**AI-Driven**

**Destiny 2 Coach**

**(Vanguard Mentor)**

Author:

Declan Roy Alan Wadsworth (200061043)

[drw8@aber.ac.uk](mailto:drw8@aber.ac.uk)

Supervisor:

Dr Chuan Lu

[cul@aber.ac.uk](mailto:cul@aber.ac.uk)

Creation Date:

25/04/2025

Version:

1. (Release)

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Wales

United Kingdom of Great Britain and Northern Ireland

**Declaration of Originality**

I confirm that:

* This submission is my own work, except where clearly indicated.
* I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
* I have read the regulations on Unacceptable Academic Practice from the University’s Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
* In submitting this work, I understand and agree to abide by the University’s regulations governing these issues.

Name: Declan Roy Alan Wadsworth

Date: 28/04/2025

**Acknowledgements**

Thank you to my supervisor Chuan Lu, not only for her help in generating a project problem that was relevant and interesting, but for the weekly meetings that kept me accountable and on track, as well as for all her suggestions and guidance throughout the process. This project would look very different without her input.

Thank you to Neil Taylor, for his support throughout the year and his guidance for this module.

**Abstract**

This project consists of a prototype client-server application, with a browser-based client that accesses a remote REST API server, handling all AI reasoning logic and security flows. It is designed for adherence to industry guidelines and standards for constructing RESTful APIs, as well as integrating the use of the OpenAPI[1] specifications for documenting such and ensuring endpoint consistency.

The API server itself utilizes Node.js for its non-blocking I/O and asynchronousity; making the processing of endpoint requests and requests to other APIs fast, and highly scalable for a production environment. The server consists of a modular separation of the services provided; these services being:

* **Authorisation:** Server authorisation makes use of Bungie`s OAuth2 flow, providing the client with tokens that the server encrypts before returning. A client utilises these tokens to access protected endpoints on the prototype API, as well as for re-authorisation on the server in the case of imminent token expiry. Tokens are stored on a database for persistent storage.
* **Coach:** The coach service combines data fetched from the Bungie API and parsed by separate server modules, with the LLM technologies offered by Google through the Google Gemini JavaScript API, utilizing the new Gemini Flash 2.0 model for the best reasoning and analysis abilities, balanced with the need for no-cost API request limits. The coach also provides progress tracking data, fetched from, and stored on a server-side MySQL database.
* **Bungie:** The API provides endpoints for receiving JSON data related to the game. This includes data such as a players recent activities; for making requests to the coaching service that require such, as well as a manually generated knowledge base; consisting of key terminology from the game separated as terminology type by object property.

The prototype client makes the necessary HTTP requests to the API for each coaching tool it provides, filling in any necessary path parameters and header content, and displaying the loaded resource for user consumption. This client serves as an example of the capabilities of the service as a web-application, as well as forming a foundation to test the API in real use-case scenarios.

**Figures**

Figure 1 – Architecture of a client and remote RESTful API…………………………………………………12

Figure 2 – Bungie API Response example………………………………………………………………………..17

Figure 3 – Bungie module initialisation flow diagram…………………………………………………………18

Figure 4 – Example reasoner response schema………………………………………………………………..20

Figure 5 – Example reasoner prompt………………………………………………………………………………20

Figure 6 – Entity Relationship diagram for persistent storage method………………………………….26

**Tables**

Table 1 – System Testing table……………………………………………………………………………………….34

1. **Background, Analysis and Process**
   1. **Background**

In 2014, the games studio Bungie (responsible for early iterations of the Halo games), released a live-service RPG title taking place in a sci-fi setting, with complex character and activity configuration mechanics, known as Destiny. Since the initial release of this title, a sequel in a similar vein followed, as well as numerous major updates to the content and mechanics that existed therein. The current iteration of Destiny 2 has now been running for nearly 8 years.

This continual growth to a relatively old game has however, made learning the plethora of mechanics and play aspects a strenuous, sometimes insurmountable task for new players.

* + 1. **Background – Game Aspects**

As mentioned, the game consists of many different character and activity mechanics, accompanied by an overabundance of terminology to become familiar with. This includes the following foremost properties, of which more detail can be found in Appendix B:

* **Weapons:** What weapons a player has equipped, and how their properties affect gameplay
* **Armours:** The stat rolls of an armour piece and any intrinsic traits
* **Subclass Configuration:** The abilities a configuration provides a player in combat, what skills are focused on in the choices such as damage output, survivability, etc.
* **Character Class Abilities:** What abilities for subclasses does this character have, how do they affect gameplay.
* **Activities, Roles and Modifiers:** Activities feature different kinds of environments and difficulty tiers that affect a players weapon choices. Further to their choices, some activities can have modifiers that put further strain on how a player configures their character. Some activities also include mechanics which require multiple players to communicate and take up key roles, which again, puts further strain on wider character configuration.
  + 1. **Background – Potential**

The degree scheme this report gives completion credit towards requires the project to have an element of artificial intelligence embedded. An option that presented itself through the suggestion of the supervisor, was to leverage emerging LLM technologies, and combine them with relevant data of some kind to provide a clear function.

With this in mind, integrating this type of technology with its reasoning and analysis capabilities, into a tool which could combat the steep learning curve of the game, seemed to not only be an interesting proposition for someone already deeply invested in the game, but one that would be relevant to the games wider community; and potentially add itself to the list of tools adopted community wide. This integration could potentially provide a number of different functions such as:

* Suggest character builds for specific activities
* Analyse post-activity reports to give feedback
* Analyse weapon statistics for potential weaknesses or lack of wider spreading skills between weapon types

With a prototype, the potential of LLM technologies as a form of games coach could be informally investigated, as well as provide a strong foundation to extend and refine its capabilities- outside of the scope of this project - for a full community release.

**1.2 Problem Analysis**

Analysing what aspects of Destiny 2 the project could target was the first aspect to tackle. Considerations had to be made about what coaching functionality could be implemented within the project time-limit, and what capabilities would be appropriate for the prototype, as to showcase the potential of the project past the current scope.

This involved investigating the method for retrieving player data, the processes this retrieval utilized, as well as if this data could be added into some form of pre-structured prompts, or if some pre-processing of data was required.

The selection of an LLM would also be a core consideration, involving comparisons on API rate limits and reasoning capabilities.

Finally, the prototype would have to highly modular and easily extensible, for further development past the prototype stage, outside of the project scope.

**1.2.1 Analysis – Game Data**

The only viable source of player data is straight from the source; utilizing the API provided by Bungie themselves[2]. This is the source of data for all other community tools. The Bungie API is a RESTful API, meaning HTTP requests are made to specific endpoints on the API to retrieve certain data; such as player items, combat statistics and so on. The data returned from the API however, is mostly made up of internal hashes and Enums, mapping to specific static data that can be requested from a manifest. Pre-processing of some kind would have to occur on the retrieved data before it could be combined with any LLM technology. Not only pre-processing, but the server-side of the prototype would have to either request extra data from the API to translate hashes into meaningful identifiers, or locally store maps of these hashes and the data they translate to.

For a web-application, processing requests as quickly as possible is paramount, so that the server-side does not become encumbered under many requests from different client instances. A player can have upwards of 700 items spread across characters and in their vault, as well as an extensive activity history. Making extra requests for each and every item to translate their properties into meaningful identifiers for any potential LLM integration would be expensive. Storing them locally would remove this problem, but would propose its own issues in the likelihood of concurrent requests attempting to access a global store of these maps - leading to race conditions.

* + 1. **Analysis – Prototype Architecture**

As seen in appendix C, there exists a wide range of community developed tools that have been developed and evolved over the near 11 year history of the franchise. Their accessibility is one of the core reasons as to why they have been so widely adopted. Appendix D investigates the potential platforms for which the project could develop a prototype for, and takes into consideration the successfulness of these existing tools and the platforms they exist on. Other platforms were investigated such as native-running applications for different device environments, but the result of these investigations made it apparent that reaching as wide an audience as is possible, would only be viable through a web-application.

Its responsive design elements means multiple different device types (desktop, mobile, etc.) can be targeted in a single application, providing the best chance of community adoption past the project scope.

An additional piece of analysis is required here, as to what architecture the server should be modelled on. Several options exist here, such as a simple front-end and back-end architecture that most websites utilize, having all core software components and computations existing on a back-end server. This would require not only hosting of a client, but a continually operating server also. Some tools exist here, such as Amazon Web Services[3] or Microsoft Azure[4]. These options include free-tiers but are incredibly limited in that limit, as are most other hosting sites. An argument could be made here that as the project focuses on a prototype, a permanent solution for hosting is not required. However, this would increase the amount of work required past the project scope to evolve the system to a place of viability.

One other option is utilizing static hosting services for an SPA (Single Page Application) client such as the widely used GitHub Pages[5] service, that will make simple HTTP requests for authorisation and coaching data to a remote, RESTful API. This would allow the server to be easily hosted on a secured local machine and network, with the client being a lightweight example (due to GitHub Pages 1gb repository limit) of the applications capabilities. Not only this, but a remote RESTful API could potentially aid in community adoption past the project scope, allowing other community members to utilize its functionality; developing their own clients and utilizing other technologies to conceivably extend the applications abilities. Being RESTful would make the API much easier to interact with by any external developer, especially if adherence to industry standards for this technology (such as the OpenAPI Specification[1]) occurs.

* + 1. **Analysis – Large Language Models**

As seen in appendix E, a small-scale investigation was conducted to choose the most appropriate chatbot service for this project. The project does not have funding to pay for any services, nor would such be appropriate for a prototype system. Therefore, a balance would be required between an acceptable “free-tier” of any API requests, and the reasoning/analysis capabilities of the chatbot.

Another problem with the integration of LLMs exists. Most simply output text, which would be hard to parse into any key response components and require extra software to handle such. Having dedicated response properties would make any analysis displayed in a client more digestible to the target audience. With the complexity of the game stated, simply outputting large pieces of text into a client would be counter-productive to the core aim of the prototype.

The LLM technology would also have to be reliably repetitive in some of its responses, where determinism is preferred in the model output, such as where evaluation or analysis of a players skill is required. Many chatbot services provide an option known as “temperature”, which is usually a closed interval between 0 and 1, with 0 being completely deterministic. This would provide the functionality in the LLM that would be required for certain coaching ability, as well as for design decisions where such determinism is counter-productive (such as where the focus of a tool is a players increasing familiarity with core game concepts, and not specific analysis).

* + 1. **Analysis – Coaching Aspects**

With all other analysis completed, determining what coaching tools the prototype could provide was easier. For a prototype with a time-limit, it was best to identify the learning difficulties of the game core components - as specified in appendix B - and what parsing capabilities of Bungie API responses could be implemented in that time.

* 1. **Objectives**

This section covers all objectives and requirements the project must meet.

* + 1. **Coaching Objectives**

The prototype should provide the below coaching tools to aid in a players adoption of key game knowledge, concepts and mechanics; as well as improvement of a players skills:

1. The prototype will provide a character build for a specific activity
   1. The user will select their character and the activity
   2. The prototype will fetch all a players items with meaningful identifiers
   3. The prototype will fetch the activity and any modifiers with meaningful identifiers
   4. The prototype will combine this data with pre-structured prompts to produce structured output.
2. The prototype will analyse a players weapon statistics for each type, and suggest weapon types to improve skills with based off distribution of use, as well as precision of each type
   1. The prototype will fetch the players weapon statistics
   2. The prototype will combine this data with pre-structured chatbot prompts to produce structured output.
3. The prototype will analyse one of the players current character configurations, analysing and explaining key aspects and interactions between sub-components
   1. The user will select one of their characters for the prototype to analyse
   2. The prototype will fetch the current configuration of this character
   3. The prototype will combine this data with pre-structured chatbot prompts to produce structured output.
4. The prototype will analyse an activity report for which a player took part in, providing feedback on the players performance and any critique.
   1. The user will select one of their most recent activities
   2. The prototype will fetch and parse the activities PGCR
   3. The will combine this data with pre-structured chatbot prompts to produce structured output.
5. The prototype will provide a knowledge base to which a player can continually refer to, capturing all key terminology a player must be familiar with in the game.
   1. The user will select a keyword from a manually constructed knowledge base separated by type.
   2. The prototype will combine this keyword and its type with a pre-structured prompt, to produce a structured output specific to the type of keyword.
6. The prototype will analyse the entirety of a users activity history, providing a higher-level analysis of a users general activity skills, providing general feedback as well suggestions of activities and activity types to become more familiar with.
   1. The prototype will fetch the players entire activity history
   2. The prototype will combine this data with pre-structured chatbot prompts to produce structured output.
7. The prototype will analyse a players weapon, player versus player and player versus enemy statistics; providing a set of targets that could help boost a players skills.
   * 1. **Server Functional Objectives**

The prototype will handle all required API requests, coaching logic, as well as security within a RESTful API server; a decision reached through analysis of potential architectures and their viability in the project scope, and past such. This API will have the following functional requirements:

1. Provide all coaching objectives provided in 1.4.1 at exposed endpoints adhering to REST standards.
2. Expose endpoints where a client can authenticate with the API and be provided authorised access to protected endpoints
3. Track player performance on a week by week basis
4. The server must be accessible to a remote client
   * 1. **Server Non-Functional Objectives**
5. The server will be highly modular in nature, providing higher maintainability and extensibility outside of the project scope.
6. The server will only communicate over HTTPS
7. The server will implement security best practices for the industry
   * 1. **Client Functional Objectives**
8. The client must implement all coaching functionality provided to the server
9. The client must be able to communicate with the server
10. The client must provide login functionality
11. The client must provide a user the ability to consent to storage of Bungie API tokens
    * 1. **Client Non-Functional Objectives**
    1. **Process**

The proposed system was likely to create a high level of complexity, especially in the servers functionality; where multiple users must be managed, implementation of security features, as well as integration of LLM technologies in combination with Bungie API requests.

This complexity first led to the consideration of a plan-based methodology, namely the Waterfall Method[6]. This would have allowed for greater scrutiny in planning and design of server modules specifically – where the majority of potential complexity lays. Due to the sole-developer nature of the project and limited time, any methodology that introduced production efficiency had to be considered. On the other hand - due to the prototyping nature of the project - any plan-based approach would have introduced a strictness to features of the prototype, which would have diminished any exploratory functionality that could be introduced to showcase the projects potential as a relevant community tool.

This analysis of process led to the conclusion that a more agile approach would be beneficial in the context of a prototype, as well as such approaches being relatively common for prototyping in the real-world. There is also the fact that prompt engineering for LLM integrations in software is a relatively new field, with little data and research existing where concrete planning of prompts could be interpreted as reliable.

In the end, the Scrum[7] methodology was selected as the most ideal option. This is due not only to its agile foundation, but its relative simplicity in practice. No prior experience exists with web-development; apart from an introductory module in the first year of the degree this report gives completion credit towards. This means a fair amount of research as well as accumulation of knowledge and skill would be a part of any development, making the relative simplicity of scrum a huge bonus to reducing project overhead.

* 1. **Project Toolset**

For reasons mentioned in the process section, the selection of aspects such as programming language, development environment and other technologies, must be kept as simple as possible to reduce learning overhead in the limited lifecycle of this project. Alternative technologies were considered initially, but a static set of tools was eventually curated for the project.

* + 1. **Project Toolset – API Server Language**

Due to the live-service nature of the server-side of this project, a language which prioritizes efficient processing of requests and asynchronousity was paramount. Luckily, an option presents itself here immediately in the form of server-side JavaScript; namely Node.JS[8]. Many real-world enterprises have made use of this language, such as Netflix, PayPal and LinkedIn (see appendix F for a brief investigation of this language). It offers a base of high scalability - necessary for a prototype API – as well as offering asynchronousity as a core feature of the language. Node uses its non-blocking I/O and event loop to handle asynchronous calls, providing the ability to continue (or await if appropriate) with processing while the event loop waits for a “promise” to complete.

It is a single-threaded language at its core, making use of this for its event loop and call stack. This does introduce some performance considerations, especially seen as the server will run on a local machine with slightly dated hardware specs. If necessary however, there are several libraries available to leverage concurrent or parallel processing, strengthening this language as an appropriate choice for this project.

An argument could possibly be made to consider PHP[9] as a server-side language, it provides libraries akin to Node.js that allow for easier development of RESTful APIs. However, it is a synchronous language, making external and concurrent API requests such as to bungie or the LLM would be slow. It would also add undue complexity to the project. Node.js would effectively allow the front-end and back-end to be coded in the same language, and would mean building skills in only one language, instead of two.

* + 1. **Project Toolset – Client Languages**

It is without assertion that a combination of HTML and CSS would be used for the user interface aspect of the client. Integration of frameworks like React[10] was considered, but due to the constraint of keeping any overhead as minimal as possible, it was discarded as an option. No initial familiarity existed with React, but experience with DOM (Document Object Model) based design did exist.

The client-side code language also have a few options. There are languages such as TypeScript[11] and JavaScript[12]. TypeScript is a superset of JavaScript with static typing for more readable and maintainable code. This would benefit a certain aspect of the prototyping nature of the project, that being supporting development outside of the project scope. It does however also impede other aspects. Static typing implies a general plan for development, where classes, modules and functions will have parameters and yields set in stone. This introduction of strictness would hinder the exploratory nature of the project; whereas JavaScript – although more “messy” and less readable – would allow for a stronger dynamic and iterative approach to development. Changes to code through each iteration could be more easily made as types are simply not considered. It would also – as discussed in the server language section – allow for both the server and client to be essentially coded in the same language, reducing complexity in the project overhead.

* + 1. **Server Operating System**

Choosing an operating system for the server is mostly an arbitrary choice. Node.js is a cross-platform language, meaning it is able to operate on any machine regardless of the OS. There may be edge cases for certain versions/if the OS version is dated, but all modern operating systems are supported by it, and code developed on one will run on another.

The code will likely not be compiled into a native executable as well, as this process is usually only for customer facing native applications. Leaving the server as raw JavaScript code that is executed from the node interpreter makes any updates and modifications to the code base easier; removing the step of compilation and any problems experienced therein, as well as making the program more transferable in case of machine failure.

Due to these reasons – and the central factor the developer has direct access to a windows device – the code was developed on Windows 10.

* + 1. **Client Hosting Service**

As detailed in the initial analysis, several options exist for hosting the client. An accessible client is required when registering for a Bungie API key, as a redirect address is necessary for their OAuth2 flow. The simplest solution here is to use GitHub Pages. The free version offers a repository limit of 1gb for public pages (which will likely not be exceeded due to the majority of computation occurring on the server) and automatic building of pages.

There are some limitations such as only supporting static sites and SPAs, but the client will be relatively simplistic, focusing on showcasing the core functionality of the software as well as its potential for development outside of the project scope.

* + 1. **LLM Integration**

As of the commencement of this project, several options exist in the space of integrating LLM/chatbot services into software applications. Langchain was initially suggested by the supervisor, but was later discarded as an option after preliminary research concluded it was an overengineered approach as the projects objectives were formally documented.

A number of LLM services were considered. Vellum[13] was chosen to initially be utilized as a benchmark platform to compare different available models and their capabilities. Below is the benchmarks that had the most impact in selecting a model:

* **Best in Reasoning (GPQA Diamond Benchmark[14]):** The model would have to perform several different kind of analysis on player data to provide feedback, critique and suggestions depending on the coaching tool. Therefore a model with a strong reasoning base was required.
* **Humanity`s Last Exam[15]:** This multi-modal (tests different mediums such as text and images) benchmark is one of the most challenging, comprising of 2500 curated questions across multiple different spaces of academia. Every available model struggles with accuracy here demonstrating a significant gap between human learning ability and that of LLMs. Answers to questions used here are not easily accessible via the internet; this links in with the academic ability the model must demonstrate for coaching objectives such as the knowledge base, as well as activity environments that would affect other coaching analysis.

The outcome of these comparisons found that ChatGPT and Google Gemini[18] 2.0 Pro led respectively (this may have changed since tool selection due to the ever-evolving nature of the large language model market).

But the overall ability of the model wasn’t the only factor. Pricing was another. ChatGPT does not offer any free limit on API requests, which immediately took it out of the running. Paying for a service for prototyping would have been an unjustified design choice as well as an unsustainable one outside the project scope; for a full-community release. Due to Bungie API terms of service, any application integrating it cannot be used commercially, so paying for any LLM services would be solely through community donations, which would again be unsustainable.

This left Gemini flash models as one of the best options for ability and pricing, including a free tier which included 1500 requests per month, and 15 per minute. It also included a context window of up to 1 million tokens, which could be incredibly beneficial if extra context for each prompt was implemented, such as descriptions of activities that would enable higher accuracy in the model and less reliance on web search tools.

During the analysis of potential models, DeepSeak R1 was released, including not just an API, but open source and locally executable models. This could have proven an effective implementation of LLM for the project; removing any rate limits and introducing more control over model training data, reasoning capabilities and many other aspects. There were however severe hardware limitations. Even the smallest of models provided (as low as 7 billion parameters) could not be sustainably ran on the development machine, or any machine located on Aberystwyth University grounds, requiring upwards of 32GB of ram and at least 16GB of GPU VRAM.

* + 1. **Integrated Development Environment**

Constrictions on selecting an IDE are based nearly solely on reducing overhead. The developer has a strong familiarity with JetBrains IDEs, which utilize a mostly standard interface across all their IDEs.

The IDE developed for web development is called WebStorm, with standard features such as a debugger, code completion, syntax highlighting and error detection. It also has specific features for web development and node.js, such as interpreter integration, local package management, support for ECMAScript 6[16] file types as well as support for file types such as .json, .css and .html.

* + 1. **Database Technology**

Initially Postgres was selected as the implementation for persistent storage, chosen for its strong ACID compliance and other rich features. But it was later decided an overengineered approach to a relatively small web application, which in production would likely not see more than a few thousand user records. The project is also a prototype, and so reinforcing this evaluation. More about this is discussed in section.

* + 1. **Server Libraries**

Several server libraries were defined in the preliminary stage of the project that could aid in implementing all objectives and requirements.

* **Express:** Express is a node.js module that allows for a more streamlined implementation and configuration of a REST server. It also provides features for implementations of middleware, either external or developer defined functions that can act on any requests.
* **HTTPS:** Part of nodes standards library, this library can take an instance of an express app and initialise an HTTPS server.
* **Prompts:** An external library providing synchronous console input
  + 1. **HTTPS Configuration Tool**

HTTPS communication requires a valid TLS (Transport Layer Security) certificate as well as a private key for encryption. A client verifies the validity of the certificate with a CA, and gains the private key through standard asymmetric TLS handshaking.

Many services offer certificate signing, but at a cost for the majority (such as DigiCert). Some services do provide free certificate signing, but it was decided OpenSSL which can be run locally could be used for signing certificates for the prototype as well as generating a private key.

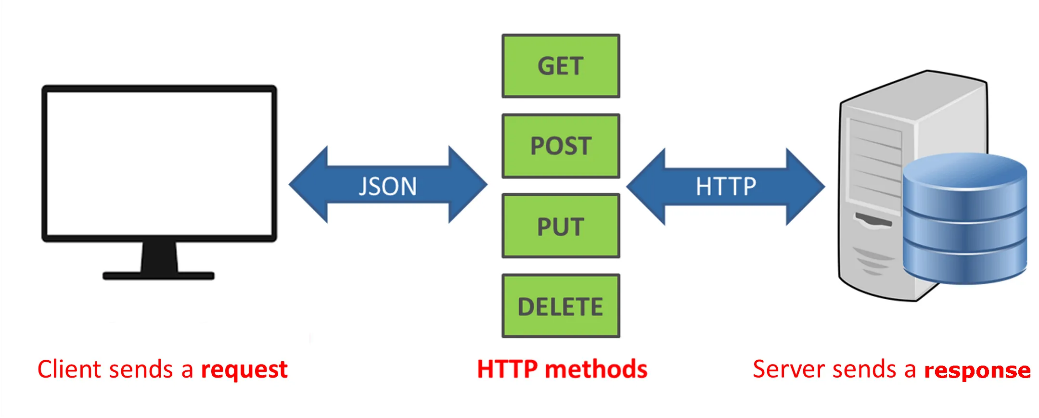
* 1. **Security Considerations**

1. **Design**

This section comprises of the key design aspects of the prototype application In its final state. This covers aspects such as modular setup, application architectures as well as other design properties.

* 1. **Application Architecture**

The prototype application is cleanly split for this kind of project into a front-end and back-end, client and server. Where many common websites and web-based applications have both of these exist on the same domain, such as for server-side rendering of HTML and any necessary computation; our architecture utilizes a remote API server to provide necessary parameters from the client for a coaching tool , and displays the content of the response once fetched.



*Figure 1[17]: Architecture of a client and remote RESTful API*

The client is designed to be a minimal example of the potential use of the API server and integrating it into a community-accessible web application. However, it forms as part of testing the API server. Each coaching tool requires data retrieved about a player from the Bungie API, which further requires access by a Bungie provided OAuth2 code to exchange such for a token pair (access and refresh); using these tokens at protected endpoints. As specified in 1.6.4, a public client is required as part of their OAuth2 flow and for the above reasons.

So while a majority of the work would focus on developing a fully-operational API and integration of coaching features, a dedicated portion of the project lifecycle had to be focused on implementation of a client.

* 1. **Server Architecture**

The server was designed to be highly modular, taking advantage of modular features provided by the ECMAScript 6 language specification that node (and therefore JavaScript) are built upon.

As seen from the above, over the course of the project lifecycle, server structure evolved over time to introduce several layers of abstraction, centralized data retrieval and storage, as well as more unified processing of requests the server.

* 1. **Bungie Access**

The bungie access module handles all interaction with the Bungie API, delegating parsing of responses to an external module (BungieParser.mjs), and returning highly verbose objects to the module that implements it.

The initial implementation of this module occurred in sprint X,

The access module handles a few core functions in this responsibility:

* Provide token pairs from an OAuth2 code or refresh token to be used as a header parameter at protected endpoints
* Fetch platform parameters including membership type enum value and membership ID string
* Fetch all player characters and display name
* Fetch player activity history
* Fetch player inventories (separated by function into fetching equipped items, unequipped and what is stored in a players vault)
* Fetch player weapon statistics
* Collect static data from the manifest for mapping hashes in responses into meaningful identifiers
  + 1. **Bungie Access – Module Initialisation**

The module requires executing an initialisation function before it can be used, which deals with fetching the static data from the API and passing it to the parser module which converts the arrays of game data into hash maps.

In the initial problem analysis, this issue of providing meaningful identifiers was assessed and possible approaches covered. But in development, the decision to store these globally seemed the most appropriate approach. This would allow for much quicker parsing of game data, all identifiers and their respective hashes stored as objects in cache which can be referenced in O(1) time. This did introduce the problem of race conditions for concurrent requests to the server. But this could be handled in a separate project sprint, with solutions available such as variable locks through mutex`s or resource managers such as semaphores. The focus of the sprint that this aspect of the module was part of put the focus on providing one of the coaching tools for the mid-project demonstration, and so ensuring we could retrieve meaningful data for any AI integration was a must.

A screenshot of a computer code

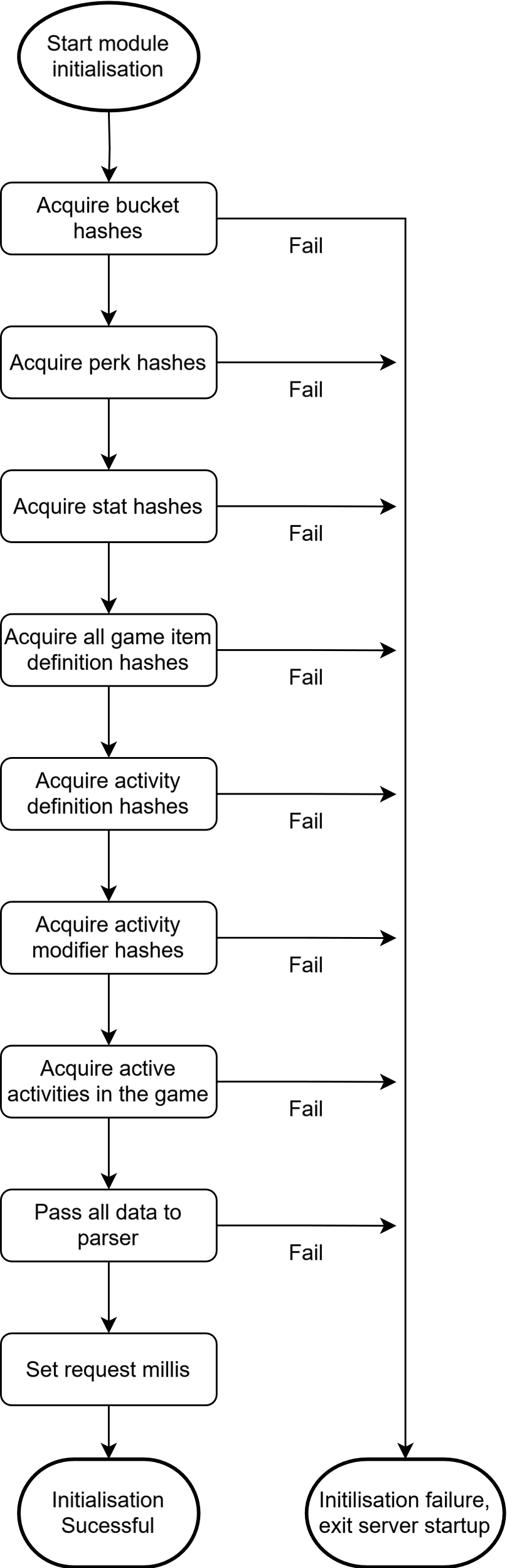
AI-generated content may be incorrect.

*Figure 2: Extract from invetoryData\_example.json, an example response from requesting a player characters unequipped items from the Bungie API.*

Initially, only bucket, perk, stat and game item definition hashes were included, but later other static data was included for implementing further coaching tools, and fetching different data for these from Bungie.

There was an option of storing this data in a database, for persistent storage. However, bungie updates the game every week on a Tuesday, which can include small additions of content or removal; meaning this data would need updating every week. It could have removed the problem of possible race conditions to module global variables; but on the other hand, this would have introduced a substantial amount of extra work on methods/functions to store and update this data. There is no need for persistence of this data between server shutdowns (such as in the case of any fatal errors thrown) in contrast to token storage or user data; the maps are simply reloaded, and a function could be executed on a weekly basis to take the server from active service into a maintenance mode, to update this data and other aspects that will be covered later.

If fetching of any individual pieces of data fails, this would mean all server functionality would be unavailable as there would be no access to verbose game data for the LLM. If this occurs, the server exists startup completely.



*Figure 3: Flow diagram for module initialisation*

* 1. **Server Request Processing**

Request processing at all protected endpoints follows a near identical execution, and is assisted by utility modules which helped reduce the amount of repeating code, as well as introduce aspects of concern separation. These are “ErrorHandler.mjs” and “ParamHandler.mjs”.

* + 1. **Server Request Processing – Parameter Extraction**

At one of the relevant endpoints, “parseAllParams” is called (the centralized function exported by the parameter handler module). If even a single parameter is missing from what is requested by the endpoint, the function returns a response to the client (from a passed Express response object) indicating failure, otherwise returning an object with the extracted values.

* + 1. **Server Request Processing – Error Handling**

Each module on the server throws a custom error if anything goes wrong. In deeper call stacks, these are handled as the error travels up if appropriate, and a new error thrown by the module/function call doing any handling.

Essentially, any error will reach the top-level of the request (i.e. the endpoint function). All endpoint logic is encapsulated in a try-catch block, where in the catch, the endpoint calls “handle” (exported by “ErrorHandler.mjs”) in the case of an error, passing this error and the response object for the module to respond to the client with.

* 1. **LLM Integration**

LLM functionality for the coaching service was encapsulated in an ES6 class, namely Reasoner.mjs. This class is instantiated as a singleton at server startup, and passed to every coach object for a user. This is done to keep server memory usage low, as the reasoner loads pre-structured prompts and chatbot response schemas into global variables. As with other uses of global variables, this does introduce a problem of race conditions.

The reasoner class has each coaching functionality in its own function, with a private function that handles generating content, and allows for dynamic aspects of content generation to be passed as parameters, such as response temperature where a coaching tool needed (or did not need) a measure of determinism. The response schema and prompt (which has its game data parameters injected) are also passed to this function.

Each function called by the coach class (the functions that provide coach functionality) follows the below execution flow.

The only exception here is the function used for keyword explanations. The response for each type of keyword is specialised to receive specific properties of that type. The function uses a switch statement on the “type” parameter to select the correct schema from the global store.

* + 1. **LLM Integration – Response Schemas**

Each coaching tool has a defined response schema to retrieve model output in a specific structure. Not only that, but this aspect of LLM integration allows for each property of the coaching response to be provided with a description, specifying what analysis needs to be carried out on the provided game data.

Initial versions of this class did not utilize response schemas, and used the prompt itself to direct the model to produce its output in a specific format. This did produce inconsistent response bodies and would have required extra parsing capabilities, increasing workload.

With response schemas, analysis and the response properties could be streamlined. It also means for development and maintenance outside of the project scope, altering response schemas does not affect the prompt itself.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 4: Example Response Schema – This specific schema is used as part of the application knowledge base, providing a response structure for explanations of specific activities in the game, such as instances of raids or dungeons.*

* + 1. **Prompt Engineering**

Prompt engineering – like LLMs themselves – is a relatively new scope of software engineering. Many applications that integrate LLM functionality utilize pre-structured prompts that inject any necessary data into the prompt. This is the approach the project utilized for introducing player game data into prompts. Utilizing regular expression pattern matching, identifiers in the prompt would be replaced with JSON.

A destiny 2 player wants to improve their weapon skills, specifically improve their skills with different weapon types in  
different situations (activities). The player has the current statistics for each weapon type, including the number of precision kills and total kills:  
  
-STATISTICS-  
  
Analyse these statistics, looking for the weapon types the player should become more familiar and types they should practice becoming more precise with. Take into account the fact that melee, super and grenade precision kills aren`t applicable

*Figure 5: Example pre-structured prompt. This specific prompt is for providing the coaching tool that analyses a players weapon skills, yielding suggestions on which to practice with, what environments/activities to practice in, as well as providing explanations for how each weapon type aids combat, and the unseen penalties for omitting them from a players general arsenal.*

The main problem for prompt engineering is ensuring consistency of responses. This is not only the deterministic aspects where necessary, but also what details they include, such as descriptions of activities, consistency of coaching tone and analysis properties for applicable coaching tools. The general evolution of the prompt set here to achieve this was through evaluating the qualitative aspects of responses from Gemini, and refining each prompt in an iterative process.

* 1. **User Services**

User services is designed to encapsulate fetching non-authentication related data from bungie and the user database, reducing the number of dependencies on coach instances. It essentially acts as an abstraction layer between coach instances and the data required to be passed to the reasoner singleton for text generation; handling all retrieval logic and creating clear separation of concerns between coach instances and the handling/retrieval of data, acting as a core service of the API (that is also abstracted away from server endpoints).

It also acts as a near complete single source of truth (authentication services is the only other place where database access is used, and the functions it has access to are severely restricted to just those required for authorisation and authentication) for the entire application, as setting data in the database and retrieval from both it and bungie occurs through here, and no other source. While it is not the “truth holder”, it is the single source for all access and manipulation. This kind of behaviour is a core feature of many multi-user applications and systems, ensuring data consistency and synchronization. Past the project scope, this module can be easily expanded for other more complex functionality, based on this foundation.

Most functions that require data from the bungie API first fetch the standard path parameter set for protected bungie endpoints from the user database, utilizing a “userid” (which is the primary key in all database tables and is the players global bungie display name; a unique value) to fetch such. The functions then generally use this parameter set to retrieve data through functions on the bungie access module.

User services exports the below functionality, with brief descriptions of how they relate to the rest of the applications features.

* + 1. **User Services - Storing User Items**

user items were stored in the database to increase speed of coach requests where player items are a parameter of the reasoner. Initially the speed of requesting all items from bungie took upwards of 40 seconds, as the items retrieved from the Destiny2.GetProfile response would need further requests

**Evolution**

This functionality was later deprecated however, and is no longer connected to any execution flows. The items stored for a player would need to be updated regularly, requesting the items list from bungie, and checking this against the items stored for a player. This would have introduced significant work to develop and implement an algorithm to handle such. Not only that, it was later discovered that certain query parameters could be combined with the endpoint to retrieve the extra data on item instances in a single call (perks, stats, etc.).

* + 1. **User Services – Retrieve an activity summary**

As part of coaching objective 4, the coach should provide analysis and feedback from an activity summary. One of the endpoints on the server allows a client to request a players most recent activities, providing the instance Id of the activity as part of this. A client submits this back for the endpoint that handles this functionality as a path parameter, which the coach instance provides to the user service function.

* + 1. **User Services – Retrieve a players entire activity history**

A players entire activity history is retrieved for coach objective 6, to analyse its entirety and suggesting new activities to improve skills, activities to repeat and so on.

* + 1. **User Services – Retrieve player weapon statistics**

This functionality maps to coach objective 2, retrieving a players general weapon statistics which includes properties for each type of weapon, as well as the distribution between precision and total kills with this type.

* + 1. **User Services – Retrieve a characters current build**

This functionality maps to coach objective 3, and retrieves the current build of the requested character, passed as a path parameter on the appropriate server endpoint from the client, and passed as a function parameter here

**Evolution**

Initially, this function required several calls to the bungie access module, as a centralized function for character configuration had not been implemented; instead separated into different calls to the Destiny2.GetProfile endpoint with different component values.

* + 1. **– User Services – Retrieve players recent activities**

This forms as part of analysing activity summaries, where a user of the client can select an activity from their most recent, submit this to the VanguardMentor.getActivityAnalysis endpoint.

**Evolution**

This was one of the later additions to the modules, implemented as part of sprint X, focusing on implementing necessary endpoints for the client to have what it needs for further requests.

* + 1. **– User Services – Retrieve knowledge base**

The server provides a curated list of all key terminology found in the game, stored as json on the servers local machine. This isn’t called from a coach instance, rather called from the top-level of the endpoint itself for simplicity. Coach instances only retrieve data that will be introduced into LLM technology. Implementing this functionality here made the most sense. User services deals with essentially all game data, albeit most of it is player game data. But it encapsulates the context of the game, making implementation here the most logical choice.

The client requests this data from the server at the VanguardMentor.getKnowledgeBase endpoint.

* + 1. **– User Services – Retrieve all items stored in a players vault**

Retrieving the items stored in a players vault is a key component of coaching objective X. This tool suggests a build for specific activities, using all available player items to construct such.

**Evolution**

This function was not implemented until sprint X, due to lack of clarity in the Bungie API docs as to how this information could be retrieved.

* + 1. **– User Services – Retrieve player progress data**

As part of coaching objective X, the application must provide a way for a user to see how their skills and ability are evolving. Certain key statistics are stored in a database about a player (covered in a later section) which update on a weekly basis. The database stores the previous and current weeks values for weekly comparison.

**Evolution**

The initial plan for tracking player progress was to utilize global leaderboards, and aggregate across multiple different types of leaderboards. At first it seemed like the Bungie API provided access to player leaderboards, requesting this through a component value on the X endpoint.

However, it became apparent that this was not provided currently. There was no indication of this being an in progress feature of the API. Several other options were investigated, such as other community tools that provided their own leaderboards constructed manually, or implementing an algorithm to do this manually. One option was DestinyStatTracker[17], but it was found their API was no longer public. Developing an algorithm was possible, as activity post game carnage reports (This includes detailed properties of an activity completion and is used in bungie access and its parser) included details for other players where we could request their recent activities. This would have led to a type of crawler algorithm, likely based in recursion. This would’ve although, introduced significant overhead for a relatively small feature of the application.

The decision was made to select core statistics offered by the bungie API that best gave weight to a players general combat skill. More information regarding this can be found in section 2.8.

* + 1. **User Services – Weekly Update**

User services acts as a centralized controller for all data that flows through the server, excluding authentication flows. Any data that needs updating, removing or fetching is requested from here. This basically creates a “single source of truth” as neither the database nor bungie access module are accessed directly anywhere else.

Due to this controller aspect of this module, it was decided that scheduling updates to any data would be done here.

The weekly update is executed every Tuesday at 18:05 BST. Bungie updates the game data, activities and can also add or remove content at these times. From the weekly update, we can trigger bungie access to refresh the static data it passes to its parser module. Failing to update this data means the active activities in the game provided to a client (upon request) will be inaccurate, such as the weekly nightfall, raid modifier rotations and any new pieces of content introduced.

The weekly update function also handles updating coaching data held about a user. In this, all user ID’s and bungie request parameters are fetched from the database, and a request made to bungie for each; fetching the new statistics and updating the database through the database module.

* 1. **User Database**

The database architecture in the application can essentially be split into two components, the MySQL server itself with the database schema, and the module which interacts with it (user\_database.mjs).

A relational database was selected for several reasons. One of the foremost is that initially it seemed a lot of the data that would be stored about a player would be highly decomposable, and this atomisation would yield a consistent structure. For example, the user table stores the global display name, account identifier, membership identifier and type, as well as user characters.

* + 1. **User Database – Server Module Overview**

Server side interaction with the database is encapsulated within a single module, providing a single source of truth for all persistent user data. It exports two distinct objects for implementation in certain controller modules:

* “**dbBaseServices**”**:** This object includes all functions not related to server security services, such as updating progression data for users and retrieving the standard set of path parameters required for requests to the Bungie API.
* “**dbAuthServices**”**:** This object includes all database functions required for authorisation and authentication that occurs on the server. This is implemented by the UserAuthentication.mjs module, which deals specifically with authenticating via OAuth 2, as well as authorisation at protected endpoints and requesting new token pairs.

The module connects to a MySQL instance root user. Instead of storing the password in an environment variable, the module asks the user to provide this at module initialisation, to provide an aspect of improved security. Tokens are stored in the database, so it is important to protect this information wherever possible.

The module utilizes several global variables for performing its duties:

* “**driver**”: This is the MySQL2 node.js package used for asynchronous access to an external MySQL server instance. Several libraries exist here, but this specific package provides the asynchronous operation we require. While the MySQL server runs local to the server and therefore experiences little to no latency, it does avoid any potential issues where port access is slowed, as well as to support the prototype past the project scope, where the possibility exists that the MySQL server may not be local.
* “**console\_reader**”**:** This holds the dependency object for the “prompts” node.js package. This provides a high-level interface for reading data from the console. In this case, it is used to read the password to the MySQL server. The primary feature for its use is the inclusion of an input masking option, important for reading a password from the console.
* “**connectionPool**”: Upon module initialisation, the module creates what is known as a connection pool. This is essentially the consideration for concurrent database access. A single connection was originally implemented, but it was later decided that a pool of connections managed by the driver library was more appropriate in the context of the project.

The module itself makes use of stored procedures created on the database schema, protecting against any chance of SQL injection attacks; with stored procedures being parametrised so no values can be interpreted as query code. Every function interacting with the database uses a stored procedure. There is an exception of one here, where a function exists to fetch all user ID’s from the database, but this takes no parameters so is implicitly safe.

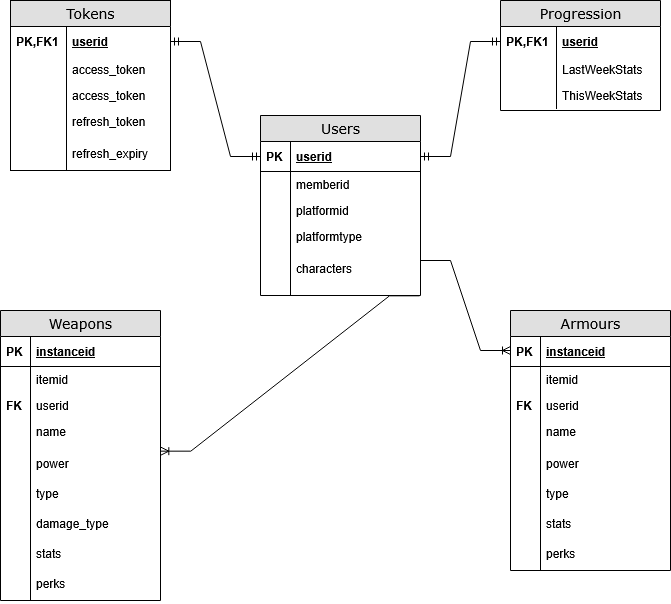
* + 1. **User Database – Server Module Initialisation**

The module requires initialisation before any server operations can utilize its services. This process is called from the top-level of the server, like all modules requiring initialisation.

Dependencies are injected here, as well as the connection to the database established by reading configuration values from a value and receiving the root password from console input. If the configuration is not rejected by the MySQL server instance, server startup continues, but will exit if any errors occur.

* + 1. **User Database – MySQL Server**

The database itself is made up of 5 tables, however two of these are deprecated in the final version of the prototype, but remain in case any use can be identified past the project scope; these being the Armours and Weapons table.



*Figure 6: Entity Relationship diagram of MentorDB*

The database does not make use of transaction locks. This is a deliberate design decision. There will never be concurrent access to a single record on a table. Each table utilizes the user ID of the users table as its primary, and foreign key; and every request to the server (that requires access to the database), will only ever be requesting data for a single user. There is the possible edge case of a user opening two client instances at the same time, but this was deemed an acceptable risk for a prototype.

**Evolution**

Initially, PostgreSQL was selected as the database technology for the project. However, it is a more complex RDBMS (Relational Database Management System, or in the case of Postgres, an ORDBMS) compared to MySQL, which benefits from a more streamlined setup process and is generally accepted as the preferred option for small-scale applications. While Postgres is a valid option for web applications, it is more optimized towards enterprise applications; while MySQL offers the light implementation that smaller-scale applications of any kind benefit from.

In the context of a prototype, an RDBMS with less overhead and simpler interactions with a server instance is a preferred choice. Coupling this argument with the fact that database interaction is encapsulated within a single module, it makes switching out the database service with a different RDBMS if development outside of the project scope could benefit from such.

The database also evolved its table set over time. Initially the user, weapons and armour tables were implemented, and the bungie refresh token stored alongside basic player data. This wasn’t very effective use of an RDBMS, as well as the fact the server structure was evolving (as detailed in section 2.1).

* 1. **Authentication Services**

Server security is handled by a single module, namely UserAuthentication.mjs. It is only implemented in the main server file, exporting a single object for the server to use its functions; a singleton like many other services on the server.

The module has two dependencies, from both data sources. As previously mentioned, these data sources export two distinct objects, one for base services (implemented by User Services) and one for authentication services (implemented by this module). This is done for clear separation of concerns for these different but similarly rooted services, providing an element of low coupling and higher cohesion.

This module encapsulates logic for the following use-cases:

* **Authentication via Bungie OAuth2 code**:

This is encapsulated in a single function call (like all other use-cases) to “baseAuthentication”, passing in a newly instantiated coach object, as well as the auth code passed in the body of the request. The method fetches all basic details from the bungie module and attaches the necessary elements to the coach object. It then checks with the database to ensure no existing entry for this user in server persistent storage. If no existing entry is found, we add them to the database; if the opposite is true, we only need update the tokens on the database (as fetching a user from Bungie involves requesting tokens, which will invalidate what exists on the DB).

* **Authorisation at protected endpoints:**

This is a relatively simple procedure. A token is passed to “authorize()” where it internally calls functions on the database module to fetch a user ID tied to this token; yielding such to the top of the call stack (top-level of the endpoint). If no matching token is found, the function throws a “InvalidTokenError” (defined in Errors.mjs), which is caught and handled at the top-level of the endpoint, responding to the client appropriately.

* **Reauthentication via refresh token:**

This is also a relatively simple procedure. A refresh token is passed to “reauthorize()” where new token pairs are requested from bungie (via the Bungie module), updating the database, and yielding the new tokens to the top of the call stack for the server to return to the client.

* 1. **Coaching**

Each user who registers their presence on the server is tied to an instance of the coach class, found in Coach.mjs. It acts as a core service of the API, exposing it as a utility/service at endpoint URI`s providing coaching.

This object stores some basic information about a user, including the global bungie display name, mapping to a user ID on the database.

The object takes two dependencies, the reasoner singleton, and the user service singleton; provided to it at instantiation at the top-level of the server, specifically the endpoint designated for base authentication (authentication by bungie OAuth2 code). For each coaching functionality stated in section 1.4.1, the coach class exposes a method for retrieving such content. It fetches the necessary data from user services, and passes this data to its reasoner dependency for generating the necessary content.

The class also includes methods for retrieving user persistent data and bungie data from user services, such as progress tracking data (from the DB) and a players recent activities (from Bungie).

**Evolution**

The coach class is one that has changed considerably throughout the project lifecycle. Initially, it stored nearly all data now found in the user table, as properties of itself; this was before the first version of the database was implemented, in the sprint running up to the mid-project demonstration.

This included the access token necessary for requesting data from bungie. At the time, user services did not exist either, and the coach interacted directly with the bungie module; making an in-memory store of standard path parameter data and tokens an effective design choice. But this did have flaws for token management, and created use-cases where tokens become invalidated, cannot be updated centrally, leading to inconsistencies in data stored for a user across the server.

Eventually the database evolved and user services was implemented to act as a data controller and abstraction layer for the two sources. The coach requests data from user services simply by using the user ID tied to itself, which user services then incorporates for its operations.

* 1. **User Pool**

The user pool class is a core component of processing requests made to the server involving the coach class. When a request is authorised at a coaching endpoint, the user pool is the entry point for processing. It stores in a global hash-map (a property/attribute of itself and implemented specifically for its O(1) read time) every coach object instantiated for an active user.

As with other core services, only a single object is instantiated, making it a singleton.

The authorisation function at a user-specific endpoint yields the user id that maps to the provided access token. The user id is passed to the pools “process” method, alongside a request code and any arguments extracted from the request that are necessary for further processing (such as character ID`s, activity ID`s, etc.).

**Evolution**

The user pool initially had coach access tied to the access token, acting as a quasi-authorisation system. However, as with the evolution of the coaching class away from decentralized token storage, this would ultimately lead to token inconsistencies between in-memory storage and persistent storage on the database. This would have also led to automatic token invalidation if the server crashed or was brought offline. At the suggestion of the supervisor, token management was moved to a centralized token table and management module.

Instead, the authorisation module was implemented to fetch the associated user ID, and this was used instead to identify coach instances in the global map.

* + 1. **User Pool – Process**

This is the entry point for user-specific endpoint processing. Using the user ID, it extracts the reflecting coach instance (or atleast its memory address seen as node passes objects by reference, and primitives by value) in the global hash map.

* + 1. **User Pool – Request Codes**

When the process method is called, a request code is also passed (an enumeration value is actually passed, the declaration of which can be found in modules/Enums/PoolRequestCode.mjs), which is used in the user pool to call a specific method on the coach object.

* + 1. **User Pool – Append**

At the base authentication endpoint, a coach object is instantiated for the user and appended to the “active\_pool”. In this process, the coach instance is appended to the pools global map by its user ID. Before appending however, the pool checks for an existence instance tied to that user, in the use case of authenticating twice at base authentication.

* 1. **Client Design**

The design of the client can essentially be decomposed into two distinct (albeit differently sized) components; logic and document elements for logging on, and the same for logged on. This essentially divides the client into two separate states. This is also due to the static nature of the client hosting service (and SPA aspect); a single page is utilized for the entirety of the client (namely index.html).

This has the basic header information expected with any html document, but also executes paveProvider.mjs upon instantiation (client is accessed by a user).

This script essentially checks if a bungie OAuth2 code is present as a query parameter in the URL. If it is, the script fetches “coach.html” (contains the authenticated body) from the repository/domain and injects this into index.html

* + 1. **Client Design – Non-Authenticated State**

It will also after doing so load in the necessary scripts required by the authenticated state (injecting a body does not automatically execute any tagged scripts in the body, so must be injected separately to be executed). If the auth code is not present, the page provider fetches and injects the “login.html” body, containing a button that once clicked, redirects the browser to bungie`s OAuth2 login.

When a client is redirected to bungies login, if authentication is successful, bungie will redirect the user to the address provided when registering an application for their API; placing the authentication code as a query parameter in the URL.

* + 1. **Client Design – Authenticated State**

The authenticated body is composed of 4 key divisions:

* **Page Header:** The page header contains some aesthetic elements, but most importantly the element that allows a user to select the character to use for any coaching tools that need that parameter.
* **Tool Selection:** All coaching tools are contained in a vertically organised “div” element with some aesthetic icons provided by googles material 3 design.
* **Tool Container:** A “div” element that holds any other necessary selectors specific to the tool and what displays coach content fetched from the API. Each method attached to a button deals specifically with loading this element.
* **Bottom Header:** Contains some basic information including the users Bungie global display name as well as their progression data stored on the API`s database.

1. **Implementation**

This sections discusses and documents the main difficulties experienced during implementation and how design was modified (if applicable) to overcome such. Each section will cover a specific difficulty.

* 1. **Bungie API Response Parsing**

Implementing parsing algorithms for Bungie responses proved to be a time-consuming and cumbersome process. Parsing algorithms were implemented one by one – where the sprint required it – and continuously introduced complex sequences of operations to generate verbose object structure.

Responses from Bungie can be inconsistent in some cases, with missing object properties, nulled values and unresolved references. Most responses are also schema-heavy and verbose in metadata (see figure 4, or JSON used as test data for parser unit tests in the technical submission). With these properties, comprehension of response structures alone was a time-consuming undertaking, and did result in some early algorithms that didn’t take full advantage of API capabilities and structure (such as documented in section 2.3.2).

The documentation for the API can also in some cases provide insufficient detail on current status of API features (such as that discussed in the evolution sub-section of 2.6.9).

* + 1. **Bungie API Response Parsing – Parse Items**

One example of the complexity these algorithms can reach is best evidenced in the “parseItems” function exported by the parser module. This algorithm was implemented as part of the last sprint to optimize efficiency, inclusion of more item data.

The initial iteration (before parsing was relocated to a separate module) did not include weapon types, and was not capable of parsing items fetched from the vault. Items on a character (equipped or unequipped) have distinct differences in metadata compared to those in a user’s vault:

* **Bucket Hashes:** The bucket hash of an item determines the slot it fills, such as kinetic (primary weapon slot), energy (secondary weapon slot), power (heavy weapon slot) and every other slot found in the game (in some instances referred to as plugs for armour and weapon modifications). For items returned from a players character, the bucket hash evaluates correctly. But for items from the vault, all hashes evaluate to “general”.
  1. **LLM Response Consistency and Analysis Ability**

The properties and content of responses from the Gemini LLM introduced numerous issues throughout the project life-cycle. Each coaching tool had a checklist (see appendix X) that acted as an evaluation criteria. If elements of the checklists were not met, the prompt was adjusted, culminating in an iterative process of improvement. Several general response aspects can be identified, these being those that were continually tough to address:

* **Tone:** The tone of the response was continually inconsistent. The aim was to have response text generated as if it was speaking to a player in the context of the game. Introducing this request in the prompt was not always acknowledged.
* **Web Search Tools:** For some tools, web searches are required for the LLM to collect detailed activity details such as environments, activity mechanics, enemies, etc. The results of these were not always inclusive of all details. Higher clarity in the prompt on how and what to search did improve this by a measure, but there is still occurrences where the model is not including all necessary details. While this could be solved by introducing pre-defined context for each activity (The Gemini model utilized has an extensive context window); doing so for every activity to an effective extent would have been incredibly time-consuming and would have compromised meeting all coaching objectives. In addition, the ever-evolving nature of the game and regular addition of and updates to activities would introduce an aspect of lower maintainability to the server.
* **Response Structure:** throughout the project lifecycle until sprint X (where response schemas was discovered as a solution to this), the structure of the response was detailed in the prompts themselves. It would usually include text outside of the structure, and would not always adhere completely to the in-line schema either.
* **Loss of Context:** for all prompt implementations, the model would continually miss aspects of any required analysis in generating a response. For the deeply analytical coaching functions (such as coaching objectives 1,3,4), the prompt architecture is made up of 3 components:
  + **Introduction:** A brief description of the objective of the response.
  + **Analytic Properties:** Instructions to build an internal analysis with a list of key aspects to investigate.
  + **Final Instruction:** A final confirmation of instructions, and other response details such as tone as well as any necessary constraints (where these constraints are deemed effective).

This architecture proved the most effective in ensuring consistency of analysis, although in the final prototype, it does still occur at random intervals.

* **Maximum Output Tokens:** This proved a consistent issue throughout coaching development aspects. For the coaching objectives (specifically objectives 1 and 3) that required more extensive analysis, Gemini would often halt response generation (FinishReason: MAX\_TOKENS) in the middle of analysis.

Multiple alternating strategies were implemented to attempt to counter this; such as including in the prompt to cover all response properties within the token limit (for Gemini-2.0-Flash, this is 8192) as well as specifying in certain response properties that implicitly require heavier detail to be concise. These strategies were neither effective nor reliable and severely compromised the prototypes ability to meet its requirements.

A solution presented itself through the release of a new experimental model (gemini-2.5-flash-preview-04-17) with a greater token limit (nearly 65,000). While this solution now allowed extensive and detailed responses for all objectives, it is an experimental model with much lower rate-limits and subject to change. It is appropriate for a prototype, but considerations past the project scope will most likely have to be made to utilize a more reliable LLM, as it will not be appropriate for a community release due to the aforementioned reasons.

* 1. **Server – Client Communication**

Communication between the two stacks of the project was initially established through the use of an NGrok Secure HTTPS tunnel established on the server machine, allowing the client located on GitHub Pages to communicate through this proxy and to the server. This proved for a majority portion of the project lifecycle to be an effective implementation. It kept the servers local IP protected (any address in the client are publicly exposed due to the nature of GitHub Pages) and NGrok could handle any abuse of endpoints. It was initially introduced for these reasons, but also due to an issue (that is still unknown) that stopped the server receiving any requests.

However, in the final sprint when endpoints were connected to the relevant back-end services and tested, it was found NGrok would intercept requests with a security page, asking the client if they wished to proceed. This did not occur at the endpoint for basic authentication, which for the majority of the project lifecycle was the only active endpoint. Due to this, it was decided to expose the local IP for simplicity and only ever done so during manual testing of any implementation done. After this exposure, local windows commands could be used to renew the IP with the router and release it.

1. **Testing**

Three different types of testing were carried out for this project, divided into the sub-sections below.

**4.1 System Testing**

System testing is utilized for verification of the entire system including client – server communication. This essentially tests the entire sequence of calls that occur when endpoints receive requests, the ability of the client to send requests, and to display them. All tests are performed from the client using the provided tool buttons.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test ID** | **Function** | **Instructions** | **Expected** | **Actual** | **Pass/Fail** | **Comments** |
| **1** | Request an analysis of a character | Login to client, click character analysis button with a character selected | Response received and displayed in tool window | Response received, not displayed | Partial | Client bug where paragraph element not displaying content |
| **2** | Request analysis of activity performance | Login to client. Click activity summaries button. Randomly select activity, click go | Response received and displayed in tool window | Response received and displayed in tool window | **Pass** | None |
| **3** | Request weapon skills analysis | Login to client, click weapon skills button | Response received and displayed in tool window | Response received, not displayed | **Partial** | Client bug where paragraph element not displaying content |
| **4** |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |
| **6** | Request targets | N/A | N/A | N/A | **Fail** | Not implemented to the client. Server missing game data |
| **7** | Request Activity skills analysis | **N/A** | N/A | N/A | **Fail** | Partial implementation to the client. Server missing response schema |

Table 1: System Testing

**4.2 Unit Testing**

Unit testing is implemented for the parser module verify its ability to deal with the sometimes deeply nested, heavy payload JSON that the Bungie API produces. The path to the unit testing module in the technical submission can be found below:

* “server/Tests/ParserTests.mjs”

Executing this module will automatically run all tests for all cases of parsing requirement.

**4.3 Requirements Based Validation Testing**

Each coaching functionality has a type of requirements specification to which responses are checked against. An LLM request is performed identically 5 times and checked against its requirements.

A testing module exists to fetch LLM responses without testing the entire system and call stack. This was utilized to verify these requirements and is available to use.

**4.4 User Testing**

User testing was carried out with an individual for all coaching functionality the project provides. Feedback and consent forms can be found in the relevant submission section.

1. **Critical Evaluation and Insight**

This section discusses and evaluates the strengths and weaknesses of the project in its final state, as well as applying scrutiny to the decisions made throughout the project lifecycle, and the general approach to the problem outlined in section 1. It will also make suggestions for improvements and features that could be implemented past the project scope.

**5.1 Evaluation of Coaching Objectives and Implementation**

Coaching objectives are perhaps the most critical aspect of the project. The modular approach taken to implement these objectives make it highly extensible; where additional prompts, response schemas and alternative response temperatures can be implemented to refine and expand existing functionality. Further to this, the entire module can be switched out to implement a different LLM SDK or API; an effective mode of application for an industry that evolves by the day with new or improved models released with enhanced reasoning capabilities.

The objectives themselves are mostly fulfilled, as evidenced in section 4.3. Tone, context gathering and response consistency could all be improved through further refinement of materials and the inclusion of rich context where appropriate (context for activities, environments, etc.). Regardless of this, some do produce useful content for a new or beginner player, and certainly aid in flattening the games learning curve to some extent.

The prototype only covers a small portion of the games complexity, and while these objectives can be considered to address the foremost difficulties a player would encounter, there are a plethora of additional content that the project could be extended to cover. Such as:

* **Weapon Crafting:** Certain weapons can have their “shape” extracted, and shaped at the relic in the enclave; a process which can be difficult to understand to even the more intermediate players who have not interacted with this content. The coach could suggest certain weapons to craft that would aid a players arsenal, and the perks to select in that crafting.
* **Quests:** For a new player, they are overloaded with a plethora of quests to complete, but are provided little to no context to how it relates to a part of the game. The coach could provide an order for these quests, as well as context for each for what expansion or season it links too, how its completion extends their knowledge and skills, as well as any other context that could be identified.

The requirements for each coaching objective were identified without any formal analysis of the game features/content they address. While the developer has extensive knowledge of the game, making this approach mostly appropriate for a prototype; a lack of formal investigation into the properties of the content leads to the fact that coaching may lack complete cover over the contents complexities and therefore reduced coaching efficiency.

There is also a lack of initial research into the capabilities of the Gemini SDK, which led to instances of refactoring the coaching sub-system to make full use of the library (inclusion of response schemas, temperature, etc.). Time was lost to these instances that could have been spent on other components of the software.

**5.1.1 Evaluation of Coaching against Objectives**

In the technical submission, 5/7 coaching objectives have been met. These are 1,2,3,4,5. Omitted objectives are:

* 6: Analysis of a players entire activity history to provide feedback on a players wider skills was not implemented. The project fetches the correct data from the Bungie API, and a prompt does exist, but neither a response schema, nor dedicated server endpoint were introduced to provide this to a potential client. This is mostly due to time constraints and the unexpected complexity that the project adopted throughout the project lifecycle. This objective is partially met in the final version.
* 7: Target objectives were intended to be manually tracked by the server. This did not come to frutition however, and only receives
  1. **Evaluation of Server**

The design of the server evolved over the course of the project. In its final state, it benefits from a highly modular structure, appropriate not only for a prototype where the software can be easily modified based on its evaluation, but for software maintainability and extensibility in general. While some modules do rely on direct dependencies (as is normally the case with complex systems), many logical responsibilities are atomized and relegated to separate server modules (constants, enumerators, error and parameter handling, utilities). There are several critiques that can be made however.

* + 1. **Evaluation of Server - index.mjs**

The main file of the server encompasses many responsibilities, including and limited to:

* Dependency configuration and injection
* Module initialisation
* Express, HTTPS and Cross-Origin Resource Sharing configuration
* API route definitions
* Server binding and listening

These responsibilities could be delegated to their own modules for a clearer and more maintainable startup process. Not only this, but configuration could also be made more dynamic, loading configuration values such as CORS details and server bindings from a file, leading to increased transferability. Throughout the project lifecycle, index.mjs had its responsibilities continually increased, with no sprints dedicated to dealing with this.

* + 1. **Evaluation of Server -**  **Security**

Security in general is handled by a single module, and tokens stored on the database. However, some important security implementations are absent in the prototype:

* **Token Renewal:** A users Bungie API tokens are not automatically renewed with bungie upon expiry. This was a planned feature that was eventually discarded due to the time constraints of the project. Expiry times are stored, so implementing this process would only require a mechanism to track expiry for users.
* **Token Encryption:** Typically where tokens are stored in persistent storage, they are encrypted so that in the case of a database breach, any stolen tokens cannot be used. Techniques for implementing this were considered, utilizing the crypto library part of Node`s standard library as well as key rotation for added security precautions where the encryption key is stolen or exposed, limiting the window of vulnerability.

While this feature may not be implicitly required for a prototype, it does add inflation to required system hardening before it can move into a production phase.

* **Server Tokens:** The server does not implement its own token system, merely borrows from Bungie`s OAuth2 flow. This is an inherently flawed approach. In the case of any token leakage-either client-side or server-side-a bad actor would have the ability to access and manipulate a potential users data. It would also have the indirect consequence of diminished trust in the application in a production environment as well as potential legal ramifications through Bungie`s terms of service for the API. There is a small amount of protection here through the scoping of tokens the project holds and the granular control aspects of Bungies OAuth2. The application only requests read rights when signing up for an API key, and a user must give consent to this when giving this application permission to access the API on their behalf.

Disregarding these missing aspects, security is generally handled well. All endpoints accessing a users bungie data requires an access token, and the expiry of said tokens is handled, where an endpoint is provided for a client to reauthenticate via a refresh token.

* + 1. **Evaluation of Server – Functional and Non-Functional Objectives**

The server side of the project/prototype was allocated the majority of the development, and is hence where most of functionality lies. The server has achieved nearly all of its objectives.

All endpoints adhere to REST naming including naming conventions and consistency; as specified in the OpenAPI specification. Endpoints make appropriate use of path, query and header parameters in addition to correct use of HTTP methods in relation to resource manipulations.

The server also exposes nearly all necessary endpoints, but the following are omitted which would be required in a client:

* **Currently Available Activities:** The current activities would be required by a client for a potential user to select one for retrieving an activity build suggestion.
  1. **Evaluation of Client Objectives**

At the commencement of development, the client was intended to be an example of what functionality could be provided. More time than originally planned was however dedicated to server-side feature implementations, with the UI implemented in two separate sprints (sprint leading up to and including the mid-project demonstration and sprint leading up to submission). This last sprint did not allow for full implementation of UI. A bug exists where content is retrieved from the server is not displayed in some instances

* 1. **Evaluation of Persistent Storage**

Retrospectively, a more appropriate choice of database technology would have been document based. While a portion of data kept about users can be atomized into fields for a relation DBMS based in rigid-schemas, and an aspect of normalization can exist; in the current scope a great portion of persistent data is kept as JSON, and so the schema-less design of these kinds of database implementations would prove more effective, especially for development and extension of functionality past the project scope. It would also prove beneficial if caching of deterministic coaching responses was implemented to save on network request times.

* 1. **Appropriateness of Process**
  2. **Hardening Tasks**

This section covers the different hardening tasks that would require (but also includes recommended) implementation before the project can move into any potential production phase.

* + 1. **Hardening Tasks – Required**

Below are the tasks that are critical requirements for the project to be reliable and sustainable outside the current scope of a prototype:

* **Schema Validation:** Where JSON is expected or produced, validation of the structure must occur to protect against any inconsistencies or malformations, including the responses generated by the client as well as what is received and produced by the Bungie integration systems.
* **Server Token System:** The server should implement and distribute to clients/users tokens it creates, and not expose Bungie`s tokens.
* **Full Implementation of objectives:** Coaching objectives that are not met should be implemented.
* **Database Encryption:** Tokens stored in the database should be encrypted and implement one of the available key rotation services on the market.
* **Improved handling of Bungie API Errors:** 
  + 1. **Hardening Tasks – Recommended**

Below are the tasks that are recommended for the project outside of the project scope, focusing on efficiency, coaching ability, and other necessary factors as detailed:

* **Refactoring of Bungie sub-system:** Deprecated functions should be removed where not required for any proposed extension to current functionality. Parsing of static game data should be delegated appropriately to parser module.
* **Configuration:** Configuration aspects of multiple sub-systems are hardcoded (reasoner configuration such as Gemini model, server configuration including network interface and port). These should be relocated to configuration files for improved maintainability and transferability.
* **Refactoring of index.mjs:** As discussed in section 5.2.1, decomposition of the main server file should be performed for improved readability, maintainability and separation of concerns.
* **Refactor coach class into singleton:** The initial design for the server was inadequate for the complexity it currently encompasses. A coach instance was designed to hold all relevant data (such as standard path parameters required) about a user. With the implementation of user services as an abstraction layer, a coach instance only holds the display name of a user, which does not require to be store in active memory between requests.

1. **Annotated Bibliography**

Below, a reader can find references to all technologies, articles and other resources involved in the development of the project as well as its report.

[1] Open API Specification, “OpenAPI Docs”, Updated: 24/10/2024. [Online]. Available: <https://swagger.io/specification/>. [Accessed 07/04/2025].

The OpenAPI specification documents a standard for developing application programming interfaces, specifically including the development of RESTful APIs. This resource was used to document endpoints for the project server.

[2] Bungie API, “Bungie API Docs”, Updated: 08/12/2024. [Online]. Available: <https://bungie-net.github.io/>. [Accessed 31/01/2025].

The Bungie API is used for accessing everything related to Destiny 2 including player Information, activities, weapons and confidential player data upon credentials. It is necessary for the software, allowing for tailored responses from the AI for the player.

[3] Amazon Web Services, “AWS Documentation”, Updated:. [Online]. Available: <https://docs.aws.amazon.com/>. [Accessed 01/02/2025].

Amazon Web Services provides numerous different cloud tools, such as databases, hosting services as well as AI services. It was briefly investigated as an option for hosting either the server or client.

[4] Microsoft Azure, “Azure Documentation”, Updated:. [Online]. Available: <https://learn.microsoft.com/en-us/azure/>. [Accessed 01/02/2025].

Akin to AWS, Azure provides numerous cloud, database and hosting services; but also provides services such as application design and architecture development, virtual machines and desktops as well as. It was briefly investigated as an option for hosting either the server or client.

[5] GitHub Pages, “GitHub Pages Documentation”, Updated:. [Online]. Available: <https://docs.github.com/en/pages>. [Accessed 05/02/2025].

GitHub Pages is a web hosting service provided by GitHub Incorporated. At the basic free level, this allows a service user to make a repository public and enable the repository to be built into an accessible website at [https://[userid].github.io/[repoName]/](https://[userid].github.io/%5brepoName%5d/). The free tier allows a maximum repository size of 2gb, and supports static, single-page applications (SPA). GitHub Pages was used to remotely host the client, as an accessible redirect address is required as part of Bungie`s OAuth2 flow.

[6] Waterfall Methodology, “Institute of Project Management”, Updated: 20/05/2022. [Online]. Available: https://instituteprojectmanagement.com/blog/waterfall-methodology/. [Accessed 10/02/2025].

The waterfall methodology is a plan based software development strategy focusing on initial planning of design, system architecture and any other relevant software components. It was briefly considered as a possible methodology for this project.

[7] Scrum Methodology, “Scrum.org”, Updated:. [Online]. Available: <https://www.scrum.org/resources/what-scrum-module>. [Accessed 13/02/2025].

Scrum is a software design methodology based on agile approaches to software development. It utilizes an iterative process (known as sprints) where certain tasks are assigned in spans generally not longer than two weeks. It was utilized as the development methodology for this project.

[8] Node.JS, “Node JS Home”, Updated:. [Online]. Available: <https://nodejs.org/en>. [Accessed 03/02/2025].

Node.JS is a free, open-source and cross platform JavaScript runtime environment, allowing a developer to build applications such as servers, scripts and command line tools on a local machine. In essence, the language is server-side JavaScript, interpreted on a local machine instead of in a browser. It was utilized as the core programming language for the project server, making extensive use of its event-driven I/O and promise based asynchronousity.

[9] PHP, “PHP Home”, Updated: 10/04/2025. [Online]. Available: <https://www.php.net/>. [Accessed 03/02/2025].

PHP is a general-purpose scripting language, but is optimized for use in web applications and development. It provides features such as Server-Side Rendering (SSR) of webpages, server-side logic as well as numerous other features. It was initially considered as an option for the server-side language, as it does feature libraries and features for creating a RESTful API.

[10] React Framework, “React Home”, Updated:. [Online]. Available: <https://react.dev/>. [Accessed 02/02/2025].

React is a mostly front-end framework for developing user-interfaces from pre-defined “components”. It was initially considered as a component of front-end development, providing an aspect of modernism to the client.

[11] TypeScript, “TypeScriptLang”, Updated:. [Online]. Available: <https://www.typescriptlang.org/>. [Accessed 03/02/2025].

TypeScript is a superset of JavaScript, introducing syntax for static typing. It is able to run anywhere JavaScript runs, as it can be converted easily into JavaScript. It was initially considered for both server-side (executed by the node interpreter) as well as for client-side code.

[12] JavaScript, “MDN Web Docs: JavaScript”, Updated:. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. [Accessed 03/02/2025].

JavaScript is a dynamic, interpreted/just-in-time compiled language used predominantly for execution of client code in a browser. It includes features such as runtime object construction, object introspection, as well as object-oriented, imperative and declarative paradigm styles. It is used as the main language of the project, used on the server-side through the Node.js interpreter, and makes up all client-side code.

[13] Vellum, “Vellum LLM Leaderboards”, Updated:. [Online]. Available: https://www.vellum.ai/llm-leaderboard. [Accessed 01/02/2025].

Vellum is an end-end AI development platform offering numerous tools for development teams to deploy applications with AI integration. It also offers a comprehensive comparison tool of available LLM models, using standardized benchmarks to compare performance in different aspects of LLM functionality. It was utilized initially to compare the performance of LLMs to help in selecting the best model for integration with this project.

[14] GPQA Diamond, “GPQA Diamond LLM Benchmark”, [Article]

[15] Humanities Last Exam, “arXiv:2501.14219”. [Citation]

[16] EcmaScript 6, “ES6”,

[17] Destiny Statistic Tracker, “DestinyStatsTracker”, [Online]. Available: https://destinytracker.com/. [Accessed 08/02/2025].

Example of an existing community tool.

[18] Node Express, “Express Library”. [Online]. Available: https://expressjs.com/. [Accessed 10/02/2025].

Express is utilized to configure and create a REST based API server, offering features including implementation of middleware.

[19] “Google Gemini SDK”, Available: https://ai.google.dev/gemini-api/docs/libraries,

Gemini is the integrated LLM for the project, utilizing its reasoning capabilities for coaching functionality.

[20] Node HTTPS Library, available: https://www.npmjs.com/package/node-https

This library is used to isntantiate an HTTPS server.

[21] Node Prompts, available: https://www.npmjs.com/package/prompts

Utilized for console input.

[22] MySQL2, available: https://www.npmjs.com/package/mysql2

Database integration.

[23] Node-Cron, available: https://www.npmjs.com/package/node-cron

Scheduling functions.

**Appendices**

**Appendix A – Use of Third-Part Code, Libraries and Generative AI**

This project includes work developed and published by other persons. Below, a list can be found of all examples of this.

**Libraries**

Several different libraries are used as part of the architecture of this project:

* The **Express library**[18] was utilized for high-level implementation and configuration of a RESTful API, allowing for the use of user-defined (and developed by other persons) middleware. The library is developed and managed by the OpenJS Foundation. The library was used without modification. The library is used under the MIT License.
* The **GoogleGenAI**[19] SDK for Node.jsis a library that allows high-level interaction with the Google Gemini API, and is used extensively in this project to fulfil generative objectives. The library is developed and managed by Google INC. The library is used under the Apache License 2.0.
* The **Node:HTTPS**[20] library is an inclusion in nodes standard library. It allows for high-level creation and configuration of an HTTPS server instance, core to the projects server operations and security. The library is developed and managed by the OpenJS foundation. The library is used under the MIT License.
* The **Prompts**[21] library is an external module that provides a high-level interface for creating synchronous console input events. The library is available under the MIT license.
* The **MySQL2**[22] library is an external module that provides asynchronous communication with a MySQL server instance. It is a continuation of MySQL Native implementing features such as promise wrappers, connection pooling as well as improved performance over already popular MySQL driver implementations. The library is available under the MIT license.
* The **Node-Cron**[23] library is a node module providing scheduling functionality. A developer can schedule Cron to call a specific function at a certain time. Used for weekly update through user services. The library is available under the ISC (Internet Systems Consortium) license.

**Generative AI**

Generative AI was used to assist in learning the core languages of the project, in place of traditional web searches. It acted as a quick reference tool for certain syntax, core language libraries, functionality and typical errors, as well as quick references to Bungie API endpoints.

All code, algorithms and design architectures are of my own work.

**Appendix B – Game Properties**

The following is a collation of the key game properties and mechanics that the project addresses. This accumulation of data was necessary in the development of key LLM capabilities in the project and the engineering of prompts.

**Weapon Properties**

The game includes a plethora of weapon types, including and limited to the following:

* Glaives
* Auto Rifles
* Sniper Rifles
* Trace Rifles
* Fusion Rifles
* Machine Guns
* Submachine Guns
* Sidearms
* Bows
* Swords
* Shotguns

Weapons in the game are acquired through activity rewards, known as a “weapon roll”. Each roll can come with slightly different statistics, but also very different perk assignments. Each weapon will have the following properties:

* **Weapon Type**: From the above list of types,
* **Weapon Perks**: Each weapon roll will have assigned specific perks, which determine its ability and effectiveness in different combat scenarios. See section on perks for more information.
* **Weapon Stats**: Each weapon has a value for each of its statistics. For each roll of the same weapon, a select amount of the weapons statistics will have slightly different values.

Each class of character (Warlock, Hunter, Titan) has vastly different capabilities. Subclass abilities are different for each and armours are specific to each class of character.

**Appendix C – Coaching Tool Requirements**

This appendix documents the individual requirements of each coaching function of the prototype. Each can be identified by the objective number specified in section 1.4.1.

**Objective 1**

Objective 1 must have the following properties:

* The response will consider the activity environments when suggesting a build, and note this where appropriate.
* The response may not suggest an optimal build, but notable cohesion to the activity and any applicable modifiers will be present.
* The response

**Appendix D – Target Platform Analysis**

This appendix documents a brief preliminary investigation carried out to investigate potential platforms for the project.

**Candidate Platforms**

Several different platforms are available for development. For a public application, there are 4 main platforms to consider as potential deployment targets:

* Windows Desktop (Minimum: 10/11)
* Web Application (Minimum: HTML5, CSS Level 3)
* Android Application (Minimum: SDK 24)
* IOS Application

**Candidate Platforms - Exclusions**

MacOS and Linux desktops are disregarded due to the game being unavailable on those devices natively, requiring streaming services (GeForce Now, Steam Deck, etc.) to play. Exact statistics for players by platform are not disclosed for Destiny 2, meaning it is hard to discern what percentage of active players utilize cloud gaming services.

From statistics published by Statista, the leading cloud services have a combined 33.2 million users (Xbox Cloud, NVIDIA GeForce Now). In contrast, Steam for desktop PC has had a peak upwards of 33 million players every month for the past year[4]STEAMDB, with a Newzoo report from 2023 finding 611 million console players in 2022 and upwards of 1.1 billion PC players. These numbers were expected to grow.

For these reasons, it is reasonable to exclude Linux and MacOS as potential targets due to their perceived lack of prevalence in the context of this project.

**Appendix E – Target Audience Identification**

This appendix covers the specification of the target audience the prototype aims to cater to with its coaching functionality.

**Identification**

The target audience for this product can be categorized into the following groups:

* **New Players (Playtime<=5 Hours):** The most inexperienced type of player, with only a brief introduction to the game and its character build screens. They will have a very basic understanding of the game space including:
  + Some of the activity types
  + Character and inventory screen
  + Player item vault
  + Character Class
  + Character Sub-Class
  + Character weapons and Armor
* **Beginner Players (50 Hours >= Playtime > 5 Hours):** A player who has become more familiar with the core game mechanics and the concept of a character build. They will understand the importance of utilizing certain items for more specific and challenging activities. They will have a beginner understanding of all items in the previous user category, as well as:
  + Sub-Class configuration (aspect, fragment, grenade, melee and class-ability)
  + Weapon modifications
  + Armor modifications
  + All activity types
* **Intermediary Players (300 Hours >= Playtime > 50 Hours):** A player who has completed some of the more challenging activities such as raids or dungeons, but may still not completely grasp the more fine-detailed aspects of the game such as individual elemental effects. They will however be aware of the existence of most status effects and the elementals which create them.

**Appendix F – Product Backlog**

This appendix is a minimal product backlog created as part of the chosen project methodology. Here you can find all user stories and the chain of sprints followed throughout the project lifecycle:

**User Stories**

**Sprints**

Sprint 1 (10/02/2025-24/02/2025):

* Initial bungie wrapper class version implemented to fetch basic profile data such as tokens. Initial client interface implemented with documented hosting service.

Sprint 2 (25/02/2025-10/03/2025):

Sprint 3 (11/03/2025-13/03/2025):

* Mini-sprint leading up to Mid-Project demonstration that implemented core features such as client page initialisation (login and authenticated state)