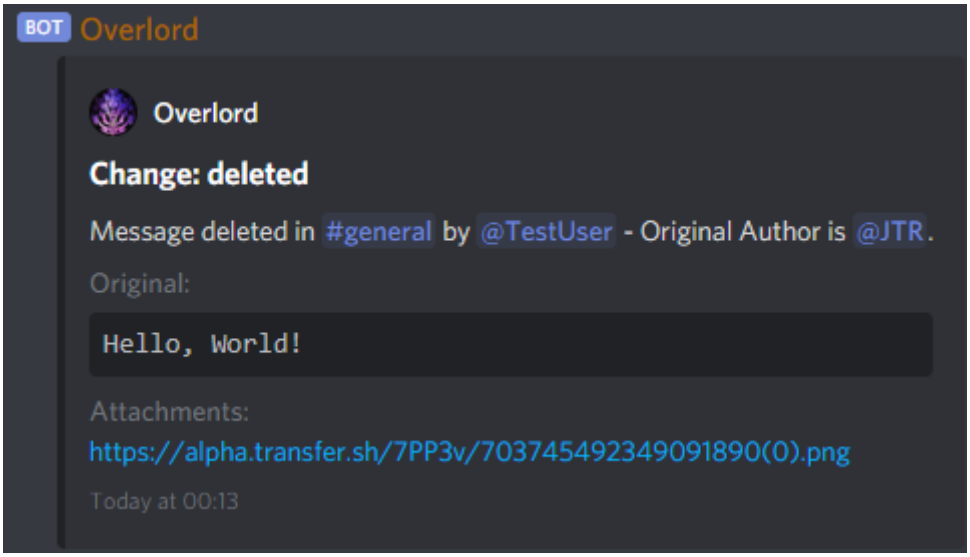

Figure 31 - Deleted Message Audit Embed

< - embed containing altered title as the bot has detected a suitable audit log entry for the deletion of the message.

Comments:

There is one final issue with these events – this event (and others that rely on messages) are only triggered for messages that the bot has within its internal message cache (at least for the Discord.js version Overlord is using). This is workable for other events due to the inclusion of the raw event, that will show any and all changes, and as such a bridge for events can be devised that loads the message into cache. However in the case of a message deletion, the client is unable to retrieve the message as it has already been deleted. Implementation of such a bridge was not initially intended, but as it would massively expand the functional time scope of the bot (as it removes messages from cache after a set time to reduce memory footprint), meaning few/no messages are 'ignored' due to not being in the cache. Implementing this will be the next prototype. At least it would be if initial testing showed usefulness for any of the affected events (mainly 'messageUpdate', 'messageDelete').

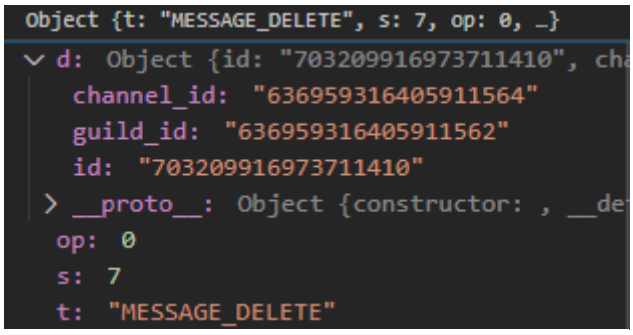

Figure 32 - Raw Event Property Diagram

Figure 32 shows a debug snippet with lacklustre contents of the raw 'messageDelete' packet. All attempts to fetch the message referenced as d.id failed. With 'messageUpdate', the message is resolvable, but does not contain the original contents, just the edited one, making it useless.

This Unfortunately means that this concepts in non-viable, as having to rely on messages within the cache to receive updates about these messages is the reason for the rather aggressive focus on keeping a low bot memory footprint –but so that the message cache has as much space as it requires, as well as performance. Although the implementation of a lower-level cache management system for the bot has not yet been fully implemented (as it is not a high-priority for development), the issue of caching messages remains.

3.25 PROTOTYPE 25 – PM2 INTEGRATION

Goals: implement PM2 integration fully within the bot

Whilst PM2 integration has been a key point of note for Overlord as a scalable system, a proper integration with it has not yet occurred. The main point of integration is the construction of an ecosystem.config file, which specifies the environment that Overlord should be run in via PM2. The contents of this file are as follows:

./ecosystem.config.js

```
module.exports = {
 // Options reference: https://pm2.io/doc/en/runtime/reference/ecosystem-
file/ - very helpful resource!
 apps: [{
  name: "Overlord",
  script: "./Overlord.js",
  instances: 1,
  autorestart: true,
  watch: false,
  env: {
   NODE_ENV: "production"
  },
  max_memory_restart: "2G",
  combine_logs: true,
  log: ".\\main.log",
  log_date_format: "YYYY-MM-DD HH:mm:ss"
 }],
};
```

Comments:

This file specifies the application name and start path, the number of configured instances to run, whether to auto restart the application upon it exiting/crashing, whether to ‘watch’ files used by the app for changes, and upon any changes, restart the bot. It also specifies the node environment flag “production” which tells the bot to use the configured production token over the development token. Following this, it specifies the maximum memory the application can utilise, above which it will be restarted automatically via usage of a process message or a SIGINT (see ./Overlord.js for the integration for this). Normally, PM2 has two logs, an error log, and a ‘regular’ log. For Overlord, It was decided to merge these because most errors are caught and logged using my logging system, which means some errors do not end up in the correct log file. This also means that I can get the contextual information about the error from the log, rather than having to just go off an error message stored in the error log.

3.26 PROTOTYPE 26 – FINALISE HELP COMMAND

Goals: finalise the help command

So far, many 'stub' commands for command reloading, rebooting the bot, and evaluating code from the bot owner have been completed. However, Overlord is still lacking one crucial command – the help command.

This command, as the name suggests, provides help to an otherwise confused user about what commands the bot has, and how to use them – this command is directly referenced by the bot in the message it sends a user after being directly mentioned as a way of providing assistance. This is one of the areas of the bot that needs to very explicitly tell the user how it works.

Requirements:

This command needs to do the following:

Upon invocation, the bot needs to provide the user with a summary of all the commands they can execute, as well as their aliases and usage information. The user can then specify a command to get more detailed information on, such as usage. Ideally, only commands that the user can use are shown by this command.

Pseudocode:

Set cembed to a new embed with some generic attributes.

Set commandList to empty array

If no argument is specified;

For every command in message.settings.commands;

If client.canExecute(command);

Push cmd to commandList

Add field to cembed : data commandList.join(\n) – to create newlines

Send the embed to the user via DMs

If an argument has been specified:

If client.canExecute (command);

Add a field to cembed with cmd

Send embed to the user via DMs

Relevant code:

./commands/help.js

```
/**
 * small command to list the commands the user can execute, as well as any configure
d aliases, with the option to return more information once the command name is speci
fied.
 * @param {object} client
 * @param {object}
 */
exports.run = (client, message, args) => {
 const Discord = require("discord.js");
 const cembed = new Discord.RichEmbed() //creates a new embed instance
  .setTitle(`${client.user.username}'s List of commands`)

.setAuthor(`${client.user.username}`, `${client.user.displayAvatarURL}`) //adding el
ements
```

```
  .setColor("#1a1aff") //this is a dark blue-ish colour
 let clist = [] //list of commands in array format
 if (!args[1]) {
  Object.entries(message.settings.commands).forEach(cmd => {
   if (client.canExecute(client, message, cmd[0]) === "passed") {

//check if the user can run the command - no point telling them if they can't use it
!

clist.push(`\`\`\`\n${cmd[0]}: aliases: ${cmd[1].aliases.filter(alias => alias != cm
d[0]).join(", ") || "none"} ${cmd[1].info ? "Info: " + cmd[1].info : ""}\n\`\`\``)
   }
  })
  cembed.addField("List of Available Commands:", clist)

.setFooter("Use the help command followed by the command you want more information o
n.") //prompt the user as to additional functionality.
  message.author.send({ embed: cembed });


 } else {

if (client.canExecute(client, message, args[1]) === "passed") { //another check as t
his is a different usecase
   let cmd = message.settings.commands[args[1]]

cembed.setTitle(args[1]) //shows usage information as well as any info, both are opt
ional.

.setDescription(`\`\`\`\naliases: ${cmd.aliases}\ninfo: ${cmd.info || "No info"}\nus
age: ${cmd.usage.replace("$", message.settings.prefix) || "no usage information"}\n\
`\`\``)
   message.author.send({ embed: cembed });
   return
  }
 }
}
exports.defaultConfig = {
 aliases: ["help", "commands"],
 info: "lists the commands the user is able to execute in the current server",
 usage: "$help <command name/alias>",
 enabled: true,
 permReq: [],
 cooldown: 10000,
 allowedChannels: [],
```

```
};
```

Input	Output	Expected	Bug
help command – no arguments	User sent embed containing commands that they can execute, as well as any aliases and info.	User sent embed containing commands that they can execute, as well as any aliases and info.	None
Help command – args	User sent embed containing more information about the specified command (if they can execute it)	User sent embed containing more information about the specified command (if they can execute it)	None

Table 32 - Prototype 26 Testing Table

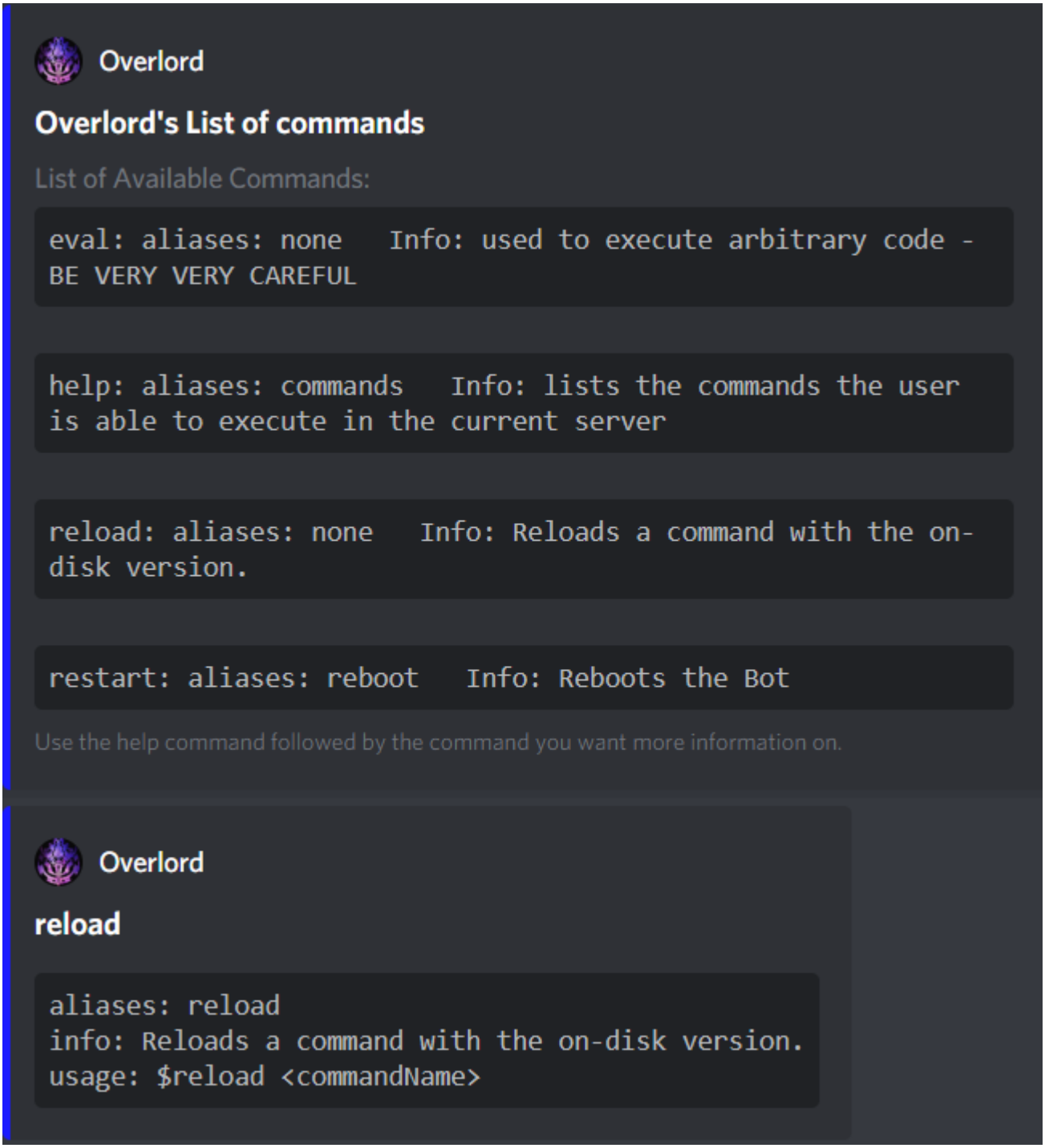

Figure 33 - Example Help Embed

Embeds from a general help command as well as a specific help command, showing increased detail on a command.

As all data about usage and info is defined in the configuration, it can be configured per-guild – these are the defaults but can easily be changed.

With these prototypes assembled and tested, the ‘Core’ of Overlord has been fully established. Apart from some refinements from full system testing, the code from these prototypes is essentially complete.

3.27 COMPLETED SYSTEM TESTING

With the main core of Overlord now complete, testing of the system as a whole can begin. This testing will, like the prototype testing, be grey box testing. Unlike the prototype testing however, it will be more focused on a black box approach, testing overall functionality from the perspective of an end user rather than a developer. This testing will test a variety of inputs the system may encounter, and for the final test will be used live on a Discord guild.

Test No.	Objective	Test Data	Expected Result	Actual Result
1	Test bot initialisation routines with a blank database	Valid application token, starting bot via PM2 command.	Bot authenticates and created database entries for guilds	Successful authentication, all guilds initialised (see figs. 34.x)
2	Test message event	Message sent in a guild channel	Bot receives message event, passes data to message.js then Heuristics	Message.js triggered, heuristics triggered.
3	Test Command Processing	Message from user: $help	User is DM'd by Overlord with the help embed	Help embed DM'd to user (fig 35)
4	Test Command help	Message from user: $help reload	User DM'd with embed containing info on the reload command	User DM'd with embed containing info on the reload command
5	Test Command reload	Message: $reload help	Help reloaded with change	Help reloaded with change
6	Test Command limiter - permissions	Message: $eval client.DB	Command Failure due to lacking perms	Command Failure due to lacking perms
7	Test Command limiter - cooldown	Message : $help x3	First command executes – subsequent fail.	First command executes – subsequent fail.
8	Test Command Limiter - Non-existent	Message: $ThisDoesntExist	Execution fails: unknown command	Execution fails: unknown command
9	Test command limiter - disabled	Message: $reload help	Execution fails – command disabled	Execution fails – command disabled
10	Test command limiter - channel	Message: $reload help	Execution fails – current channel not in whitelist	Execution fails – current channel not in whitelist
11	Test command limiter - channel	Message: $reload help, sent in whitelisted channel	Command executes as normal	Command executes as normal
12	Test command limiter - blacklist	Message: $reload help	Execution fails – user is blacklisted	Execution fails – user is blacklisted
13	Test Heuristics - Toxicity classifier	Message from member of test group containing toxic content	Content correctly classified and flagged. Action to remove message sent	Content correctly classified and flagged. Action to remove message sent.
14	Test heuristics – Toxicity classifier	Message containing no toxic content: "Hello, world!"	Content correctly classified as non-toxic	Content correctly classified as non-toxic
15	Test heuristics – spam	"Hello" x5 in 3 seconds	Detected as spam – action created	Detected as spam – action created
16	Test heuristics – spam	"Hello" x5, split by 1s intervals	Not detected as spam – no action	Not detected as spam – no action

17	Test heuristics - charFlood	Message: “aaaaaaaaaa!”	Not classified due to below threshold length	Message not classified as charFlood.
18	Test heuristics - charFlood	Message: “a” x 24	Classified as charFlood	Classified as charFlood, action to remove created.
19	Test heuristics - charFlood	Message “a” x 11, “b” x 13	Classified as charFlood, one char greater than x% of message	Classfied as charFlood, action to remove created.
20	Test heuristics - caps	Message: Alphabet but all in uppercase (caps)	Classified as caps flood due to exceeding x% caps limit	Classified as caps flood – action to remove message created
21	Test heuristics – excluded roles	Message: Alphabet but uppercase, user has excluded role	Not classified due to user having excluded role	Not classified/processed due to user having excluded role
22	Test Module - AttachmentRecorder	Message: “Hello!”	No processing due to lack of media	No processing due to lack of media content
23	Test Module - AttachmentRecorder	Message with attached media	Media processed according to config	Media processed according to config – download then upload to transfer.sh, URL stored in DB
24	Test Module - AttachmentRecorder	Message with media URL	Media processed according to config	Media processed according to config – download then upload to transfer.sh, URL stored in DB
25	Test Module - AttachmentRecorder	Message with multiple media URLs/attachments	Media processed according to config	Media processed according to config – downloaded then uploaded to transfer.sh, URLs stored in DB as an array.
26	Test heuristics - NSFWClassifier	Message from member of test group containing NSFW content	Media downloaded and classified as NSFW	Media downloaded, classified as NSFW, action for removal created
27	Test heuristics - NSFWClassifier	Message from member of test group containing SFW media	Media downloaded and classified as SFW	Media downloaded and classified as SFW. No action created.
28	Testing moderation - action	Message spam (5 messages in 3 second)	Detected as spam – action embed to remove spammed messages. Reacting to embed removes messages.	Detected as spam – action embed to remove spammed messages. Reacting to embed removes messages.
29	Testing moderation – action	Message spam – changed requested action to mute	Detected as spam, action embed requesting mute created. Upon reacting, user muted.	Detected as spam, action embed requesting mute created. Upon reacting, user muted.
30	Testing moderation – action	Message spam – changed requested action to ban	Detected as spam, action embed requesting mute created. Upon reacting, user banned.	Detected as spam, action embed requesting mute created. Upon reacting, user banned.

31	Testing moderation – Punishment	Message spam – adds 1 demerit	Detected as spam, demerits added to the user. No penalty.	Detected as spam, demerits added to the user. No penalty.
32	Testing moderation – Punishment	Message spam – adds 5 demerits	Detected as spam, demerits added to the user. User penalised with a mute. User notified via DMs	Detected as spam, demerits added to the user. User penalised with a mute. User notified via DMs
33	Testing moderation – Punishment	Message spam – adds 10 demerits	Detected as spam, demerits added to the user. User penalised with a mute and a temporary ban. User notified via DMs	Detected as spam, demerits added to the user. User penalised with a mute and a temporary ban. User notified via DMs
34	Testing moderation – Punishment	Message spam – adds 15 demerits	Detected as spam, demerits added to the user. User penalised with a permanent ban. User notified via DMs	Detected as spam, demerits added to the user. User penalised with a permanent ban. User notified via DMs
35	Testing moderation – Punishment	Data from test 34, after a few decay cycles	User's demerits decay, punishments reversed (except for the Permanent ban)	User's demerits decay, punishments reversed – user gets unbanned then unmuted – including the permanent ban. Added a fix for this by changing the permBan end threshold to -1, an impossible value.
36	Testing moderation – Punishment	Data from test 34, after a few decay cycles	User's demerits decay, punishments reversed (except for the Permanent ban)	User's demerits decay, punishments reversed (except for the Permanent ban)
37	Testing audit – message deletion	Message sent: "Hello, world!" and then deleted	Message delete event sends audit log embed to channel	Message delete event sends audit log embed to channel
38	Testing audit – message deletion	Message sent: "Hello, world!" with attachment and then deleted	Message delete event sends audit log embed to channel. Attachment recorder stores URL which is appended to the embed.	Message delete event audit embed sent to audit log channel – URLs for the deleted contents attachments pulled from database.
39	Testing audit – message edit (update)	Message sent: "Hello, World!" and then edited.	Message Update event triggered, audit embed sent	Message Update event triggered, audit embed sent
40	Testing events - guildMemberAdd	Member added to guild	Member initialised into DB, welcomed via welcome message.	Member initialised into DB, welcomed via welcome message.

41	Testing events - guildMemberAdd	Member re-joins guild	Member welcomed, and if they had a saved state, their nickname and roles are restored.	Member welcomed, and if they had a saved state, their nickname and roles are restored.
42	Testing events - guildMemberRemove	Member leaves the guild	Member state saved to their database entry	Member state saved to their database entry
43	Test command - restart	Message: $restart	Bot gracefully shuts down, preventing all interaction until it restarts	Bot gracefully shuts down, preventing all interaction until it restarts
44	Test command - eval	Message: $eval client	Evaluates statement -Prints client object	Evaluates statement -Prints client object
45	Testing scheduler	Scheduler object with type reminder	At the specified timestamp, scheduler processed action, sending reminder to user.	At the specified timestamp, scheduler processed action, sending reminder to user.

This testing allowed me to test various features of the bot whilst they were interacting with each other, something that was not extensively tested within each prototype. This testing primarily focused just on the systems that were co-dependent or critical in nature, such as the message event, the moderation subsystem and all the other systems that use it – primary other modules that perform heuristics on UGC, eg NSFW/Toxic Classifier. This testing showed very few bugs – a fact that I'm surprised with given the lack of testing for integration between systems in prototype tests. With this testing complete, Overlord's codebase has been all but solidified for a production run.

Test table figures:

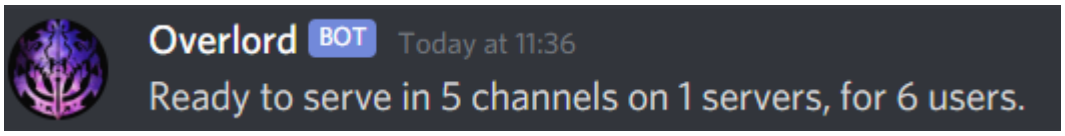

Figure 34 - Test 1 - Initialisation Message

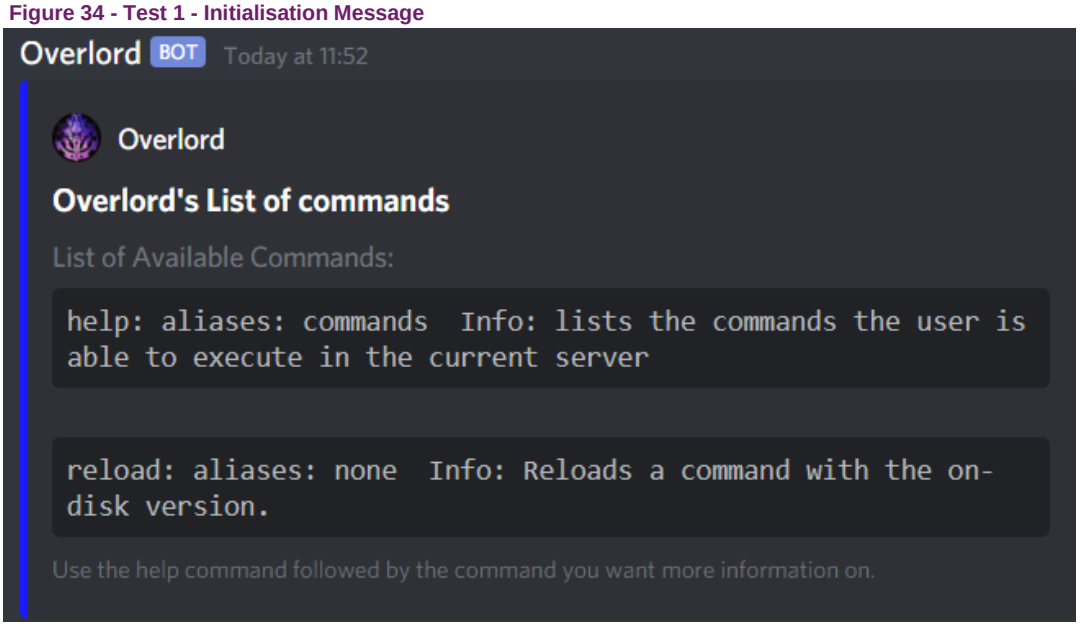

Figure 35 - Test 3 - Help Embed

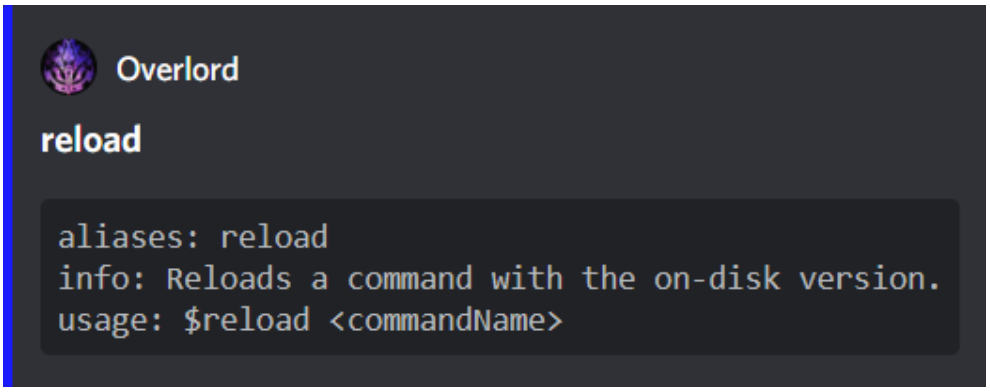

Figure 36 - Test 4 - Reload Help Embed

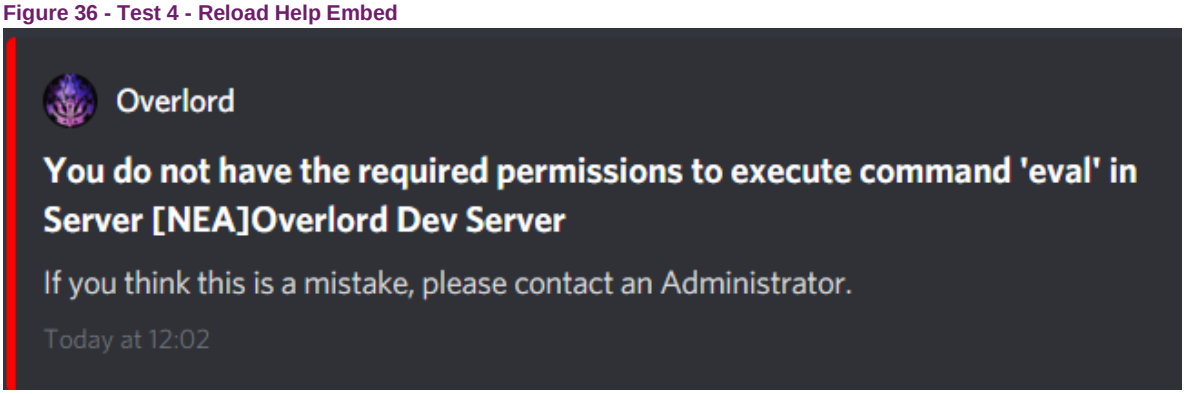

Figure 37 - Test 6 - Eval Error Embed

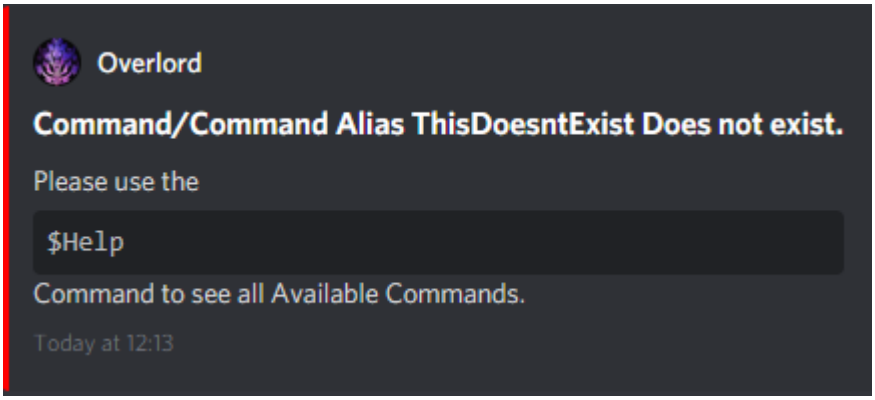

Figure 38 - Test 8 - Non-Existant Error Embed

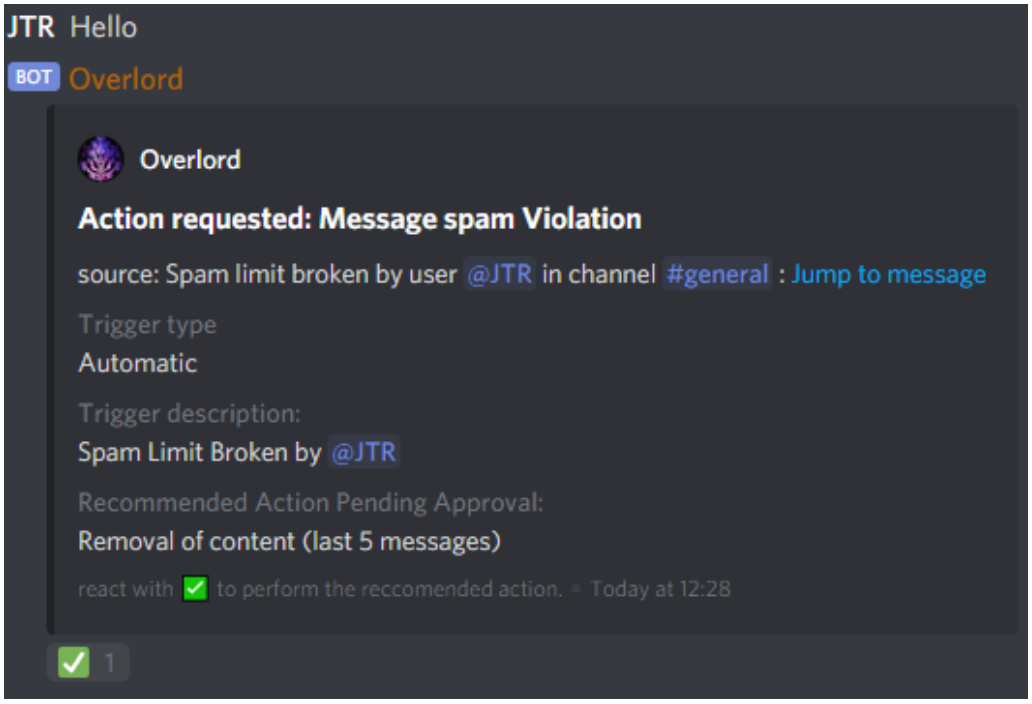

Figure 39 - Test 15 - Spam Removal Action Embed

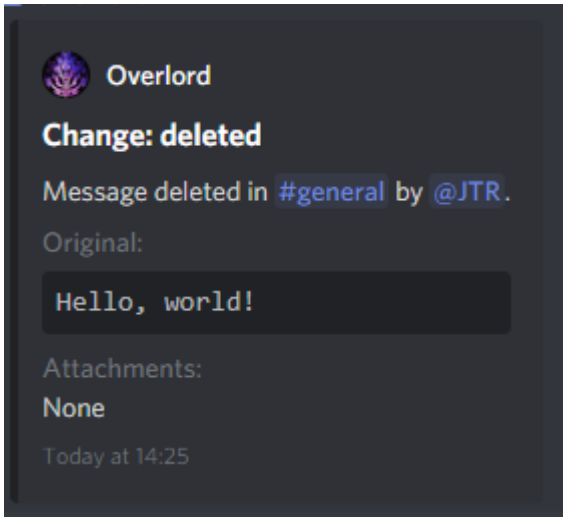

Figure 40 - Test 37 - Message Deletion Audit Embed

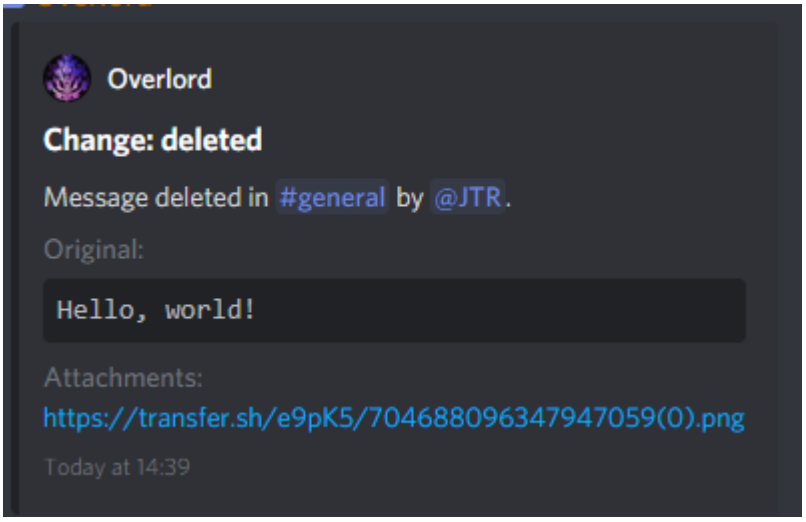

Figure 41 - Test 38 - Message Deletion with Attachment Audit Embed

Figure 42 - Test 39 - Message Update Audit Embed

3.28 FEEDBACK FROM A LIVE RUN ON AN ACTIVE DISCORD GUILD

This feedback is from a member of the admin client group, who hosted the current version of Overlord on their server, and what feedback they had after doing so for a few days:

"Although it [Overlord] took a bit of work to set up, once it was up and running it ran fine – it did crash once due to my internet cutting out for a bit, but I didn't even notice until the logs were being read later.

As far as functionality went, the biggest thing I and my fellow staff members noticed was the fact that it gave us so much more detail on events occurring than other bots we've used and are still using; we were particularly impressed with the 'action' system, which seemed to have all the information needed to allow my team and I the ability to make moderation decisions – even while purposefully overloading it a little by telling my server members to break as many moderation guidelines that it [Overlord] would detect to see if it would miss anything, which it seems it didn't! We especially loved the ability to take suggested actions for certain events, as well as the usage of AI to find content we would've otherwise missed. After playing around with the punishment system, we also really liked the ability it can be adjusted to whatever we need it to be.[…] Although there were only a few commands, the ones that were there worked really well – we got kinda of confused seeing the reactions under the [command] messages at first but we quickly figured out what was going on. Have to say, that's a very nice way of telling the user they've messed up somewhere – better than what I've seen before anyway.

Overall, Overlord did exactly what Jesse said it would, although it can be a little slow [this is due to his connection, not the bot] *and I would've loved to see some more commands added. I think it's rather innovative – something you don't see much of in this space* [Discord bot ecosystem] *anymore. Me and my team are seriously considering using Overlord full-time as our new moderation bot – if that doesn't tell you how impressed we were I'm not sure what will!"* – Member of the Admin Client Group.

3.29 FINALISED STRUCTURES

This section is a more in-depth breakdown of the systems developed through prototypes – specifically in regards to data structures.

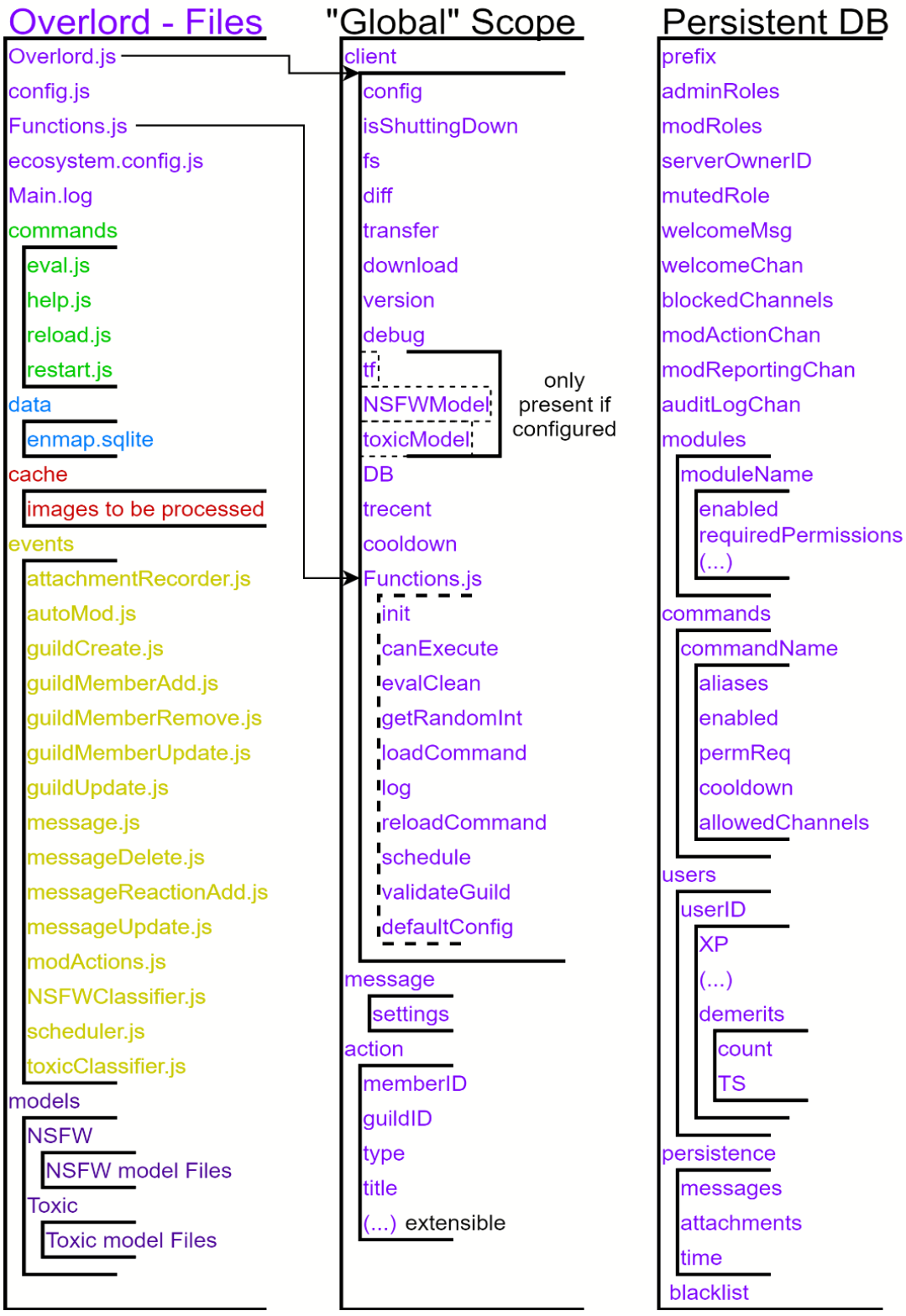

Figure 43 - System Data structure Diagram

This diagram shows all the non-standard and public-scope file/data structures that Overlord uses – with the project directory tree (left) followed by public 'global' scope novel attributes (middle) and the database schema (right). Overall process flow diagrams: these diagrams show the data and execution flow in Overlord as a whole, as well as within the most common event: message.

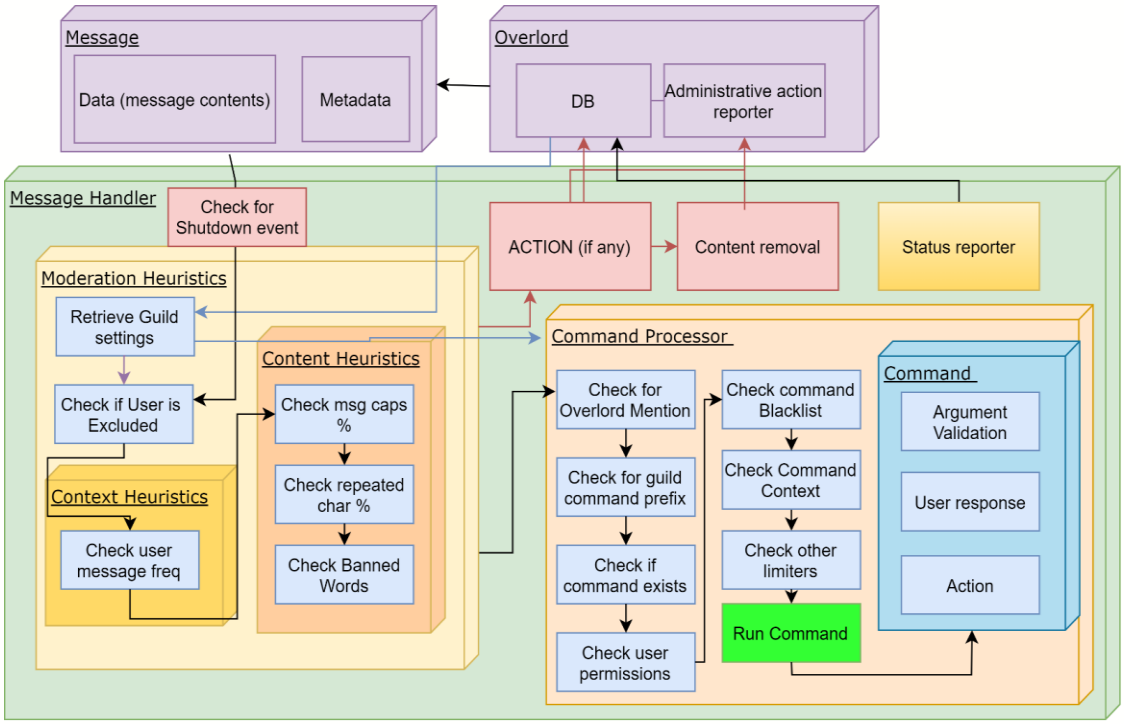

Figure 44 - Overlord Message Event Flow

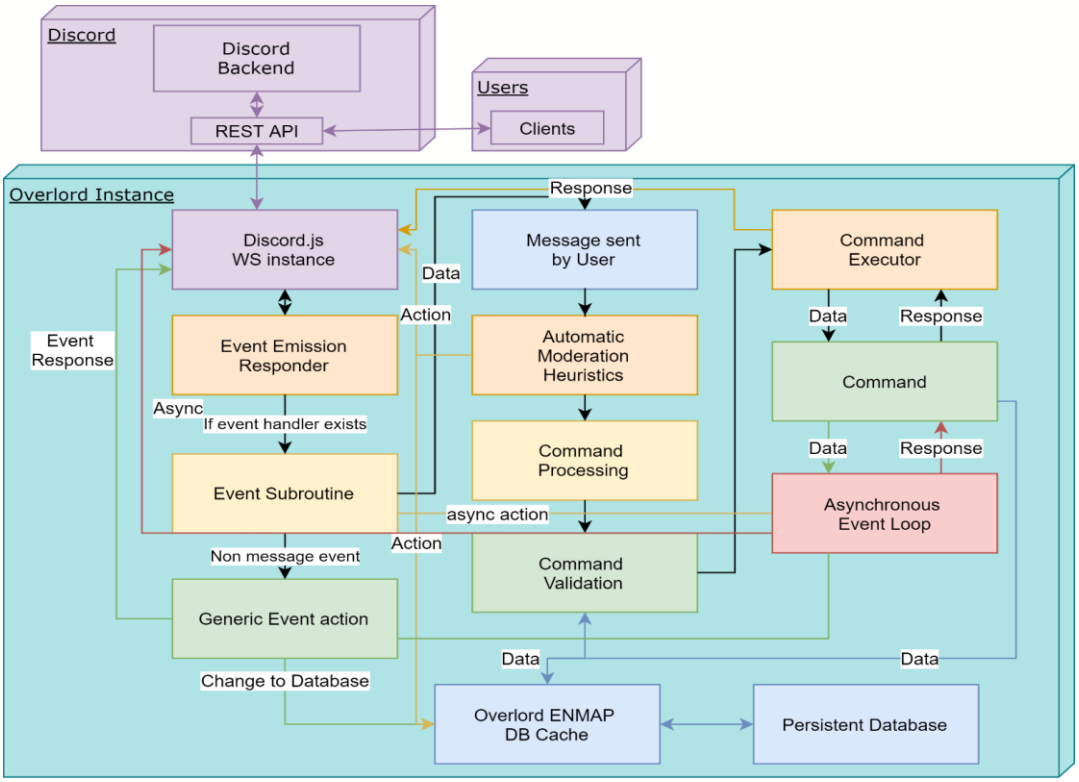

Figure 45 - Overlord System Event Flow

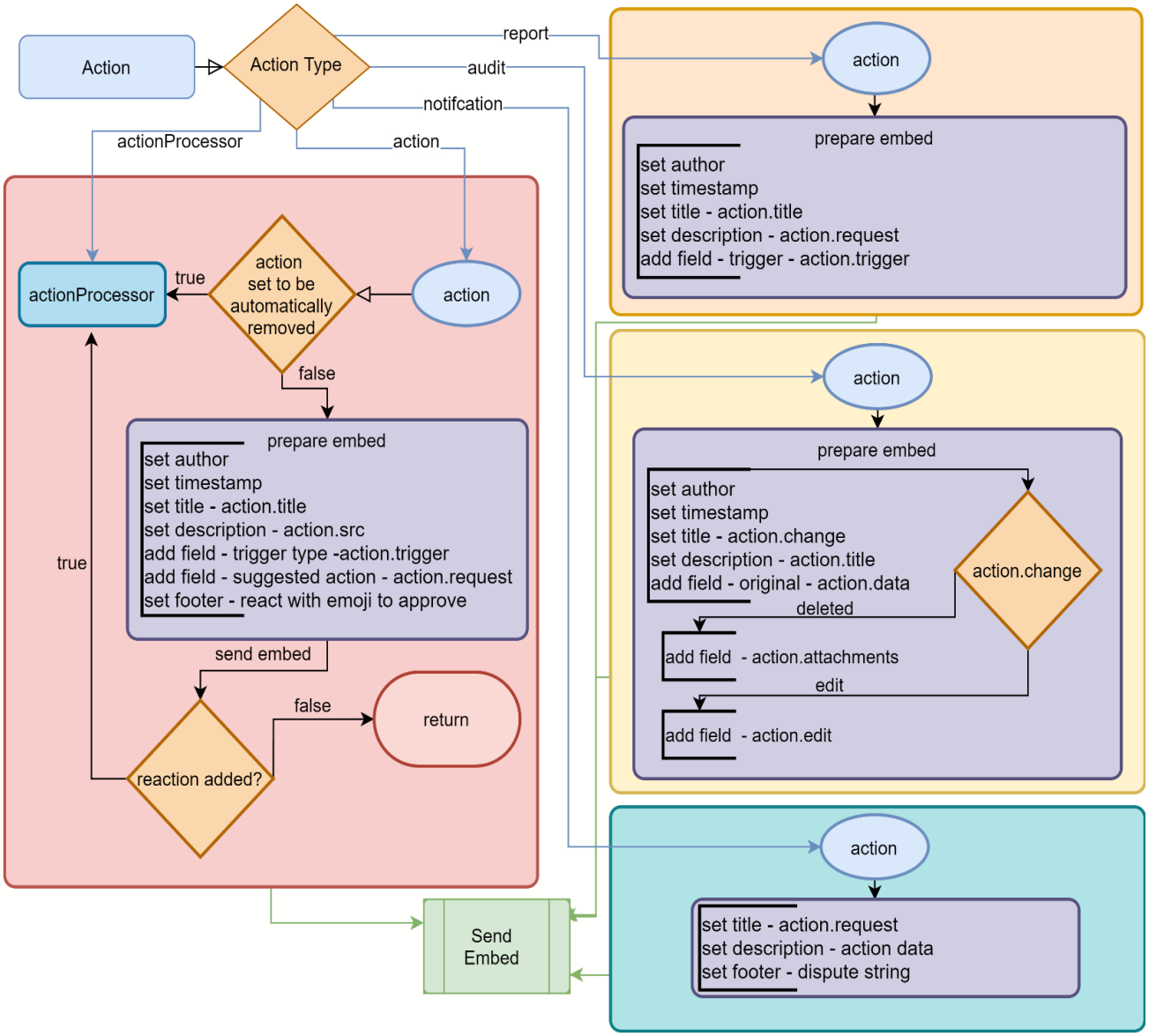

Figure 46 - Automatic Moderation Process Flow Diagram

Data Structure Specification:

Client.DB Database Schema

Although Overlord initially did not need to store much persistent data at all, over time as the concept and goals developed, so did the need for a persistent key:value database with a flexible schema. In Overlord, most entries use their respective Discord-generated Unique ID as the key, eg. each guild has its own database entry, using the guild's ID as the primary key and the schema as the value. The database is split into six functional sections, five of which are discrete. These sections are as follows (referencing Figure 34):

Prefix -> auditLogChan : This is the 'configuration' section for guild-wide configuration – originally its own section, but It was determined that this was not needed due to the low number of keys and lack of planned extensibility. Each of these keys is used as a 'public' config for all modules/commands to access and use as required, unlike the configurations in Modules and Commands, which Is considered 'private', even though data hiding is not actually implemented.

Modules: Contains subObjects for every module (file in ./events/), keyed by the name of the Module in question (the file name within this folder). Contains by default a copy of the defaultConfig (if applicable) and is where

modules executed in the context of the guild get their configuration from. No set schema for these subObjects as the properties required can change drastically between modules, but they have some common keys (see fig 47)

```
module.exports.defaultConfig = {
    enabled: true,
    storageDir: "./cache/",
    keep: true,
    requiredPermissions: ["MANAGE_MESSAGES"]
};
```

Figure 47 - Example Module defaultConfig

Commands: Like Modules, contains subObjects keyed by the name of the command – as with modules, this is the filename of the file located in ./commands/, such as help for ./commands/help.js. like modules, it also contains the configuration for the command – by default, it is a copy of the commands defaultConfig object. Unlike modules, commands have a more rigid schema for the defaultConfig Object (see Fig 48)

```
exports.defaultConfig = {
    aliases: ["alias1"],
    enabled: true,
    permReq: ["BOT_OWNER"],
    cooldown: 1000,
    allowedChannels: [],
};
```

Figure 48 - Example Command defaultConfig

Users: This is the most extensible region of the database and is where almost all the data related to users is stored. Each user has their own subObject keyed to their userID. By default, this only contains a single 'xp' key, but has no set schema and can include other objects such as 'savedState' and 'demerits', both of which Overlord uses for various bits of functionality.

Persistence: This section deals with any Discord related data that the bot needs to keep track of on a guild-wide scale. This includes two subObjects – keyed as messages and attachments - as well as an array keyed as time. Messages is a placeholder for features that are planned to be implemented beyond the current scope of the project. Meanwhile, attachments is used by the bot for storage of any links to attachments for a message that have been uploaded to transfer.sh. These are keyed by messageID, and have a value of an array of URLs. Finally, the array keyed as time is used for the scheduler, as a means of keeping persistent copies of actions "to be done at some point". Each action is an object, that contains a timestamp and other variable properties. This system is not a key:value object system due to the fact that 1). unlike other entries, it has no consistent correlating Discord ID and 2). using an array is easier to manipulate and better suited for this use case (as entries get added/removed very frequently).

Blacklist: Is an object that is extended by a subObject with the key being the command's name that specifies an array of userIDs that are prohibited from executing the given command within the guild. This is way of implementing a guild-wide restriction on command execution.

3.30 EVALUATION

Objective	**Criteria**
Fluent User Experience	The bot needs to be able to fluently communicate information in various forms to users whilst remaining within the Discord app in such a way that no unexpected behaviour on the part of the bot is left unexplained to the user
Consistent uptime (4.5 9's)	The bot needs to be able to rapidly recover from a critical failure state and needs to be mitigating as many failure states as possible to ensure uptime. Under no circumstances should the bot be left in a state wherein it is inoperable due to a failure state within itself. 4.5 9's = 26 minutes of downtime/year (99.995% uptime)
Automatic Moderation	The bot should be able to automatically screen and identify many types of content, penalising it appropriately and enacting punishments if required. It should do so by following a set of heuristic guidelines that it is configured with. The bot should be able to moderate with only optional human intervention, but human intervention should be an option for certain systems.
Ease of administration	The bot should handle automated role assignment, as well as the ability to perform actions to a group of users, as well as implementing other features that streamline the tasks of moderators.
High performance	The bot should make very few unconditional compromises against performance, especially concerning key processing routines. The bot should use a programming language, API wrapper, and database solution geared towards enabling this objective.
Extensibility /modularity	The bot should have almost all functions easily extensible or accessible via client-generated functionality, should the client create additional/new functionality for the bot.
Privacy	The bot should ensure that any crucial information recorded is done so in a secure fashion and users have the option to remove any stored data provided the data does not expire in some fashion.

Objective	**Performance Criteria**
Authentication and API interaction with Discord	Overlord can login using a specified token, and can send and receive events to/from Discord to accomplish actions (e.g. sending messages)
Modularity	Overlord should, upon a configurable trigger, execute a specified file in a specified directory with relevant data – both for client events and commands. the bot should be able to allow for and respect the instanced configuration of these files within the database, as well as being able to deal with the addition/removal of files between restarts. This allows the bot to create as many asynchronous threads as required to scale with processing demands.
Uptime/performance	Overlord should, with use of a 3rd party application manager, restart itself if (and only if) the application is in an unstable state. The bot should ensure that all critical data is stored within a persistent solution and that the bot is otherwise resilient towards errors, with error handling techniques such as try..catch being used throughout. Overlord should prioritise performance heavily, especially for more commonly utilised execution paths – and as such have toggles for more resource-intensive operations.

UX	Usage of embeds as opposed to regular messages wherever possible due to the clarity of conveying larger amounts of information. Integration of theorised UX techniques laid out in 2.3.12, as well as techniques to better help unfamiliar users understand how to use Overlord. Any UGC that is saved by the bot must be inherently transient – it must be automatically removed after a set period of time in order to comply with Discord ToS.
Automatic Moderation	Overlord should, on a configurable basis, discriminate and classify content based on content (media -NSFW and text – Toxicity, char, caps) as well as context (spam). Once the content is classified, a recommended action should be suggested and, on a configurable basis, this action should be processed automatically. Otherwise, a moderator needs to accept this action. Once the action has been approved, the action needs to handled depending on its type and other requirements. Any actions taken by the bot should be reported and affected users notified. Overlord should also offer tertiary functionality to better aid in moderation efforts
Configurability	Overlord should, at start up, ensure that all present commands, modules, guilds and users have the proper configurations present for operation, creating new entries as/when required form default values. Under no circumstances should existing/modified configuration be overwritten by default values. Upon command/module execution, functionality should be beholden to the configuration for this functionality contained within the database entry for the guild this command/module.
Quality of Moderation	Overlord should Implement key auditing features to ensure that Moderators have all the information they need about a potential case at all times – ensuring that events that react to any changes to any content are logged, as well as any actions taken by the bot that could potentially affect their duties. This should be done through processing of messageDelete and messageUpdate events, sent via embeds.

Table 33 - Copy of the Technical Objectives Table

Using the performance criteria outlined in the Project objectives, as well as the technical development objectives, It will be determined if and how Overlord has met an area of these two sets of criteria. The technical development Criteria expand on the requirements laid out in the initial Criteria for fulfilling the same Objectives, going into more technical detail about how the initial Criteria will be met by the technical solution.

Objective/Criteria	Development Objectives/Criteria	Areas of implementation
User Experience (UX): *"The bot needs to be able to fluently communicate information in various forms to users whilst remaining within the Discord app in such a way that no unexpected behaviour on the part of the bot is left unexplained to the user".* Privacy : *"The bot should ensure that any crucial information recorded is done so in a secure fashion and users have the option to remove any stored data provided the data does not expire in some fashion."*	Via technical development objectives - UX - "Usage of embeds as opposed to regular messages wherever possible due to the clarity of conveying larger amounts of information. Integration of theorised UX techniques laid out in 2.3.12, as well as techniques to better help unfamiliar users understand how to use Overlord. Any UGC that is saved by the bot must be inherently transient – it must be automatically removed after a set period of time in order to comply with Discord ToS.".	Subsection 4.2.12 Prototype 10: Rich Presence. Prototype 15: Scheduler and Persistence Prototype 16: On-demand reporting embeds for errors with commands. Prototype 18: Attachment recording Prototype 21: Embeds used for action types. Finalised Codebase – help.js message,js Functions.js attachmentRecorder.js scheduler.js
Consistent Uptime + High Performance: *"The bot needs to*	Via technical development objectives – *"Overlord should,*	Subsections 2.4.1 -2.4.3 Prototype 15 – Custom scheduler

be able to rapidly recover from a critical failure state and needs to be mitigating as many failure states as possible to ensure uptime. Under no circumstances should the bot be left in a state wherein it is inoperable due to a failure state within itself" and *"The bot should make very few unconditional compromises against performance, especially concerning key processing routines. The bot should use a programming language, API wrapper, and database solution geared towards enabling this objective."*	*with use of a 3rd party application manager, restart itself if (and only if) the application is in an unstable state. The bot should ensure that all critical data is stored within a persistent solution and that the bot is otherwise resilient towards errors, with error handling techniques such as try..catch being used throughout. Overlord should prioritise performance heavily, especially for more commonly utilised execution paths – and as such have toggles for more resource-intensive operations."*	Prototype 25 – PM2 Integration Finalised Codebase – Overlord.js and Config.js Generalised Optimisations throughout the codebase
Automatic Moderation – *"The bot should be able to automatically screen and identify many types of content, penalising it appropriately and enacting punishments if required. It should do so by following a set of heuristic guidelines that it is configured with. The bot should be able to moderate with only optional human intervention, but human intervention should be an option for certain systems."*	Automatic Moderation – *"Overlord should, on a configurable basis, discriminate and classify content based on content (media -NSFW and text – Toxicity, char, caps) as well as context (spam). Once the content is classified, a recommended action should be suggested and, on a configurable basis, this action should be processed automatically. Otherwise, a moderator needs to accept this action. Once the action has been approved, the action needs to handled depending on its type and other requirements. Any actions taken by the bot should be reported and affected users notified. Overlord should also offer tertiary functionality to better aid in moderation efforts."*	Subsections 2.4.6 – 2.4.11 Prototype 17 – Moderation Heuristics Prototype 19 – NSFW UGC Classification Prototype 20 – UGC Toxicity Classifier Prototype 21 – Automated Moderation Backend Prototype 22 - Punishment Finalised Codebase – autoMod.js, message.js, NSFW/toxicClassifier.js, modActions.js
Ease of administration – *"The bot should handle automated role assignment, as well as the ability to perform actions to a group of users, as well as implementing other features that streamline the tasks of moderators."*	Quality of Moderation – *"Overlord should Implement key auditing features to ensure that Moderators have all the information they need about a potential case at all times – ensuring that events that react to any changes to any content are logged, as well as any actions taken by the bot that could potentially affect their duties. This should be done through processing of messageDelete and messageUpdate events, sent via embeds."*	Subsection 2.4.10 Prototype 18 – Attachment Recording Prototype 11 -Implementation of client events Prototype 24 – Event Stub Development Finalised Codebase – messageDelete.js messageUpdate.js guildMemberAdd.js guildMemberRemove.js
Extensibility /modularity – *"The bot should have almost all functions easily extensible or accessible via client-generated functionality, should the client create additional/new functionality for the bot."*	Modularity – *"Overlord should, upon a configurable trigger, execute a specified file in a specified directory with relevant data – both for client events and commands. the bot should be able to allow for and respect the*	Subsection 2.4.5 Prototype 3 – Modular event code Prototype 5 -Database initialisation Prototype 13 – Basic Modular Commands

	instanced configuration of these files within the database, as well as being able to deal with the addition/removal of files between restarts. This allows the bot to create as many asynchronous threads as required to scale with processing demands."	Prototype 16 – Finalise message event Finalised Codebase – Message.js Functions.js

Evaluation:

Overall, looking at the Project as a whole, I'm very happy with the way in which Overlord has developed to fulfil the initially daunting objectives laid out for it – this coupled with the excellent response to a live test from a member of the admin client group allows me to say that Overlord has accomplished almost every aspect of the Objectives laid out for it. That is not so say, however, that there are not some potential improvements that could be made to the bot.

Potential improvement 1). Addition of more commands

Currently, Overlord has next to no user interactivity via commands due to the focus on providing moderation features over general user features. However, the command groundwork has been entirely laid to create commands implementing new functionality over the course of Overlord's continued development. This was also the primary point of criticism that my live test user(s) had about the bot.

2). Migration to a higher API version

Whilst overlord was in the midst of development, a new Discord.js wrapper version was released – while this doesn't add much in terms of functionality, it does add a lot of under-the-hood improvements to the wrapper and applications built on it. Due to the drastic differences between the data structures between the latest version and the one Overlord is written for, a quick migration was never an option, and had to be shelved in favour of more functionality.

3.) Further Training of the NSFW Neural Network

Currently, the NSFWClassifier's Neural Network is one that has been trained by a 3rd party on a relatively small dataset and for only a couple hundred training hours – lacklustre in comparison to that of the Toxicity Neural Network. It became apparent that the accuracy of this Neural Network could be greatly increased with additional training and a larger dataset.

4.) Creation of simplified guidance material / an 'API' reference

Overlord is a rather complex application, and for any potential developers, finding out exactly how to create functionality for Overlord is equally daunting. Given the emphasis this capability has had as a result of my modular approach, creating a set of guidance material to explain how to create new capabilities would be of great help to enabling this creation.

With these improvements in mind, as well as a few members of my Client group already Adopting Overlord as their de-facto Moderation bot, I plan to continue development of the project well into the future beyond A-Level as Overlord (hopefully) continues to gain popularity among server administrators and users alike.

4 APPENDIX - FINALISED CODEBASE

This ending section contains the final versions of the entire codebase, in order of the Overlord-Files schema, and only including novel/relevant files.

Some files include minor optimisations/tweaks from their Prototype counterparts – these generally do not alter the fundamental functionality, and only tend to make the code more efficient. All functions within Overlord have a JSDoc comment, which can be used to compile external documentation. As JS is not an indentation-blocking language, some indentation has been altered to enable the code to better fit the page.

4.1 OVERLORD.JS

```
/**Dependancy Import and initialisation */
console.time("init");
const Discord = require("discord.js");
const enmap = require("enmap");
//autoRecconect: client will automatically reconnect to the WS if connection is lost
//messageCacheMaxSize: the maximum number of messages that the bot should keep in ca
che before indescriminantly discarding.
//messageCacheLifetime: the number of ms that a message should be kept in the cache
before being swept.
//messageSweepInterval: number of ms between sweeps of the message cache.
//discord client - main way that the bot interfaces with discord. extended with some
 custom attributes later on.
//configured to keep 20k messages max, with a max lifetime of a day, with messages s
wept every 50 seconds
let client = new Discord.Client(
{ autoReconnect: true, messageCacheMaxSize: 20000, messageCacheLifetime: 86400000, m
essageSweepInterval: 50000, partials: ['MESSAGE', 'REACTION'] });
//load configuration settings from the config file - bind to config to make it 'glob
al'
client.config = require("./config.js");
//flag used to signify to other events to halt any processing. only changed when the
 gracefulShutdown event is emitted.
client.isShuttingDown = false
client.fs = require("fs");
//used to determine changes between two instances of an object
client.diff = require("deep-object-diff").detailedDiff;
//used to upload files to transfer.sh
client.transfer = require("transfer-sh");
//used to download files from discord
client.download = require("download-file");
client.version = "1.0.3.07042020"; //release.major.minor.date
/*debug flag set if the bot is not run with the enviroment variable "production".
if this is not set, the bot ignores all non-tagged logging.*/
client.debug = (process.env.NODE_ENV === "production" ? false : true)
```

```
//console bootup starting message
console.log(`!== Overlord v${client.version} Intialisation starting. current date/ti
me is ${new Date()} ==! `);


/**
 * This function Loads the neural net models for  both the NSFWClassifier as well as
 the ToxicityClassifier.
 * @param {object} client
 */
let modelLoad = async (client) => {
 //checks that the models aren't disabled in the main bot config.
 if (!client.config.enableModels) return;
 client.tf = require("@tensorflow/tfjs-node");

//flag to tell TF that this is a production app. nets some major performance improve
ments.
 client.tf.enableProdMode()
 var toxicity = require("@tensorflow-models/toxicity");
 //loads the NSFWModel via NSFWjs Using local model files.

client.NSFWModel = await require("nsfwjs").load("file://./models/NSFW/", { size: 299
 });
 //creates a new instance of the ToxicityClassifier
 client.toxicModel = new toxicity.ToxicityClassifier;
 //overwrite default LoadModel method due to *hard coded* reliance on web-
based model files.
 // I didn't like this reliance so I made it use local files instead.
 client.toxicModel.loadModel = () => {
  return require("@tensorflow/tfjs-
converter").loadGraphModel("file://./models/toxic/model.json");
 };
 //wait for the models to load.
 await client.toxicModel.load();
 client.log("Models loaded!");
}
//triggers the loading of the NeuralNet Models. for some reason a self-
invoking anonymous function refused to work.
modelLoad(client)
/** assigns the client Object a New enmap instance ("DB") - */
client.DB = new enmap({
 //identifier - shows that this ENMAP is persistent
 name: "DB",
 //automatically fetch DB keys when required
```

```
 autoFetch: true,
 //do not fetch all data from the DB into the memory cache at startup
 fetchAll: false,

//ensure will do so for properties of an object, rather than just the top level stru
cture.
 ensureProps: true,
 //the directory where the data for this persistent ENMAP will be stored.
 dataDir: client.config.datadir
});
//two non-
persistent ENMAP instances - same functionality as client.DB, just transient.
//stands for talked recent - "who talked recently?" - per-guild with per-
channel subObjects.
//eg: 54546834767588:{33750293334875:[]} - guildID:channelID:array
client.trecent = new enmap()
//used for tracking cooldowns for commands - per-guild and per-channel subObjects.
client.cooldown = new enmap()

/** Bind the exportable functions from Functions.js to the client object as methods.
 */
require("./Functions.js")(client);

/**main part of the debug system to monitor any/all changes to the ENMAP Database */
client.DB.changed((Key, Old, New) => {
 //custom logging for recording database changes.
 client.log(`${Key} - ${JSON.stringify(client.diff(Old, New))}`);
})

/** used to gracefully shutdown the bot, ensuring all current operations are complet
ed successfully and inhibiting new operations from occuring
 * used an operation 'locking' variable
(client.isShuttingDown) that if true prevents any new commands from being executed.
 * also uses setImmediate to wait for any I/O operations to prevent things such as D
B corruption etc.
 */
client.on("gracefulShutdown", (reason) => {
 //log aknowledging the request

client.log(`Successfully Received Shutdown Request - Reason: ${reason} -  bot Proces
s commencing shutdown.`, "WARN");
 //'global' flag
 client.isShuttingDown = true
 //after 5.5 seconds, and after all I/O activity has finished, quit the application.
```

```
 setTimeout(() => { setImmediate(() => { process.exit(0); }); }, 5500);
})

/**
 * every 120 seconds, clears out the loaded database keys to help reduce the memory footprint of the bot.
 */
setInterval(() => { client.DB.evict(client.DB.keyArray()); }, 120000);

/** PM2 SIGINT and Message handling for invoking a graceful shutdown through PM2 on both UNIX and windows systems */
process
 //unix SIGINT graceful PM2 app shutdown.
 .on("SIGINT", () => {
  //triggers a graceful shutdown
  client.emit("gracefulShutdown", "PM2")
 })
 //Windows "message" graceful PM2 app shutdown.
 .on("message", (msg) => {
  if (msg === "shutdown") {
   client.emit("gracefulShutdown", "PM2")
  }
 })
 .on("uncaughtException", (err) => {

/** process error catching with custom stacktrace formatting for ease of reading */
  console.dir(err.stack.replace(new RegExp(`${__dirname} / `, "g"), "./"));
  client.emit("gracefulShutdown", "Exception")
 });

/** catches and logs any Discord.js Client errors */
client
 .on("error", error => { client.log(error, "ERROR"); })
 /** if the client disconnects, report the disconnection */
 .on("disconnect", (event) => {
  client.log("Client disconnected! restarting...\n" + event, "ERROR")
  /**this event signifies that the connection to Discord cannot be re-established and will no longer be re-attempted.
  so we restart the bot process to (hopefully) fix this
   (note: requires PM2 to restart the process).*/
  client.emit("gracefulShutdown", "Disconnect");
 });
```

```
/** authenticates the bot to the Discord
backend through usage of a Token via Discord.js.
 * waits for the Database to load into memory, then starts the initialisation. */
client.login(client.config.token);
//emited when the client is fully prepared and authenticated with Discord.
client.on("ready", () => {

//waits for DB to load fully before initialising (otherwise will try to write to a n
on-existant database.)
 //this is why I do not use the ready event to start the initialisation directly.
 client.DB.defer.then(
  client.init(client)
 );
});
```

4.2 CONFIG.JS

```
/**
 * @exports ownerID
 * @exports token
 * @exports datadir
 * @exports status
 * @exports enableModels
 * @exports preLoad
 */
module.exports = {
 ownerID: "1506936700000000", //discord ID of the  bot's Owner.
 token:
  process.env.NODE_ENV === "production" //turnary operator for token selection
   ? "NTc2MzM3ODI2MAAAAAAAAwODkAAAAAAAAAAAAAAAAAAAAAAAAA " //bot's
Production Token (true)
   : "NjQ4OTU5OAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", //dev token (false)
 datadir: "./data", //data storage location for the ENMAP-SQLite backend.

status: "@ me for help – version {{version}} - now on {{guilds}} guilds!",//status m
essage of the bot.
 enableModels: true, //global toggle for enabling the ML Models

preLoad: true, //preload data - can reduce performance at bootup but increases elsew
here.
};
```

4.3 FUNCTIONS.JS

```
/**
```

```
 * contains functions that are bound to the client object at startup. DO NOT EDIT (p
ls)
 * @exports init - initialises the bot
 * @exports validateGuild - validates a guild's data
 * @exports log - custom logging handler
 * @exports canExecute - checks if a command can be executed by a user or not
 * @exports loadCommand - used to load a command file from disk
 * @exports schedule - used to schedule tasks
 * @exports reloadCommand - used to reload a command back into memory
 * @exports evalClean - used to clean the output of Eval
 * @exports getRandomInt - used to generate random integers
 */


module.exports = (client) => {
 //alias for fs
 const fs = client.fs
 //contains timers for guild scheduler instances
 client.timeouts = new Map();
 //sets the base directory to the process' current working directory.
 client.basedir = process.cwd();


 /**
  * Initialisation routine for the client, after the client has been authenticated a
nd connected to discord,
  * as well as after the ENMAP database is ready. handles the initialisation of ever
ything needed for proper function.
  */
 client.init = (client) => {
  //statement to make the owner aware the bot is in debug mode.
  if (client.debug) { client.log("BOT IS IN DEBUG MODE", "WARN") }
  //logging statement to see what user is being logged into.
  client.log(`client logging in as ${client.user.tag}`, "INFO")
  if (client.guilds.size == 0) {
   //aborts if the bot is not part of any guilds
   throw new Error("No Guilds Detected! Please check your token. aborting Init.");
  }
  if (!client.user.bot) {
   //aborts if it detects it has been given a user's token instead of a bot's token.

throw new Error("Warning: Using  bots on a user account is (for the most part) forbi
dden by Discord ToS. Please Verify your token!");
  }
```

```
//if preloading is enabled, iterate over every channel and load messages into bot me
ssage cache.
  if (client.config.preLoad) {
   client.channels.forEach(channel => {
  //categories are classed as channels and are thus ignored.
  if (channel.type === "text") {
   channel.fetchMessages({ limit: 100 }).then(c => { return })
  }
   });
  }

//status - small message that shows up under the profile of the bot. customisable in
 config.js.
  //replace placeholders with specified content

var status = client.config.status.replace("{{guilds}}", client.guilds.size).replace(
"{{version}}", client.version);
  client.user.setPresence({ activity: { name: status }, status: "active" });
  //load the files for the events
  const eventFiles = fs.readdirSync("./events/");
  client.log(`Loading ${eventFiles.length} events from ${client.basedir}/events/`);
  eventFiles.forEach(eventFile => {
   //skip anything that's not a .js file
   if (!eventFile.endsWith(".js")) return;
   //strip the file extension
   const eventName = eventFile.split(".")[0];
   client.log(`attempting to load event ${eventName}`)
   //load it from disk
   const eventObj = require(`./events/${eventFile}`);

//bind it - on eventName, eventName(client, ...other args) as everything needs clien
t

//this means that the event will always be called with client as the first arguement
.
   client.on(eventName, eventObj.bind(null, client));
   //if the event has a default config, load it if it's not already present
   if (eventObj.defaultConfig) {
  client.guilds.forEach(guild => {

//if the value for the config exists, return the existing value, else write the give
n value.
   client.DB.ensure(guild.id, eventObj.defaultConfig, `modules.${eventName}`)
```

```
  })
   }
   //log that the loading was successful.
   client.log(`Bound ${eventName} to Client Sucessfully!`);
   //delete the file from the resolve cache as we no longer need it.
   delete require.cache[require.resolve(`./events/${eventFile}`)];
  });

//iterates over each guild that the bot has access to and ensures they are present i
n the database
  client.guilds.forEach(guild => {
   client.validateGuild(client, guild);
  });
  //cleans the cache from any lingering operations that may have been interrupted.
  fs.readdir("./cache/", (err, files) => {
   if (err) throw err;
   client.log(`Deleting ${files.length} files from cache...`)

//message to alert the operator/bot owner that a unexpected shutdown probably occurr
ed.
   if (files.length > 1) {

client.log("Cache Remnants detected - this should only occur if an unexpected shutdo
wn was invoked!", "WARN")
   }
   for (const file of files) {
  //delete all the remenant files
  fs.unlink(("./cache/" + file), err => {
   if (err) throw err;
  })
   }
  })
  //ends the initialisation timer as the bot is fully operational by this point.
  console.timeEnd("init");
  //

client.log(`Ready to serve in ${client.channels.size} channels on ${client.guilds.si
ze} servers, for ${client.users.size} users.`, "INFO");
  try {

//sends a "I'm alive!" message to the owner so they know when the bot is back online
.
   (client.users.get(client.config.ownerID))
```

```
.send(`Ready to serve in ${client.channels.size} channels on ${client.guilds.size} s
ervers, for ${client.users.size} users.`);
  } catch (err) {

client.log("I'm not in the guild with the owner - unable to send bootup notification
!", "INFO")
  }
 }


/**validates a Guilds's configuration properties and database 'Presence'. called at
startup and when a new guild is created.
  * @param {object} client
  * @param {object} guild
  */
 client.validateGuild = (client, guild) => {
  //ensures each server exists within the DB.
  var guildData = client.DB.ensure(guild.id, client.defaultConfig);
  //uses the defaultConfig as the basic framework if the guild really doesn't exist

//quick and easy way of finding roles with common names and assigning them automagic
ally.
  guild.roles.forEach(role => {

client.log(`Testing role with name ${role.name} for Admin/Mod/Muted availability.`);

if (["Admin", "Administrator"].includes(role.name)) { client.DB.push(guild.id, role.
id, "adminRoles"); }

if (["Mod", "Moderator"].includes(role.name)) { client.DB.push(guild.id, role.id, "m
odRoles"); }

if (["Muted", "Mute"].includes(role.name)) { client.DB.set(guild.id, role.id, "muted
Role"); }
  });
  client.DB.set(guild.id, {}, "commandsTable")

//ensures entry for the guild in trecent (used for antispam - stands for TalkedRecen
t, as in "Who has talked (sent messages) recently?")
  client.trecent.ensure(guild.id, {})

//ensures entry for the guild for command cooldowns - used to ratelimit command exec
ution.
  client.cooldown.ensure(guild.id, {})
```

```
//ensures each server has all it's users initialised correctly - each user is given
an object with just a property for XP (for now)
  guild.members.forEach(member => {
   //this object is extended/pruned as required by various submodules, eg demerits.
   client.DB.ensure(guild.id, { xp: 0 }, `users.${member.id}`);
  });
  //evict all those recently loaded keys from memory cache.
  client.DB.evict(client.DB.keyArray())
  //logs completion of validation.
  client.log(`Sucessfully Verified/initialised Guild ${guild.name} to DB`);
  //finds and automatically sets the ownerID for the server -> automatic admin
  client.DB.set(guild.id, guild.owner.user.id, "serverOwnerID");
  //load commands with default config into guild config (if it doesn't exist)
  const commandFiles = fs.readdirSync("./commands/");

client.log(`Loading ${commandFiles.length} events from ${client.basedir}/commands/`)
;
  commandFiles.forEach(command => {
   //ignore all non-.js files
   if (!command.endsWith(".js")) return;
   //strip file extention
   var command = command.split(".")[0];
   //run another function to fully load the command
   client.loadCommand(command, guild.id);
  });

//check module permission requirements to determine the permissions required by the
bot. starts with a basic set:
  let reqPermissions = ["SEND_MESSAGES", "READ_MESSAGES", "VIEW_CHANNEL"]
  //iterates over every module, checking the permissions declared as required.
  Object.keys(guildData.modules).forEach(key => {
   let Module = guildData.modules[key]
   if (!Module.requiredPermissions) return;
   Module.requiredPermissions.forEach(perm => {
  //append missing perms to the array.
  if (!reqPermissions.includes(perm)) { reqPermissions.push(perm) }
   })
  });
  //log requested permissions

client.log(`Requested permissions for server ${guild.name} : ${reqPermissions.toStri
ng()}`)
```

```
  //find the permissions that the client is missing.

let missingPerms = reqPermissions.filter(perm => !(guild.members.get(client.user.id)
.permissions.toArray()).includes(perm))
  //override if the user has administrative permissions

if ((guild.members.get(client.user.id).permissions.toArray()).includes("ADMINISTRATO
R")) { missingPerms = [] }
  //if there are any missing perms...
  if (missingPerms.length >= 1) {
   //notify the bot owner as well as guild admins to the missing permissions.

client.log(`bot is missing permisions : ${missingPerms.toString()} in guild ${guild.
name}`, "ERROR")
   //send a message to the guild to request the missing permissions.

guild.channels.get(guildData.modActionChan) || guild.channels.filter(chan => chan.ty
pe === "text").values().next().value
  .send(`I am missing permissions : ${missingPerms.toString()}!`)
  }
  //check attachments, and remove those that have expired.
  let attachments = client.DB.get(guild.id, "persistence.attachments")
  client.log(`Verifying Persistence data for guild ${guild.name}`)
  //for every attachment...
  Object.keys(attachments).forEach(key => {
   //if the attachment has expired
   if (attachments[key].expiry < new Date()) {
  //delete the attachment
  client.DB.delete(guild.id, `persistence.attachments.${key}`)
   }
  })
  //check any persistent messages for expiary
  let messages = client.DB.get(guild.id, "persistence.messages")
  //for every message...
  Object.keys(messages).forEach(key => {
   //get the 'key' of the message - guildID:channelID:messageID
   let messageKey = messages[key].key

client.guilds.get(guild.id).channels.get(messageKey.split(":")[0].toString()).fetchM
essage(messageKey.split(":").toString()[1]).catch(err => {

//error only if the message is no longer reachable - eg deleted - therefore remove f
rom DB.
  client.DB.remove(guild.id, `persistence.messages.${key}`)
```

```
   })
  })
  //manually trigger the scheduler every 300 seconds to check this guild
  setInterval(() => { client.emit("scheduler", guild.id) }, 300000, client, guild)
  // start the scheduler for this guild
  client.emit("scheduler", guild.id)
 };


 /**
  * custom logging system - uses optional tags as well as stack tracing to determine
 caller locations for ease of access.
  * 4 'levels' - untagged - ignored if not in debug mode
  * INFO - information, always displayed
  * WARN - warning, always displayed
  * ERROR - Error - something has gone wrong. always displayed.
  * helpful as it allows the owner to differentiate types of information easily.
  */
 client.log = (message, type) => {

//using stacktrace to locate caller code/function. useful for determining exactly wh
ere information came from.

let caller = ((new Error).stack).split(" at ")[2].trim().replace(client.basedir, "."
)
  //message - comprised of the type and stringified message, as well as the caller.
  //each type/flag gets different 'treatment'
  if (!type) type = "DEBUG"
  //formats the message with some addition info.
  let msg = `[${type}] ${(JSON.stringify(message)).replace(/"/g, "")} 〖${caller}〗 `
  switch (type) {
   case "ERROR":
  console.error(msg);
  break;
   case "WARN":
  console.warn(msg);
  break;
   case "INFO":
  console.log(msg)
  break;
   case "DEBUG":
  if (!client.debug) break
  console.log(msg)
```

```
  break
   default:
  console.log(`Invalid logging type! ${msg}`)
  }
 }


 /**
  * given a command name as well as context (eg guild/channel) from the message obje
ct, determines
  * whether or not the given user can execute the command they are requesting to exe
cute.
  */

client.canExecute = (client, message, cmdName) => { //check if a user can execute a
command or not
  //gets the current guilds configuration from the message via the non-
standard settings attribute - added in message.js
  let cmdCfg = message.settings.commands[cmdName]

//checks lots of conditions to determine if the user can execute a command - reports
 what 'stage' they fail at, if any.
  //if the user is the owner, pass.
  if (message.author.id === client.config.ownerID) {
   return "passed"
  }
  //if the command doesn't exist, fail with error 'nonexistant'
  if (!cmdName) {
   return "nonexistant"
  }
  //if the command is disabled on a guild-wide basis, fail with 'disabled'
  else if (!cmdCfg.enabled) {
   return "disabled"
  }
  //if the user is in the guild's blacklist for the command, fail with 'blacklist'

else if (Object.keys(client.DB.ensure(message.guild.id, [], `blacklist.${cmdName}`))
.includes(message.member.id)) {
   return "blacklist"
  }

//if the user isn't the owner and doesn;t have the correct permissions required by t
he command - fail with 'perms'
```

```
else if (cmdCfg.permReq.includes("BOT_OWNER") || !message.member.permissions.has(cmd
Cfg.permReq, true)) {
   return "perms"
  }

//if the array allowedChannels has 1+ entries, and it doesn't include the current ch
annel, fail with 'channel'

else if (!((cmdCfg.allowedChannels.length === 0) ? true : (!cmdCfg.allowedChannels.i
ncludes(message.channel.id)))) {
   return "channel"
  }
  //if the user is in the commands cooldown period, fail with 'cooldown'

else if (client.cooldown.get(message.guild.id, cmdName).filter(u => u === message.me
mber.id).length) {
   return "cooldown"
  } else {
   //if they didn't fail any checks, pass.
   return "passed"
  }
 }
 /**
  * loads a specified command to a specified guild. if no guild is specified, loads
the command for all guilds.
  * @param {string} command - the name of the command (true, no aliases)
  * @param {string} guildid - optional. ID of the guild to load the command for.
  */
 client.loadCommand = (command, guildid) => {
  if (!guildid) {
   //if no guildid is specified, loads the command for all guilds.
   //useful for initialisation of the bot, and for reloading (as these are bot-wide)
   client.guilds.forEach(guild => { client.loadCommand(command, guild.id); });
  }
  //try:catch for any errors.
  try {
   //loads contents of 'command'.js from disk
   var cmdObj = require(`${client.basedir}/commands/${command}.js`);
   //ensures each guild has the configuration data required for the command command.
   client.DB.ensure(guildid, cmdObj.defaultConfig, `commands.${command}`);
   //ensures command's presence in client.cooldown
   client.cooldown.ensure(guildid, [], command)
```

```
   //ensures that the aliases for the command are present.
   //for every alias...
   client.DB.get(guildid).commands[command].aliases.forEach(alias => {
  //ensure the alias maps to 'command' in commandsTable
  client.DB.ensure(guildid, command, `commandsTable.${alias}`)
  //log the sucessful binding of each alias.

client.log(`bound alias ${alias} to command ${command} in guild ${client.guilds.get(
guildid).name}`);
   });
  } catch (err) {
   //catch and log any errors
   client.log(`Failed to load command ${command}! : ${err}`, "ERROR")
  }
 };
 /**
  * wrapper for scheduling events (actions) for a specified guild using the schedule
r.
  * @param {string} guildID - ID of the guild the action belongs to.
  * @param {object} action - object containg data to be scheduled for execution.
  */
 client.schedule = async (guildID, action) => {
  //add the action object to the database under persistence.time
  client.DB.push(guildID, action, "persistence.time")
  //invoke scheduler to update the timeout
  client.emit("scheduler", guildID)
 }
 /**
  * command that resets the in-
memory copy of a command with that now on disk. very useful for rapid development.
  */
 client.reloadCommand = (commandName) => {
  try {
   //deletes the cached version of the command, forcing the next execution to re-
load the file into memory.
   delete require.cache[require.resolve(`./commands/${commandName}.js`)];
   //load the command back into memory from disk.
   client.loadCommand(commandName);
  } catch (err) {
   //catch and log any errors
   client.log(`Error in reloading command ${commandName} - \n${err}`, "ERROR");
  }
```

```
};


//BELOW ARE FROM ANOTHER SOURCE - DO NOT ASSESS
/**
 * cleans the content from the eval command to remove potentially dangerous informa
tion - eg the token of the bot. general sanitisation.
 * @param {object} client
 * @param {any} text - any due to the result from eval potentially being anything.
 */
client.evalClean = async (client, text) => {

//checks if the evaled code is that of a promise, if so, awaits for the promise to r
esolve before continuing.
  if (text && text.constructor.name == "Promise")
   text = await text;
  if (typeof evaled !== "string")

//inspects the content so we don't just get [object Object] or the full object via s
tringify.

//this provides a single layer of informantion, eg test:{a:{"b"}} => test:{a:{object
}}.
   text = require("util").inspect(text, { depth: 0 });
  //replaces the token with "[BOT_TOKEN]"

text = text.replace(/@/g, "@").replace(/`/g, "`").replace(client.config.token, "[BOT
_TOKEN]");
  return text;
}; //full disclosure: this code was copied off Etiket2 (another Discord
bot) as it is undoubtedly the best way to do this.


/**
 * returns a random integer between two numbers (max exclusive, min inclusive.)
 * @param {int} minimum
 * @param {int} maximum
 */
client.getRandomInt = (min, max) => {
 min = Math.ceil(min);
 max = Math.floor(max);
 //The maximum is exclusive and the minimum is inclusive
 return Math.floor(Math.random() * (max - min)) + min;
};
```

```
//everything between the other DO NOT ASSESS notice and this one is from, well, anot
her source and shouldn't be assessed.
 //base object used as a schema for every guilds database entry.

//used as a template for development as well as for initialising guilds. contains so
me populated defaults, but mostly just structure.
 client.defaultConfig = {
  prefix: "$",
  adminRoles: [],
  modRoles: [],
  serverOwnerID: 0,
  mutedRole: 0,
  welcomeMsg: "Welcome {{user}} to {{guildName}}!",
  welcomeChan: 0,
  blockedChannels: [],
  modActionChan: 0,
  modReportingChan: 0,
  auditLogChan: 0,
  modules: {},
  commands: {},
  users: {},
  persistence: {
   messages: {},
   attachments: {},
   time: [],
  },
  blacklist: {}
 };


};
```

4.4 ECOSYSTEM.CONFIG.JS

```
module.exports = {
 // Options reference: https://pm2.io/doc/en/runtime/reference/ecosystem-
file/ - very helpful resource!
 apps: [{
  name: "Overlord",
  script: "./Overlord.js",
  instances: 1,
  autorestart: true,
  watch: false,
  env: {
```

```
   NODE_ENV: "production"
  },
  max_memory_restart: "2G",
  combine_logs: true,
  log: ".\\main.log",
  log_date_format: "YYYY-MM-DD HH:mm:ss"
 }],
};
```

4.5 ./COMMANDS/EVAL.JS

```
/**
 * small command only usable by the bot owner to debug code whilst an instance is ru
nning using eval.
 * @param {object} client
 * @param {object} message
 */
exports.run = async (client, message, args) => {

const code = args.slice(1).join(" "); //removes the commandname from args, and then
executes the following statements/code.
 try {
  var evaled = eval(code); //evaluate (run) the code - VERY DANGEROUS!!
const clean = await client.evalClean(client, evaled); //small function to ensure tha
t the output doesn't contain valuable info, primarily the bot's token.
message.channel.send(`\`\`\`js\n${clean}\n\`\`\``); //send evaluated code as a code
block.
 } catch (err) {
  //send any errors to the channel.
message.channel.send(`\`\` \`\`\`js\n${await client.evalClean(client, err)}\n\`\`\``
);
 }
};

exports.defaultConfig = {
 aliases: ["eval"],
 info: "used to execute arbitrary code - BE VERY VERY CAREFUL",
 usage: "$eval <valid JS code>",
 enabled: true,
 permReq: ["BOT_OWNER"],
 cooldown: 1000,
 allowedChannels: [],
};
```

4.6 ./COMMANDS/HELP.JS

```
/**
 * small command to list the commands the user can execute, as well as any configure
d aliases, with the option to return more information once the command name is speci
fied.
 * @param {object} client
 * @param {object}
 */
exports.run = (client, message, args) => {
 const Discord = require("discord.js");
 const cembed = new Discord.RichEmbed() //creates a new embed instance
  .setTitle(`${client.user.username}'s List of commands`)

.setAuthor(`${client.user.username}`, `${client.user.displayAvatarURL}`) //adding el
ements
  .setColor("#1a1aff") //this is a dark blue-ish colour
 let clist = [] //list of commands in array format
 if (!args[1]) {
  Object.entries(message.settings.commands).forEach(cmd => {
   if (client.canExecute(client, message, cmd[0]) === "passed") {

//check if the user can run the command - no point telling them if they can't use it
!

clist.push(`\`\`\`\n${cmd[0]}: aliases: ${cmd[1].aliases.filter(alias => alias != cm
d[0]).join(", ") || "none"} ${cmd[1].info ? "Info: " + cmd[1].info : ""}\n\`\`\``)
   }
  })
  cembed.addField("List of Available Commands:", clist)

.setFooter("Use the help command followed by the command you want more information o
n.") //prompt the user as to additional functionality.
  message.author.send({ embed: cembed });


 } else {

if (client.canExecute(client, message, args[1]) === "passed") { //another check as t
his is a different usecase
   let cmd = message.settings.commands[args[1]]

cembed.setTitle(args[1]) //shows usage information as well as any info, both are opt
ional.

.setDescription(`\`\`\`\naliases: ${cmd.aliases}\ninfo: ${cmd.info || "No info"}\nus
age: ${cmd.usage.replace("$", message.settings.prefix) || "no usage information"}\n\
`\`\``)
   message.author.send({ embed: cembed });
```

```
    return
   }
  }
}
exports.defaultConfig = {
 aliases: ["help", "commands"],
 info: "lists the commands the user is able to execute in the current server",
 usage: "$help <command name/alias>",
 enabled: true,
 permReq: [],
 cooldown: 10000,
 allowedChannels: [],
};
```

4.7 ./COMMANDS/RELOAD.JS

```
/**
 * extremely basic command that reloads the command into memory after a change has o
ccurred on disk
 * @param {object} client
 * @param {object} message
 * @param {object} args
 */
exports.run = (client, message, args) => {
 //reloads the provided command
 client.reloadCommand(args[1], message.guild.id)
};
exports.defaultConfig = {
 aliases: ["reload"],
 info: "Reloads a command with the on-disk version.",
 usage: "$reload <commandName>",
 enabled: true,
 permReq: [],
 cooldown: 1000,
 allowedChannels: [],
};
```

4.8 ./COMMANDS/RESTART.JS

```
/**
 * command that signals to the  bot to shutdown.
 * @param {object} client
 * @param {object} message
 * @param {object} args
```

```
 */
exports.run = (client, message, args) => {
 message.react("✅") //reacts to the message as feedback.
 //invokes a restart with the provided reason(s)
 client.emit("gracefulShutdown", `Manual - ${args[1]}`)
};
exports.defaultConfig = {
 aliases: ["restart", "reboot"],
 info: "Reboots the  bot",
 usage: "$restart <reason for restart",
 enabled: true,
 permReq: ["BOT_OWNER"],
 cooldown: 1000,
 allowedChannels: [],
};
```

4.9 ./EVENTS/ATTACHMENTRECORDER.JS

```
/**
 * detects and downloads any attachments present in a message - detects links via th
e automated embed system discord uses.
 * uploads and pipes data to NSFWClassifier (optional). attachments used for when a
message is deleted.
 * @param {object} client
 * @param {object} Message
 */
module.exports = async (client, Message) => {
 var modConfig = Message.settings.modules.attachmentRecorder //gets module config
 //checks state
 if (!modConfig.enabled) return
 //''accumulator'' for attachment links
 var atts = []
 //sets the cache path for downloaded files.
 var path = modConfig.storageDir
 /**
  * downloads all attachments from a given message object - optional piping and stor
ing.
  * @param {object} client
  * @param {object} message
  */
 let getAttachments = (client, message) => {
  //array of URL's
  let toProcess = []
```

```
  //add any attachments URLs
  toProcess.push(...message.attachments.array().map(attachment => attachment.url))
  //add any URLs from media Embeds.

toProcess.push(...(message.embeds.filter(embed => embed.type === "image" || embed.ty
pe === "video")).map(embeds => embeds.url))
  //iterate over each URL
  toProcess.forEach(att => {
   //sets the file name = message.id + a counter

var filename = message.id + "(" + (toProcess.indexOf(att)) + ")" + "." + att.split("
/").pop().split(".")[1];
   //full file path, eg D:/Overlord/cache/445743395557322(0).jpg
   var filePath = path + filename
   //invoke download with almost unlimited timeout.

client.download(att, { directory: path, filename: filename, timeout: 99999999 }, fun
ction (err) {
  if (err) client.log(`download of attachment ${att} failed!`, "ERROR");
  else {
   client.log("download successful!");
   //check if attachments are kept and if NSFWClassifier is enabled.
   if (message.settings.modules.NSFWClassifier.enabled && !modConfig.keep) {
    //pipe data to NSFWClassifier
    client.emit("NSFWClassifier", message, filename);
   }
   //if attachments are configued to be kept..
   if (modConfig.keep) {
    client.log("starting upload...")
    //upload the file to transfer.sh.
    new client.transfer(filePath)
     .upload().then(function (link) {

//only unlink without NSFWClassifier - as NSFWC deletes the files it classifies anyw
ay.
    if (!message.settings.modules.NSFWClassifier.enabled) {
     client.fs.unlink(filePath, (err) => {

if (err) { client.log(err, "ERROR"); } else { client.log("Unlink successful!"); }
     })
    }
    //add the link to atts
    atts.push(link)
```

```
    client.log(`Upload Successful! ${link} `)
    //check that the attachment is the last one to be processed before writing entry
    if (modConfig.keep && atts.length === toProcess.length) {

//sets expiry of the attachments to 14 days, which is the expiry of data on transfer
.sh.

client.DB.set(message.guild.id, { attachments: atts, expiry: (new Date()).setDate((n
ew Date()).getDate() + 14) }, `persistence.attachments.${message.id}`)
    }
    if (message.settings.modules.NSFWClassifier.enabled) {
     //run NSFW classifier if attachments are kept.
     client.emit("NSFWClassifier", message, filename);
    }
     })
   }
  }
   });
  });

 }
 //delay to wait for embeds to be automatically generated.
 setTimeout(getAttachments, 500, client, Message)

};
//default configuration for the module.
module.exports.defaultConfig = {
 enabled: true,
 storageDir: "./cache/",
 keep: true,
 requiredPermissions: ["MANAGE_MESSAGES", "READ_MESSAGE_HISTORY", "VIEW_CHANNEL"]
};
```

4.10 ./EVENTS/AUTOMOD.JS

```
/**
 * runs optional heuristics on messages, such as antispam, charFlood, and CapsFlood.
 * @param {object} client
 * @param {object} message
 */
module.exports = async (client, message) => {
 //declare some aliases for commonly use pieces of data.
 let config = message.settings
```

```
 let modConfig = config.modules.autoMod
 let ASconfig = modConfig.antiSpam
 let member = message.member
 let trecent = client.trecent
 //if the user has one of the 'excluded' roles, ignore them.

if (Array.from(member.roles).filter(role => modConfig.excludedRoles.includes(role)).
size >= 1) { return }
 //if the user is the bot's owner, ignore them.
 if (message.author.id === config.ownerID) { return }
 //prepare action object with common data.
 let action = {
  memberID: message.author.id,
  guildID: message.guild.id,
  type: "action",
  autoRemove: false,
  //autoRemove: ASconfig.autoRemove,
  penalty: ASconfig.penalty
 }
 /**
  * runs antispam heurisitics on a given message object.
  * @param {object} client
  * @param {object} message
  */
 let antiSpam = (client, message) => {
  //check state - if not enabled, do nothing.
  if (!ASconfig.enabled) return

//ensure entry in DB exists for the specified channel. if it exists, return the curr
ent val. if not, set to provided val ([])
  let userCooldown = trecent.ensure(message.guild.id, [], message.channel.id)

//push the member ID to the relevant subObject array to indicate who has been sendin
g messages where.
  //true is used to allow for duplicate pieces of data in the array.
  trecent.push(message.guild.id, member.id, message.channel.id, true)

//after ASconfig.interval Milliseconds, remove a instance of member.id from the trec
ent array for this channel.

setTimeout(() => { trecent.remove(message.guild.id, member.id, message.channel.id) }
, ASconfig.interval)
```

```
//if the number of instances of member.id for this channel exceeds the threshold, th
e user is classified as 'spamming.'

if ((userCooldown.filter((user) => user === member.id)).length >= ASconfig.count) {
   //remove all other instances of member.id from trecent

trecent.set(message.guild.id, userCooldown.filter((user) => user != member.id), mess
age.channel.id)
   //populate action object
   Action = {
  title: "Message spam Violation",

src: `Spam limit broken by user ${message.author} in channel ${message.channel} : [J
ump to message](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `Spam Limit Broken by ${message.author}`
  },
  request: `Removal of content (last ${ASconfig.count} messages)`,
  requestedAction: {
   type: "bulkDelete",
   target: `${message.guild.id}.${message.channel.id}.${message.author.id}`,
   count: ASconfig.count
  },
   }
   client.emit("modActions", { ...action, ...Action })
  } else {

//add XP, using the antispam delay as a restriction/timeout. will be it's own module
 eventually.

client.DB.math(message.guild.id, "+", client.getRandomInt(1, 3), `users.${message.au
thor.id}.xp`)
  }
 }
 /**
  *
checks the %age of a message's contents being the same character vs a configured thr
eshold
  * @param {object} client
  * @param {object} message
  */
 let charFlood = (client, message) => {
```

```
  //check if enabled - if not, do nothing
  if (!modConfig.heuristics.charFlood.enabled) return

//if the perctange of the message being one character is above the threshold, take a
ction

if (chars.filter(char => (char[1] / message.content.length) * 100 >= modConfig.heuri
stics.charFlood.percentLimit).length != 0) {
   //flesh out the action object with properties
   Action = {
  title: "Message Flood Violation",

src: `Flood % limit broken by user ${message.author} in channel ${message.channel} :
 [Jump to message](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `Flood Limit Broken by ${message.author}`
  },
  request: `Removal of content`,
  requestedAction: {
   type: "delete",
   target: `${message.guild.id}.${message.channel.id}.${message.id}`,
  },
   }

client.emit("modActions", { ...action, ...Action }) //pass action to modActions for
processing.
  }
 }
 /**
  * checks the message contents for characters that are in upper case (caps) and det
ermines the message %age that is comprised of capitals
  * @param {object} client
  * @param {object} message
  */
 let caps = (client, message) => {
  //check if enabled - if not, do nothing.
  if (!modConfig.heuristics.caps.enabled) return

//if the total percentage of the message as caps is greater than the threshold, take
 action.

if (((chars.filter(char => char[0] === char[0].toUpperCase()).length) / message.cont
ent.length) * 100 >= modConfig.heuristics.caps.percentLimit) {
```

```
   //if caps% exceeds or equals configured limit, take action.
   Action = {
  title: "Message Caps Flood Violation",

src: `Caps % limit broken by user ${message.author} in channel ${message.channel} :
[Jump to message](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `Caps % limit Broken by ${message.author}`
  },
  request: `Removal of content`,
  requestedAction: {
   type: "delete",
   target: `${message.guild.id}.${message.channel.id}.${message.id}`,
  },
   }

client.emit("modActions", { ...action, ...Action }) //pass action to modActions for
processing.
  }
 }


 antiSpam(client, message)

//if below the ignore threshold, ignore (as smaller messages are much more likely to
 be a false +ve.)
 if (modConfig.heuristics.ignoreBelowLength >= message.content.length) return
 //below is used to check how many instances of a character exist in a string.
 let chars = new Map();

//iterates over every character as a spread string array (spreads the string into in
dividual characters, then makes each char it's own entry in the array)
 [...message.content].forEach(char => {
  //escape control chars for the RegEx to prevent them from messing anythng up
  if (["."
", "/", "\\", "(", ")", "*", "+", "?", "^", "$", "|"].includes(char)) char = "\\" +
char
  //sets the value of 'char' to however many chars there are, or simply none.
  chars.set(char, (((message.content).match(new RegExp(char, "g")) || []).length))
 })
 //convert to an array of entries : [...,[char, count],...].
 chars = Array.from(chars.entries())
 //execute heuristics (caps)
 caps(client, message)
```

```
 //execute heuristics (chars)
 charFlood(client, message)
}

//default configuration - allows for editing of thresholds and punishments
module.exports.defaultConfig = {
 enabled: true,
 bannedWords: [], //stump
 bannedURLs: [],
 excludedRoles: [],
 punishments: {
  mute: {
   start: 5,
   end: 0
  },
  tempBan: {
   start: 10,
   end: 5
  },
  ban: {
   start: 15,
   end: -1 //-1 = infinite duration
  }
 },
 decay: 3,
 heuristics: {
  ignoreBelowLength: 20,
  charFlood: {
   enabled: true,
   percentLimit: 40,
  },
  caps: {
   enabled: true,
   percentLimit: 40,
  }
 },
 antiSpam: {
  enabled: true,
  interval: 3000,
  count: 5,
  penalty: 1,
```

```
  autoRemove: true
 },
 requiredPermissions: ["MANAGE_MESSAGES"]
}
```

4.11 ./EVENTS/GUILDCREATE.JS

```
/**
 * Triggered every time a guild is joined by the bot.
 * @param {object} client
 * @param {object} guild - the guildObject for the guild that has just been joined.
 */
module.exports = (client, guild) => {
 //log that a guild has been joined.
 client.log(`Client has joined guild ${guild.name}!`, "INFO");

//invokes a check for the  bot's DB to initialise the guild in the database for inst
ant usage.
 client.validateGuild(guild)
};
```

4.12 ./EVENTS/GUILDMEMBERADD.JS

```
/**
 * triggered when a user joins (or re-joins) a guild. in the case of a rejoin,
 * the bot will reload a saved user state (roles and nickname) before they left the
guild.
 * @param {object} client
 * @param {object} member - member object for the user that just joined the guild.
 */
module.exports = async (client, member) => {
 //gets config
 let modConfig = client.DB.get(member.guild.id, `modules.guildMemberAdd`)
 //alias for guild
 let guild = member.guild;
 //logs memeber join

client.log(`new member with Name ${member.displayName} has joined ${guild.name}`, "I
NFO");
 //initialisies user to DB
 let data = client.DB.ensure(guild.id, { xp: 0 }, `users.${member.id}`);
 //if a backup of a user state exists, restore the user.
 if (data.savedState) {
  //get state
  let state = data.savedState
```

```
  //set nickname to nickname in state data
  member.setNickname(state.nick)
  //set roles to array of role ID's in state data.
  member.addRoles(state.roles)
  //action object - for notification of action to moderators/admins.
  client.emit("modActions", {
   memberID: member.id,
   guildID: member.guild.id,
   type: "report",
   title: "State restore of re-joining User",
   executor: client.user,
   request: `User ${member} has re-
joined the server, and has had their roles and nickname restored.`,
   trigger: {
  type: "automatic",
   }
  })
 }
 //if the welcome message is enabled
 if (modConfig.enabled) {
  //send it (with some parsing) to the specified Welcome channel

client.guilds.get(guild.id).channels.get(modConfig.welcomeChannel).send((modConfig.w
elcomeMessage)
   .replace("{{user}}", member).replace("{{guildName}}", guild.name))
 }
};
module.exports.defaultConfig = {
 enabled: false,
 welcomeChannel: 0,
 welcomeMessage: "welcome {{user}} to {{guildName}}!"
}
```

4.13 ./EVENTS/GUILDMEMBERREMOVE.JS

```
/**
 * Triggered when a user leaves a guild. the bot will copy their state from just bef
ore they left and save it
 * (roles and nicknames)
 * @param {object} client
 * @param {object} member - object of the member that just left the guild.
 */
module.exports = (client, member) => {
```

```
 //log the fact that a member has left the guild

client.log(`member ${member.displayname} has left guild ${member.guild.name}`, "INFO
")
 //if the module is disabled, do nothing.

if (!client.DB.get(member.guild.id, `modules.guildMemberRemove`).enabled) { return }
 //create state
 let state = {
  nick: member.displayName,
  roles: Array.from(member.roles.keys()),
  TS: new Date()
 }
 //save state
 client.DB.set(member.guild.id, state, `users.${member.id}.savedState`)
}
module.exports.defaultConfig = {
 enabled: true
}
```

4.14 ./EVENTS/GUILDMEMBERUPDATE.JS

```
/**
 * triggered when a member of a guild gets updated, eg they go offline.
 * only really used for debugging.
 * @param {object} client
 * @param {object} oldMem - member object before the update.
 * @param {object} newMem - member object after the update.
 */
module.exports = (client, oldMem, newMem) => {
 //debug stub that logs the change to the member.
 client.log(`member update: ${JSON.stringify(client.diff(oldMem, newMem))}`)
};
```

4.15 ./EVENTS/GUILDUPDATE.JS

```
/**
 * triggered when a guild get updated - eg new channel(s), config changes, etc.
 * only used for debugging right now.
 * @param {object} client
 * @param {object} oldSvr - Object of the server before the update
 * @param {object} newSvr - Object of the server after the update.
 */
module.exports = (client, oldSvr, newSvr) => {
 //debug stub that logs the change to the guild Object.
```

```
client.log(`Guild has been updated: ${JSON.stringify(client.diff(oldSvr, newSvr))}`)
;
};
```

4.16 ./EVENTS/MESSAGE.JS

```
/**
 * primary event - triggered whenever any channel in any guild the bot can 'see' get
s sent a new message.
 * almost all the processing in the bot is based off data from this event - it is th
e main event, so to speak.
 * @param {object} client
 * @param {object} message - Object of the message that has been sent by a user.
 */
module.exports = async (client, message) => {
 //partials are uncached data that can then be resolved into 'full' data structures.

//not currently used due to lower project version (v11 vs V12). once I update this w
ill become very useful.
 if (message.partial) {
  await message.fetch().catch(err => {
   client.log(err, "ERROR")
   return
  })
 }
 //ignores all messages from other bots or from non-
text channels, EG custom 'news' channels in some servers, or storefront pages, etc.
 if (message.author.bot || !message.channel.type == "text") return;

//checks if the bot is currently undergoing a shutdown. if so, interdicts all furthe
r processing.
 if (client.isShuttingDown) {

//reactions as a form of feedback to the user. these indicate 'stop' and 'wait' for
the shutdown.
  message.react("🚫").then(() => { message.react("⌛"); });
  return;
 }
 //guild-only due to increased technical complexity to account for non-
guild scope'd interactions.
 if (message.guild) {

//binds the guild's settings and the level of the user to the message object, for ea
se-of-access for later operations (eg commands)
  message.settings = client.DB.get(message.guild.id)
```

```
//fetches the member into cache if they're offline. important to do this as this can
 be used by functions later on.
  if (!message.member) await message.guild.members.fetch(message.author);

//used to check that the message contains *only* the mention of the bot, and nothing
 else.
  let  botMentionRegEx = new RegExp(`^<?@!?${client.user.id}>?$`);
  //checks if the bot, and *only* the bot, is mentioned.
  if (message.isMentioned(client.user.id) && message.uncleanContent.match(
botMentionRegEx)) {

//sends (DM's) the user the Command Prefix for the guild, or the default prefix if a
nything out of scope happens.

message.author.send(`Hi there,${message.member.displayName}, My prefix in guild ${me
ssage.guild.name} is ${message.settings.prefix || "$"}.\n for help, use the command:
 ${message.settings.prefix || "$"}help `);
   return
  }

//built in method for cleaning message input (eg converting user mentions into a str
ing to prevent issues when returning message content)
  message.content = message.cleanContent;
  //emit events for message heuristics
  client.emit("attachmentRecorder", message)
  client.emit("toxicClassifier", message)
  client.emit("autoMod", message)
  //command processing
  //checks the message starts with the prefix.
  if (message.content.startsWith(message.settings.prefix)) {
   //splits the messages content into an array (space delimited)

let args = (message.content.slice(message.settings.prefix.length).trim().split(/ +/g
))
   //resolve 'true' command name from an alias.

let cmdName = client.DB.get(message.guild.id, `commandsTable.${args[0].toLowerCase()
}`)
   //resolves the commands config using the 'true' name
   let cmdCfg = message.settings.commandsTable[cmdName]

//checks that the user can execute the command (see Functions.js). stores resultant
state as state
   let state = client.canExecute(client, message, cmdName)
   //imports discord.js to construct embeds
```

```
    let Discord = require("Discord.js")
    //creates a new embed instance
    let embed = new Discord.RichEmbed()
   .setAuthor(client.user.username, client.user.avatarURL)
   .setTimestamp(new Date())

//default content that should always be replaced. if not, it's very clear something
is not functioning correctly.

.setTitle("if you are seeing this, something has gone wrong. please inform the bot o
wner.")

.setDescription(`If you think this is a mistake, please contact an Administrator.`)
   .setColor("#FF0000") //red

message.awaitReactions((reaction, user) => { return reaction.emoji.name === "❓" &&
user.id === message.author.id }, { max: 1, time: 6000 })

//restrictions are indicated by reactions - the ❓ reaction can be used to make the
bot send the user the 'key' as to what each reaction means.
   //this code waits for a reaction, checks the reaction 'name' and react-
er before sending the embed to the user.
   //this is the best way I found to provide the user with non-
intrusive feedback for commands, etc.
   .then(collected => {

//can proceed even if no reactions occur in the time frame. added a check to ensure
one is added.
    if (collected.size) {
     message.author.send({ embed: embed })
    }
    //remove the reactions from the message - reduces visual clutter
    message.clearReactions()
   })

//switch:case for modifying the reporting embed's contents according to what the fai
lure is.
    switch (state) {
   //if the command doesn't exist
   case "nonexistant":
    embed.setTitle(`Command/Command Alias ${args[0]} Does not exist.`)

.setDescription(`Please use the \`\`\`\n${message.settings.prefix}Help\n\`\`\` Comma
nd to see all Available Commands.`)
    break
```

```
  //if the command is disabled (guild-wide)
  case "disabled":

embed.setTitle(`Command ${args[0]} has been disabled for guild ${message.guild}.`)
   break
  //if the user lacks the perms required
  case "perms":

embed.setTitle(`You do not have the required permissions to execute command '${args[
0]}' in Server ${message.guild}`)
   break
  //if the channel is not in the allowed array
  case "channel":

embed.setTitle(`Command '${args[0]}' has been disabled in channel ${message.channel}
 in Server ${message.guild}`)
   break
  //if the user is still in the command's cooldown period
  case "cooldown":

embed.setTitle(`Woah There! Please slow down with the usage of Command '${args[0]}'
in Server ${message.guild}`)
   break
  //if the user is in the blacklist for this command - guild-
wide ban of specific command usage
  case "blacklist":

embed.setTitle(`Oh Dear. Looks like you've been blacklisted from using command '${ar
gs[0]}' in Server ${message.guild}`)
   break
  //if the user passes all the checks...
  case "passed":
   //if the user is not an admin, add them to the command's cooldown.
   if (!message.member.permissions.has("ADMINISTRATOR")) {
    client.cooldown.push(message.guild.id, message.member.id, cmdName, true)

//after a set time, remove them from the cooldown, allowing them to use the command
again.

setTimeout(() => { client.cooldown.remove(message.guild.id, message.member.id, cmdNa
me); }, cmdCfg.cooldown)
   }
   //log the fact that a command has been executed.
```

```
client.log(`User ${message.author} executed command ${cmdName} with arguements ${arg
s.join(", ")} in ${message.guild}:${message.channel}`, "INFO")
   //load the command from disk as an object
   let cmdObj = require(`${client.basedir}\\commands\\${cmdName}.js`)
   try {
    //run the command - passing args and the message invoking the command.
    cmdObj.run(client, message, args)
   } catch (err) {
    //on an error, log it, then provide 'feedback' to the user

embed.setTitle(`Oh Dear. It seems that an error occurred whilst trying to execute th
is command.`)

.setDescription(`If this continues to occur, please notify ${client.users.get(client
.config.ownerID)}`)

//react to the message to allow the user to aquire feedback - optional but clear eno
ugh.
    message.react("❌").then(() => { message.react("❓") })
    client.log(`Error with command ${cmdName}`, "ERROR")
    console.log(err)
   }
   break
   }
   //this is the trigger that allows for the embed to be sent to the user.
   if (state != "passed") {
  message.react("🚫").then(() => { message.react("❓") })
  return
   }
  }
 } else {

//Owner universal restriction escape - as they control the bot anyway - useful for d
ebugging.
  //treat everything they send as a command.
  if (message.author.id === client.config.ownerID) {
   let args = (message.content.trim().split(/ +/g)) //split args into an array
   let cmdObj = require(`${client.basedir}\\commands\\${args[0]}.js`)
   try {
  cmdObj.run(client, message, args) //run the command
   } catch (err) {
  console.log(err)
```

```
   }
   return
  } else { //log any messages sent to the bot Via DM's.

client.log(`Message with contents ${message.content} sent to the  bot Via DM's by us
er ${message.author}`, "INFO")
  }
 }
}
```

4.17 ./EVENTS/MESSAGEDELETE.JS

```
/**
 * triggered whenever a message still in the bot's cache is deleted by a user. logs
the deleted message.
 * @param {object} client
 * @param {object} message
 */
module.exports = async (client, message) => {
 //check that the author of the deleted message wasn't a bot.
 if (message.author.bot || !message.guild) return;
 //if disabled, stop execution.
 if (!client.DB.get(message.guild.id).modules.messageDelete.enabled) return
 let entry = await message.guild.fetchAuditLogs({ type: "MESSAGE_DELETE" })
  .then(audit => audit.entries.first());
 //get the audit log entries and find the latest MESSAGE_DELETE entry.
 //- executor is the person who deleted the message in this case
 let executor = ""; //eslint-disable-line
 if (entry != undefined
  && (entry.extra.channel.id === message.channel.id)
  && (entry.target.id === message.author.id)
  && (entry.createdTimestamp > (Date.now() - 15000))
  && (entry.extra.count >= 1)) {
  //if this passes, then the person who deleted the message is the executor.
  //check that the executor is not a bot
  if (executor.bot) { return }
  //notification of discrepancy in author and executor.
  executor = `${entry.executor} - Original Author is ${message.author}`
 } else {
  //if not, we have to assume it was the author of the message.
  executor = message.author
 }
 //action object generated for processing.
```

```
 let action = {
  title: `Message deleted in ${message.channel} by ${executor}.`,
  type: "audit",
  change: "deleted",
  data: message.content,
  attachments: [],
  executor: "",
  guildID: message.guild.id,
 }
 if (message.attachments) {
  try {
   //get the saved attachments (if they exist) from the DB.
   action.attachments = (
  client.DB.get(message.guild.id, `persistence.attachments.${message.id}`)
   .attachments || [])
  } catch (err) {
   client.log("message had no attachments")
  }
 }
 //pass the action object to modActions for processing.
 client.emit("modActions", action)
};

module.exports.defaultConfig = {
 enabled: true,
 requiredPermissions: ["VIEW_AUDIT_LOG"],
}
```

4.18 ./EVENTS/MESSAGEREACTIONADD.JS

```
/**
 * triggered whenever a message is reacted to.
 * mainly used for debug, but will be used for features in the future.
 * @param {object} client
 * @param {object} messageReaction - Object containing the reaction that was added t
o the message
 * @param {object} user - object of the user that added the reaction.
 */
module.exports = async (client, messageReaction, user) => {

//partial resolver - for uncached data. not yet useful due to Overlord running on a
lower API Version.
```

```
 if (messageReaction.partial) {
  await messageReaction.fetch().catch(err => {
   client.log(err, "ERROR")
   return
  })
 }

client.log(`reaction "${messageReaction.emoji.name}:${messageReaction.emoji.identifi
er}" added to message with ID ${messageReaction.message.id} by user ${user.id}`);
 return;
}
```

4.19 ./EVENTS/MESSAGEUPDATE.JS

```
/**
 * triggered when a message is updated
this (annoyingly) includes the addition of automatic embeds.
 * include other useful changes such as when the content of a message changes.
 * automatically sends data to modActions for reporting.
 * @param {object} client
 * @param {object} Message - original message object
 * @param {object} nMessage - New Message object (changed/updated)
 */
module.exports = (client, Message, nMessage) => {
 if (!Message.guild) return //check that the message is in a guild
 if (!client.DB.get(Message.guild.id).modules.messageUpdate.enabled) return
 //check if reporting edits is enabled.
 nMessage.content = nMessage.cleanContent;
 //clean the content
 if (Message.content === nMessage.content) {
  client.log("messageEdit invoked - message content identical - assume autoembed");
  //if contents are identical, autoembed was most likely triggered.
  //autoembeds are created by Discord to show media content in a message.
  return
 } else {
  let action = {//prepare action object
   title: `Message by ${Message.author} edited in ${Message.channel}.`,
   type: "audit",
   change: "edited",
   data: Message.content,
   edit: nMessage.content,
   executor: Message.author,
   guildID: Message.guild.id,
```

```
   memberID: Message.author.id
  }
  client.emit("modActions", action)
 }
}
module.exports.defaultConfig = {
 enabled: true,
 requiredPermissions: ["MANAGE_MESSAGES"]
}
```

4.20 ./EVENTS/MODACTIONS.JS

```
/**
 * processes an 'action' object containing data to perform a specific action through
 various process control measures (eg optional human intervention)
 * @param {object} client "global scope" for the application. manditory.
 * @param {object} action custom object containg pseudo-
standardised data. see the basic schema below.
*/
module.exports = async (client, action) => {
 //re-imported due to needing embeds (can't access through Client Object).
 let Discord = require("Discord.js")
 let guild = client.guilds.get(action.guildID)
 let target = guild.members.get(action.memberID)
 let config = client.DB.get(action.guildID)
 let modAction = guild.channels.get(config.modActionChan)
 let audit = guild.channels.get(config.auditLogChan)
 let modReport = guild.channels.get(config.modReportingChan)
 //if any of the channels are undefined, resets them to the first valid channel.
 if (!modAction || !audit || !modReport) {

//multiAssign as well as a map iterator 'trick' to get the first valid channel witho
ut having to get the key.

modAction = audit = modReport = guild.channels.filter(chan => chan.type === "text").
values().next().value
  //notify administrators (and bot owner) that the channels are undefined.

client.log(`Undefined Reporting channels for guild ${guild.name} - falling back to f
irst valid channel`, "WARN")

modReport.send("WARNING: Reporting channels Have not been configured properly or I d
on't have access to them!")
 }
```

```
 //https://leovoel.github.io/embed-visualizer/ - link used to design embeds.

 /* example of an action:
 {
  guildID: message.guild.id,
  memberID: message.member.id,
  type: "action",
  autoRemove: modConfig.autoRemove,
  title: "Suspected Toxic Content",

src: `Posted by user ${message.author} in channel ${message.channel} : [Jump to mess
age](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `Toxic content breakdown: \n${classi.join(" ")}`,
  },
  request: "Removal of offending content",
  requestedAction: {
   type: "delete",
   target: `${message.guild.id}.${message.channel.id}.${message.id}`,
  },
  penalty: modConfig.penalty,
 }*/

 /**
  * processes an action object to request approval for an action from a moderator.
  * @param {object} client
  * @param {object} action - the action object to be processed as type modAction
  */
 const genModAction = async (client, action) => {

//if the action doesn't require user input, bypass this system and directly execute.

if (action.autoRemove) { actionProcessor(client, { ...action, ...{ executor: client.
user } }) } else {
    //create a new embed instance
    let embed = new Discord.RichEmbed()
  //add elements to the embed
  //set the author to the bot
  .setAuthor(client.user.username, client.user.avatarURL)
  //set the timestamp to now
  .setTimestamp(new Date())
```

```
  //set the title
  .setTitle(`Action requested: ${action.title}`)
  //set the description (main body of text)
  .setDescription(`source: ${action.src}`)
  //add fields for additional information
  .addField("Trigger type", action.trigger.type)
  .addField("Trigger description:", action.trigger.data)
  .addField("Recommended Action Pending Approval:", action.request)
  .setFooter("react with ✅ to perform the reccomended action.")

modAction.send({ embed: embed }).then(msg => { //wait for the message to be sent and
 reacted to.

msg.react("✅").then(() => { //waits for a ✅ reaction, then executes the requested
 action.
   //checks that only one user (not a bot) reacts with a ✅.

msg.awaitReactions((reaction, user) => { return reaction.emoji.name === "✅" && !us
er.bot }, { max: 1 })

.then(collected => { //passes through all results that passed the above filter funct
ion.
     //gets the userID of the user that reacted.

action.executor = client.guilds.get(action.guildID).members.get(Array.from(collected
.get("✅").users)[1][1].id)

action.trigger.type = "Manual" //changes trigger type as it had moderator/human inpu
t.
     actionProcessor(client, action)
    })
  })
   }).catch(err => {
  //catches and logs any errors
  client.log(err, "ERROR")
   })
  }
 }


 /**
  * processes an action object to generate an audit entry - showing that something h
as changed and what that change is.
  * generates and sends an embed so that moderators can monitor any chnages, eg mess
age deletions/edits, etc.
```

```
  * @param {object} client
  * @param {object} action - action object to be processed to produce an auditLog ch
annel entry.
  * (not to be confused with the server's built-in audit log).
  */
 const genAudit = async (client, action) => {
  //create a new embed instance
  let embed = new Discord.RichEmbed() //add attributes
   .setAuthor(client.user.username, client.user.avatarURL)
   .setTimestamp(new Date())
   .setTitle(`Change: ${action.change}`)
   .setDescription(action.title)

//specialised formatting used to generate 'code blocks' - help with contrast and bre
aking text up.
   .addField("Original:", `\`\`\`\n${action.data}\n\`\`\``)

switch (action.change) { //switch:case for specific types of change needing specific
 fields.
   case "deleted":
  //list attachments that were saved and re-uploaded.
  embed.addField("Attachments:", `${action.attachments.join("\n") || "None"}`)
  break;
   case "edited":
  //add a field showing the edited contents.
  embed.addField("Edited Contents:", `\`\`\`\n${action.edit}\n\`\`\``)
  break
  }
  //send the embed to the audit log channel.
  audit.send({ embed: embed }).catch(err => {
   //catch and log any errors
   client.log(err, "ERROR")
  })
 }


 /**
  * processes an action object to create a report of a moderator action - this is su
pposed to be 'public facing'
  * @param {object} client
  * @param {object} action - action to be processed into a report.
  */
 const genModReport = async (client, action) => {
```

```
  //create a new embed - add attributes
  let embed = new Discord.RichEmbed()
   .setAuthor(client.user.username, client.user.avatarURL)
   .setTimestamp(new Date())
   .setTitle(`Action: ${action.title}`)
   .setDescription(`Action description: ${JSON.stringify(action.request)}`)

.addField(`action trigger: ${action.trigger.type}`, `Executor: ${action.executor}`)
  //send embed to the modReport channel.
  modReport.send({ embed: embed }).catch(err => {
   client.log(err, "ERROR")
  })
 }


 /**
  * processes an action object to generate a nortification for a user to notify them
 of an action having taken place.
  * such as being muted or banned.
  * @param {object} client
  * @param {object} action
  */
 const genNotification = (client, action) => {

//this is a promise to ensure that the ban does not get triggered before the user is
 notified.
  return new Promise(resolve => {
   //create new embed instance, add attributes to it.
   let notification = new Discord.RichEmbed()
   notification
  .setTitle(`Notification of action: ${action.request}`)

.setDescription(`In server ${guild}\n - ${action.src}.\n trigger: ${action.trigger.t
ype}, executor: ${action.executor}.`)
  .setFooter("If you wish to dispute this action, please contact an Administrator.")
   target.send({ embed: notification }).then(() => {
  resolve("Sent!")//placeholder value to resolve the promise.
   })
  })
 }


 /**
  * processes an action's requested action after it has been approved.
```

```
  * eg mutes the user, bans the user, deletes some messages.
  * @param {object} client
  * @param {object} action
  */

let actionProcessor = async (client, action) => { //proccessing manager/discriminato
r for actions,

if (action.penalty) { //if the action has a 'penalty' attached to it, fork the objec
t to the scheduler to apply punishments.

//ensure the user has a demerits entry. if they did, return the value, else set to t
he provided object.

let data = client.DB.ensure(guild.id, { count: 0, TS: new Date() }, `users.${action.
memberID}.demerits`)
   //update the user with the new number of demerits and timestamp.

client.DB.set(guild.id, { count: data.count + action.penalty, TS: new Date() }, `use
rs.${action.memberID}.demerits`)
   //schedule the action to happen immeadiatly.

client.schedule(guild.id, { type: "demerit", end: new Date(), memberID: action.membe
rID, })
  }


  /**
   * actions: objects containing data to be done 'at some point', whether via a sche
duler or otherwise.
   * categorised by type, and in this case, by the type of the requested Action.
   */
  switch (action.requestedAction.type) {
   case "delete":
  deleteMessage(client, action)
  break
   case "mute":
  mute(client, action)
  break
   case "ban":
  ban(client, action)
  break
   case "bulkDelete":
  bulkDelete(client, action)
  break
```

```
    default:
  client.log(`unknown/invalid action type: ${action.requestedAction.type}`, "WARN")

  }

  /**
   * given a target and a guild, mutes the target in the guild, notifies, and then r
eports.
   * @param {*} client
   * @param {*} action
   */

function mute(client, action) { //given a target, mutes then notifies the target of
the action.
   target.addRole(config.mutedRole) //gives the target the specified muted role.
   genNotification(client, action)
   genModReport(client, action)
  }

  /**
   * given a target message in 'full path' notation, deletes the message, notifies t
hen reports.
   * @param {*} client
   * @param {*} action
   */
  function deleteMessage(client, action) {
   let args = action.requestedAction.target.split(".")

//deletes the specified message using it's "Full Path" (guildID.channelID.messageID)
   client.guilds.get(args[0]).channels.get(args[1]).messages.get(args[2]).delete()
   genNotification(client, action)
   genModReport(client, action)

  }

  /**
   * given a target user, checks they can be banned before notifying, banning and th
en reporting.
   * notifying before it does so as it cannot DM banned users.
   * @param {object} client
   * @param {object} action
   */
```

```
 async function ban(client, action) {
  //check the both can actually ban the user
  if (!target.bannable) {
 client.log(`Cannot Ban user ${target} - I do not have the permissions!`, "WARN")
 return
  } else {
 try { //catch in case the user has already been banned or some other error occurs.
  //generate and send the notification of action before banning the user.
  genNotification(client, action).then(() => {
   //ban the user
   target.ban({ reason: action.title })
  })
  //genreate a moderation report
  genModReport(client, action)
 } catch (err) {
  client.log(err, "ERROR")
 }

  }


 }


 /**
  * given a user as well as a channel, and a count, deletes the last count messages
by that user in the specified channel.
  * where count is action.requestedAction.count, then notifies and reports.
  * @param {object} client
  * @param {object} action
  */
 async function bulkDelete(client, action) {
  //get the channel
  let channel = guild.channels.get(action.requestedAction.target.split(".")[1])
  //wait for the messages to be fetched and filtered by authorID

let toDelete = await ((channel.fetchMessages().then(messages => messages.filter(m =>
 m.author.id === target.id))))
  //deletes all passed messages up to count.
  channel.bulkDelete(toDelete.array().splice(0, action.requestedAction.count))
  //generate a notification and modReport
  genNotification(client, action)
  genModReport(client, action)
```

```
  }
 }


switch (action.type) { //switch:case for flow control to descriminate against differ
ent types of actions
  case "action":
   genModAction(client, action)
   break;
  case "audit":
   genAudit(client, action)
   break;
  case "report":
   genModReport(client, action)
   break;
  case "actionProcessor":
   actionProcessor(client, action)
   break;
  default:

client.log(`incorrect action type ${action.type}! event processing failed.`, "ERROR"
)
   break;
 }


}
module.exports.defaultConfig = {
 requiredPermissions: ["MANAGE_MEMBERS", "MANAGE_GUILD"],
}
```

4.21 ./EVENTS/NSFWCLASSIFIER.JS

```
/**
 * given a filename, runs it through the NSFWModel to determine if it
 * is above the allowed thresholds for each classification of content.
 * eg above 90% certainty of being Pornographic, before triggering
 * moderation intervention.
 * @param {object} client
 * @param {object} message
 * @param {string} filename - name of the file within the cache directory.
 */
module.exports = async (client, message, filename) => {
```

```
 //checks that the models are enabled bot-wide.
 if (!client.NSFWModel) return
 //if the channel has the NSFW tag, disable the classification.
 if (message.channel.nsfw) return
 let cacheDir = message.settings.modules.attachmentRecorder.storageDir
 let modConfig = message.settings.modules.NSFWClassifier
 //if this module is disabled in the guild, return.
 if (!modConfig.enabled) return
 var classifier = async (client, img) => {
  return new Promise(resolve => {
   try {
  //read the image into a buffer
  const imageBuffer = client.fs.readFileSync(cacheDir + img);

//proccess the imageBuffer as a 3d tensor (x pixels, y pixels, z pixel depth (eg col
ours))
  const image = client.tf.node.decodeImage(imageBuffer, 3, undefined, false);
  //pass the processed data to the neural network for classification.
  client.NSFWModel.classify(image).then(predictions => {
   //resolve the promise once classification completes.
   resolve(predictions)
   //unlink (delete) the processed file from the cacheDir
   client.fs.unlink(cacheDir + filename, (err) => {

if (err) { client.log(err, "ERROR"); } else { client.log("Unlink successful!"); }
   })
   return
  });
   } catch (err) {
  console.error(err);
   }
  });
 };
 /**
  * framework built around the classifier to pipe and parse results into useable inf
ormation.
  */
 classifier(client, filename).then(predictions => {
  //logs predictions - debug

client.log(`file ${filename} has results ${JSON.stringify(predictions.join(","))}`)
  //remove SFW classes as they're not helpful.
```

```
predictions = predictions.filter(p => p.className != "Neutral" || p.className != "Dr
awing")

//if the number of predictions whose certainty that exceed their weighting exceeds t
he thresholdExceeders value,

if (predictions.filter(p => p.probability >= modConfig.classificationWeights[p.class
Name]).length >= modConfig.thresholdExceeders) {
   //map to form formatted strings

predictions.map(p => `${p.className} Certainty: ${Math.round(p.probability * 100)}%\
n`)
   var action = { //action object for modActions. add attributes
  guildID: message.guild.id,
  memberID: message.member.id,
  type: "action",
  autoRemove: modConfig.autoRemove,
  title: "Suspected NSFW Content",

src: `Posted by user ${message.author} in channel ${message.channel} : [Jump to mess
age](${message.url})`,
  trigger: {
   type: "Automatic",
   data: `NSFW content breakdown: \n${predictions.join(" ")}`,
  },
  request: "Removal of offending content",
  requestedAction: {
   type: "delete",
   target: `${message.guild.id}.${message.channel.id}.${message.id}`,
  },
  penalty: modConfig.penalty,
   }
   //send to modActions.
   client.emit("modActions", action)


  }
 })
};
//default configuration - this specifies that
//content classed as x has to have a certainty at or over y*10 %
module.exports.defaultConfig = {
 enabled: true,
 classificationWeights: {
```

```
  Hentai: 0.6,
  Porn: 0.7,
  Sexy: 0.8,
 },
 thresholdExceeders: 1,
 autoRemove: false,
 penalty: 5,
 requiredPermissions: ["MANAGE_MESSAGES"],
};
```

4.22 ./EVENTS/SCHEDULER.JS

```
/**
 * givens a min, max, and value to test, determines if a value is in range or not.
 * @param {number} x value that you want to check
 * @param {number} min the minimum of the range
 * @param {number} max the maximum of the range
 */
function inRange(x, min, max) {
 return ((x - min) * (x - max) < 0);
}
/**
 * processes actions that need to be executed by the scheduler, such as punishments,
 role removals, reminders, etc.
 * @param {object} client
 * @param {string} guildID
 * @param {object} action - object containing data for processing.
 */
let actionProcessor = async (client, guildID, action) => {
 let guild = client.guilds.get(guildID)
 let target = guild.members.get(action.memberID)
 //switch:case discrimnator for each action type
 switch (action.type) {
  case "roleRemove": //removes a given role from a given user
   target.removeRole(action.roleID)
   break
  case "reminder": //sends a given user a given string as a reminder.
   target.send(`scheduled reminder: ${action.message}`)
   break
  case "nick": //sets the nickname of a given user to a given string
   target.setNickname(action.nick)
   break
  case "roleAdd": //add a given role to a given user
```

```
   target.addRole(action.roleID)
   break;
  case "unban": //unbans a given user
   guild.unban(action.memberID)
   break;
  case "demerit":
   let mConf = client.DB.get(guild.id).modules.autoMod

let data = client.DB.ensure(guildID, { TS: new Date(), count: 0 } `users.${action.me
mberID}.demerits`)

//determine how many decay cycles have occured between this execution and last execu
tion.

let intervals = Math.max(Math.floor((new Date() - data.TS) / (3600000 * mConf.decay)
), 0)

//determine the new demerit count after the number of cycles, bounding it to be >=0.

//this system was made so that perminent punishments (eg permbans) can be given an e
nd of -1 to be infinite.
   let afterDecay = Math.max((data.count - intervals), 0)
   let pAction = {
  type: "actionProcessor",
  memberID: action.memberID,
  guildID: guildID,
  trigger: {
   type: "Automatic"
  },
  src: "Automated punishment due to reaching demerit threshold.",
  executor: client.user
   } //punishment action - variant of action
   //iterate over all the punishment tiers
   Object.entries(mConf.punishments).forEach(pt => {
  //if it was originally in range (count, pt[1]-start-
end), but is no longer (afterdecay) in range, undo the action/punishment.

if (inRange(data.count, pt[1].end, pt[1].start) && !inRange(afterDecay, pt[1].end, p
t[1].start)) {

client.log(`Removing Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
   switch (pt[0]) {
    case "mute":
```

```
actionProcessor(client, guildID, { memberID: action.memberID, type: "roleRemove", ro
leID: guild.mutedRole })
     break;
    case "tempBan":
     actionProcessor(client, guildID, { memberID: action.memberID, type: "unban" })
     break;
   }
   //if it wasn't in range and is now above the start, apply the punishment.

} else if (!inRange(data.count, pt[1].end, pt[1].start) && afterDecay > pt[1].start)
 {

client.log(`Applying Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
   //switch:case for the name of the punishment
   switch (pt[0]) {
    case "mute":
     client.emit("modActions", {
    ...pAction, ...{
     title: "Mute of User",
     requestedAction: {
      type: "mute"
     },
     request: `Mute of user ${target.tag}`,
    }
     })
     break;
    case "tempBan":
     client.emit("modActions", {
    ...pAction, ...{
     title: "Temporary ban of User",
     requestedAction: {
      type: "ban"
     },
     request: `Temporary ban of user ${target.tag}`,
    }
     })
     break;
    case "ban":
     client.emit("modActions", {
    ...pAction, ...{
```

```
      title: `Ban of User ${target.tag}`,
      requestedAction: {
       type: "ban"
      },
      request: "Ban of user",
     }
      })
      break;
    }
   }
   if (afterDecay <= 0) { //if the user is out of demerits, delete the property.
    client.DB.remove(guildID, `users.${action.memberID}.demerits`)
   } else {

//set the demerits property of the user to a modified version, with the new counts a
nd Timestamp.

client.DB.set(guildID, { count: afterDecay, TS: new Date() }, `users.${action.member
ID}.demerits`)
   }
    })


    action.end = new Date().setTime(new Date().getTime() + (mConf.decay * 3600001))

//reschedule the action, after filterrng out any duplicate entries (as these contain
 non-essential data)

client.DB.set(guildID, client.DB.get(guildID, "persistence.time").filter(act => act.
memberID !== action.memberID), "persistence.time")
    //re-schedule the action.
    client.schedule(guildID, action)
    break
   default:
    client.log(`Unknown action type/action ${JSON.stringify(action)}`, "WARN")
    break;
  }
}
/**
 * custom scheduler engine designed to have a greatly reduced memory footprint in la
rger deployments compared to the built-in scheduler.
 * @param {object} client
 * @param {string} guildID - the ID of the guild that the scheduler should check for
 tasks.
 */
```

```
module.exports = async function check(client, guildID) {
 /** Scheduler - uses setTimeout to call itself
  * array of objects (
  * "action:"{end:TS,type:typeInst,...data}
  */
 let timeouts = client.timeouts //alias for tiemouts.

if (!timeouts.has(guildID)) timeouts.set(guildID, null) //ensure the guild exists in
 timeouts.
 let timeout = timeouts.get(guildID) //get timeout for this guild

client.log(`Checking... (timeout = ${timeout}) ${client.guilds.get(guildID).name} `)
; //notification that the scheduler is operating
 // clears previous check refresher
 clearTimeout(timeout);
 const now = new Date()
 //gets persistence data from the DB
 let data = client.DB.get(guildID, "persistence.time")
 if (data.length == 0) return
 //finds the shortest delta between now and the end of all the actions.

const closest = Math.min(...data.filter(action => action.end >= now).map(action => a
ction.end));
 //executes all the actions that should've been executed
- eg if the bot crashes. this allows it to catch back up, so to speak.
 data.filter(action => action.end <= now).forEach(action => {
  actionProcessor(client, guildID, action)
 })

client.DB.set(guildID, data.filter(action => action.end >= now), "persistence.time")

if (closest === Infinity) return; //if the number is infinity then there are no pend
ing actions.
 const timeTo = closest - now;
 client.log(`checking timeout in ${timeTo} ms`)
 // will only wait a max of 2**31 - 1 because setTimeout breaks after that

timeouts.set(guildID, setTimeout(check, Math.min(timeTo, 2 ** 31 - 1), client, guild
ID))
};
module.exports.defaultConfig = {
 requiredPermissions: ["MANAGE_MESSAGES", "MANAGE_ROLES", "MANAGE_NICKNAMES"]
}
```

4.23 ./EVENTS/TOXICCLASSIFIER.JS

```
/**
 * givens a min, max, and value to test, determines if a value is in range or not.
 * @param {number} x value that you want to check
 * @param {number} min the minimum of the range
 * @param {number} max the maximum of the range
 */
function inRange(x, min, max) {
 return ((x - min) * (x - max) < 0);
}
/**
 * processes actions that need to be executed by the scheduler, such as punishments,
 role removals, reminders, etc.
 * @param {object} client
 * @param {string} guildID
 * @param {object} action - object containing data for processing.
 */
let actionProcessor = async (client, guildID, action) => {
 let guild = client.guilds.get(guildID)
 let target = guild.members.get(action.memberID)
 //switch:case discrimnator for each action type
 switch (action.type) {
  case "roleRemove": //removes a given role from a given user
   target.removeRole(action.roleID)
   break
  case "reminder": //sends a given user a given string as a reminder.
   target.send(`scheduled reminder: ${action.message}`)
   break
  case "nick": //sets the nickname of a given user to a given string
   target.setNickname(action.nick)
   break
  case "roleAdd": //add a given role to a given user
   target.addRole(action.roleID)
   break;
  case "unban": //unbans a given user
   guild.unban(action.memberID)
   break;
  case "demerit":
   let mConf = client.DB.get(guild.id).modules.autoMod

let data = client.DB.ensure(guildID, { TS: new Date(), count: 0 } `users.${action.me
mberID}.demerits`)
```

```
//determine how many decay cycles have occured between this execution and last execu
tion.

let intervals = Math.max(Math.floor((new Date() - data.TS) / (3600000 * mConf.decay)
), 0)

//determine the new demerit count after the number of cycles, bounding it to be >=0.

//this system was made so that perminent punishments (eg permbans) can be given an e
nd of -1 to be infinite.
   let afterDecay = Math.max((data.count - intervals), 0)
   let pAction = {
  type: "actionProcessor",
  memberID: action.memberID,
  guildID: guildID,
  trigger: {
   type: "Automatic"
  },
  src: "Automated punishment due to reaching demerit threshold.",
  executor: client.user
   } //punishment action - variant of action
   //iterate over all the punishment tiers
   Object.entries(mConf.punishments).forEach(pt => {
  //if it was originally in range (count, pt[1]-start-
end), but is no longer (afterdecay) in range, undo the action/punishment.

if (inRange(data.count, pt[1].end, pt[1].start) && !inRange(afterDecay, pt[1].end, p
t[1].start)) {

client.log(`Removing Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
   switch (pt[0]) {
    case "mute":

actionProcessor(client, guildID, { memberID: action.memberID, type: "roleRemove", ro
leID: guild.mutedRole })
     break;
    case "tempBan":
     actionProcessor(client, guildID, { memberID: action.memberID, type: "unban" })
     break;
   }
   //if it wasn't in range and is now above the start, apply the punishment.
```

```
} else if (!inRange(data.count, pt[1].end, pt[1].start) && afterDecay > pt[1].start)
 {

client.log(`Applying Punishment "${pt[0]}" to user ${client.users.get(action.memberI
D).tag}.`)
   //switch:case for the name of the punishment
   switch (pt[0]) {
    case "mute":
     client.emit("modActions", {
    ...pAction, ...{
     title: "Mute of User",
     requestedAction: {
      type: "mute"
     },
     request: `Mute of user ${target.tag}`,
    }
     })
     break;
    case "tempBan":
     client.emit("modActions", {
    ...pAction, ...{
     title: "Temporary ban of User",
     requestedAction: {
      type: "ban"
     },
     request: `Temporary ban of user ${target.tag}`,
    }
     })
     break;
    case "ban":
     client.emit("modActions", {
    ...pAction, ...{
     title: `Ban of User ${target.tag}`,
     requestedAction: {
      type: "ban"
     },
     request: "Ban of user",
    }
     })
     break;
   }
```

```
  }
  if (afterDecay <= 0) { //if the user is out of demerits, delete the property.
   client.DB.remove(guildID, `users.${action.memberID}.demerits`)
  } else {

//set the demerits property of the user to a modified version, with the new counts a
nd Timestamp.

client.DB.set(guildID, { count: afterDecay, TS: new Date() }, `users.${action.member
ID}.demerits`)
  }
   })


   action.end = new Date().setTime(new Date().getTime() + (mConf.decay * 3600001))

//reschedule the action, after filterrng out any duplicate entries (as these contain
 non-essential data)

client.DB.set(guildID, client.DB.get(guildID, "persistence.time").filter(act => act.
memberID !== action.memberID), "persistence.time")
   //re-schedule the action.
   client.schedule(guildID, action)
   break
  default:
   client.log(`Unknown action type/action ${JSON.stringify(action)}`, "WARN")
   break;
 }
}
/**
 * custom scheduler engine designed to have a greatly reduced memory footprint in la
rger deployments compared to the built-in scheduler.
 * @param {object} client
 * @param {string} guildID - the ID of the guild that the scheduler should check for
 tasks.
 */
module.exports = async function check(client, guildID) {
 /** Scheduler - uses setTimeout to call itself
  * array of objects (
  * "action:"{end:TS,type:typeInst,...data}
  */
 let timeouts = client.timeouts //alias for tiemouts.

if (!timeouts.has(guildID)) timeouts.set(guildID, null) //ensure the guild exists in
 timeouts.
```

```
 let timeout = timeouts.get(guildID) //get timeout for this guild

client.log(`Checking... (timeout = ${timeout}) ${client.guilds.get(guildID).name} `)
; //notification that the scheduler is operating
 // clears previous check refresher
 clearTimeout(timeout);
 const now = new Date()
 //gets persistence data from the DB
 let data = client.DB.get(guildID, "persistence.time")
 if (data.length == 0) return
 //finds the shortest delta between now and the end of all the actions.

const closest = Math.min(...data.filter(action => action.end >= now).map(action => a
ction.end));
 //executes all the actions that should've been executed
- eg if the bot crashes. this allows it to catch back up, so to speak.
 data.filter(action => action.end <= now).forEach(action => {
  actionProcessor(client, guildID, action)
 })

client.DB.set(guildID, data.filter(action => action.end >= now), "persistence.time")

if (closest === Infinity) return; //if the number is infinity then there are no pend
ing actions.
 const timeTo = closest - now;
 client.log(`checking timeout in ${timeTo} ms`)
 // will only wait a max of 2**31 - 1 because setTimeout breaks after that

timeouts.set(guildID, setTimeout(check, Math.min(timeTo, 2 ** 31 - 1), client, guild
ID))
};
module.exports.defaultConfig = {
 requiredPermissions: ["MANAGE_MESSAGES", "MANAGE_ROLES", "MANAGE_NICKNAMES"]
}
```

4.24 REQUIREMENTS/DEPENDENCIES

NPM Modules required:

```
"@tensorflow-models/toxicity": "^1.2.1",
"@tensorflow/tfjs-node": "^1.3.2",
"better-sqlite-pool": "^0.2.2",
"deep-object-diff": "^1.1.0",
"discord.js": "^11.5.1",
 "download-file": "^0.1.5",
 "enmap": "^4.8.7",
```

```
 "nsfwjs": "^2.1.0",
 "transfer-sh": "^2.1.1",
 "zlib-sync": "^0.1.6" - Optional
NPM i PM2 - g
```

ENMAP installation requires the following prerequisites:

Windows: npm i -g --add-python-to-path --vs2015 --production windows-build-tools

Linux - sudo apt-get install build-essential

Followed by:

npm i better-sqlite3

For utilising the Neural Networks, both models need to be downloaded and extracted to their respective directories.

NSFW: https://s3.amazonaws.com/nsfwdetector/nsfwjs.zip

Toxic: https://tfhub.dev/tensorflow/tfjs-model/toxicity/1/default/1?tfjs-format=compressed

-o0o-