Department of Engineering and Information Science

Bachelor's Degree in
Computer Science

FINAL PROJECT

MAPS, ROLES, AND COMMUNITIES
Designing a web platform for the Bloons TD 6 online community

Supervisor
Maurizio Marchese

Graduate student
Riccardo Sartori

[illegible]

Academic year 2024/2025

Acknowledgements

I would like to thank Professor Maurizio Marchese for accepting my thesis idea and for being my supervisor, giving structure to my project.

I would also like to thank the Bloons TD 6 Maplist community for making me part of their project, trusting in my abilities, and using the platform I developed.

TABLE OF CONTENTS

Summary

This paper documents the design and development process of BTD6 Maplist, a web platform designed to meet the needs of the Bloons TD 6 gaming community. The project arose from a concrete need that emerged within one of the Discord channels dedicated to the game: the management of lists of difficult levels created by users and their associated completions was based on tools such as Google Sheets, which over time proved unsuitable for supporting the growth of the community and the complexity of the data to be managed.

The initial idea, which was to create a simple bot to facilitate certain repetitive actions, quickly evolved into a full-stack solution. The first step was to gather requirements: through discussions with moderators, level creators, and active players, the essential features and main problems to be solved were identified. This made it possible to define a set of priorities that guided the design of the architecture and interface.

The paper analyzes the main technologies used to build the platform, with a particular focus on the reasons behind each choice. The focus was on modern tools suitable for building a scalable, fast, and easy-to-maintain application. The strengths of the libraries and frameworks adopted for both the front-end and back-end were explored, along with the solutions chosen for data management and communication between system components.

The platform also implements authorization and authentication systems through external services, and permission management based on user roles on the server. Particular attention has been paid to the separation of responsibilities between the various components of the system and the management of communication between them, including through caching and containerization with Docker.

Finally, the paper includes a detailed analysis of the platform's use in the four months following its release. It also presents the main interactions offered by the site for each user category and planned future extensions, such as event management and the historical snapshot system.

1 Introduction to the Context

All young people like to play video games, and some manage to attract more attention than others. This is the case with Bloons TD 6, a *tower defense* game developed by Ninja Kiwi and released on June 14, 2018, which fans of the series had been eagerly awaiting for many years. Once released, initially on mobile devices and later on computers, consoles, and even Netflix, it achieved extraordinary success, climbing the charts of the most popular paid games for mobile devices and becoming a benchmark in its genre.

The game has managed to maintain the enthusiasm of its initial launch even seven years later, thanks to a business model based on regular updates, similar to that adopted by other titles such as Minecraft. The constant introduction of new content encourages players to stay up to date and return periodically to discover what's new. Although it features colorful and seemingly simple graphics and atmosphere, don't be fooled: it is, in effect, a strategy game, and many users find it difficult to complete the more challenging levels. It is a game that is easy to learn but difficult to master, the ideal type for creating and maintaining a community of passionate players.

The 39th update to the game was eagerly awaited by fans, as it introduced the ability for players to create their own custom levels. This new feature gave rise to numerous small communities, each with different objectives: some dedicated themselves to recreating levels from previous games in the series, others used the editor to create artistic compositions, while others sought to design extremely complex levels, with the aim of surpassing the difficulty of the official ones already present in the game.

This latter type of user-generated content is very common in all video games that include level creation tools. A well-known example is the Demon List, a ranking of the most difficult levels created by players of Geometry Dash, another mobile game known for its high level of difficulty .

1.1 The communities created

The communities mentioned above often gather on Discord, a real-time messaging platform originally promoted as a tool for gamers, but which today, thanks to its numerous features and excellent user experience, is also used by schools, online courses, support channels for technical projects, and, in general, all kinds of communities.

Despite its practicality, Discord has some weaknesses in the long term. By its nature, each community is closed and can only be accessed if you have an invitation link. If a user wanted to search for a community dedicated to user-created levels, they would have no way of finding it through a simple search: instead, they would have to find mentions on forums or other indexed external platforms. After discovering the community's existence, they would then have to search for a valid invitation link or ask a member directly. This process is cumbersome, and for this very reason, these communities rarely manage to grow significantly unless they are actively promoted by their members on more visible platforms such as Reddit or YouTube.

This lack of visibility means that many communities face gradual decline: without a constant flow of new members who can bring fresh ideas or replace those who leave, the very survival of the community is at risk. To ensure continuity, it is therefore essential to offer them greater visibility.

1.2 BTD6 Maplist

In the past, I have carried out several projects for communities related to the game Bloons TD 6, with the aim of automating certain activities within groups in which I was actively involved. The best-known project was PandeHelper, an application used to manage and provide information on recurring events in the game. It is currently used on over 100 Discord servers.

Thanks to my previous experience with the game, the founder of one of the communities dedicated to level creation contacted me to propose digitizing the management of the community and developing a small support bot. His community contained two separate lists of levels, each with rules dictating how they could be interacted with: adding levels and recording level completions.

Although the initial request was seemingly simple, I decided to propose the creation of a complete website, tailored to the needs of the community. This proposal is based on two main reasons:

1. Based on the specifications of the requested bot, its implementation would have been impossible without first migrating all the data to a database. This centralization would have required the creation of an interface for managing the data itself. Such an interface cannot be developed in an intuitive and user-friendly way using only the Discord API, at least not in a form that truly improves the community's user experience.
2. The creation of a *full-stack* application entirely designed by me, using modern technologies, represents an excellent opportunity to enrich my resume. Although this is a project related to a video game, the technical skills acquired and demonstrated are fully transferable to broader professional fields and projects of greater relevance to the business world.

2 Requirements analysis

Initially, the community used two Google Sheets to track all the levels in the lists, one per list, and to record the levels completed and the related strategies for each user. It was clear that the site would need to make the data from the Google Sheets available in a more presentable, indexable, and possibly Discord-integrated form.

To better define the requirements, I felt it necessary to interview the community owners and moderators in order to understand not only which operations were essential, but also which aspects could be improved from a usability perspective. In addition to the moderators, I also interviewed some particularly active users and to identify the most used features and evaluate how to make them more accessible.

2.1 Interview structure

All interviews had the following questions in common:

1. Are you a minor or an adult?
2. How long have you been playing Bloons TD 6?
3. How long have you been an active member of the BTD6 Maplist?
4. If BTD6 Maplist had its own website, what aspects would you want to see well presented, intuitive, and quick to use?
5. Do you prefer to create levels or play them?

Based on the answer to the previous question, I was able to categorize respondents as level creators and/or players. Depending on their category, I varied the questions asked. I asked level creators:

1. What would you change in the process of proposing a new level?
2. What would you change in the presentation of your levels?

To players, on the other hand:

1. What would you change in the process of submitting proof of level completion?
2. What was your experience like when you first interacted with the project?
    - How did the process seem to you from the perspective of a new user?

In addition, if the user was a moderator, I also asked the following questions:

1. What tasks do you perform most often in project management?
2. What aspect of the project presentation do you think could be improved?
3. What are the most difficult tasks in project management?

2.2 Analysis of user needs

Seventeen people were interviewed: the sample was almost equally divided between minors and adults, and predominantly male. Among the interviewees, seven were moderators (including owners). In terms of game roles, 11 were level players, two were creators, and four were both.

All users interviewed pointed out that the system based on two Google Sheets was too primitive and limited, suggesting that the site should not only integrate the data from the sheets, but also dedicate a page to each level, with a completion history for each user.
Level creators wanted an easy way to make all the levels they had created visible, while players wanted a system that would make it easier to view their completions.
Some players also reported that the rules for valid completions were unclear, leading them to upload invalid evidence that was then rejected by moderators, causing them frustration.

Moderators described managing completions on the current Google Sheets as very laborious: many operations were manual and prone to error. For example, for each level, it was necessary to report the most efficient completion (the one that used the least resources). It was not uncommon for them to forget to update this data or to attribute it to the wrong user.

Since the levels are extremely difficult, each game update requires them to be beaten again (*verified*) and the first player to do so to be recorded. This operation was also a source of errors, as moderators sometimes forgot to remove old verifications or add new ones.
The management of levels, on the other hand, does not present any particular critical issues.

In addition to interviews, I directly observed how users interacted with the project. There was no dedicated platform: levels and completions were uploaded via Discord chat, with two channels for each list, one for completions and one for levels (four channels in total).
Although the system was visually messy and impractical, it had one major strength: speed. Uploading a level or completion was instantaneous, just like sending a normal message.

Analyzing community activity, there were approximately 1,300 total members, with an average of 500 monthly visitors (users who open the community at least once a month), of which approximately 90 were communicators (users who send at least three messages per month). These statistics were collected using the analytical tools provided by Discord, which allow you to obtain detailed data on the activity of each server where you have privileges to view them.
Among these, about 60 were actively playing, with an average of 148 completions uploaded per month, while 48 users uploaded new levels, with an average of 50 uploads per month.
Although the game in question had about 20,000 simultaneous players on Steam (excluding mobile players), the community catered to a very specific niche interested in optimizing strategies on extremely complex levels.
As this was not a demographic with exponential growth, the scalability of the site was not a critical priority.

The following requirements were extracted from interviews and observations.

2.3 Functional requirements

- Integration with Discord: The system must integrate with Discord to allow for quick execution of the most basic and common operations without slowing down interaction.
- Level display: Each level must have a dedicated page with clear and accessible information.
- User and completion management: Users must be able to upload proof of completion; moderators must be able to approve or reject them, ensuring that all required data is uploaded.
- Each user must have a personal page showing completed levels, created levels, and statistics such as the number of levels beaten per list, points, and position in the rankings.
- List moderation: Moderators must be able to:
    - Add, edit, or delete levels
    - Approve, reject, edit, or manually enter completions
    - View, approve, and reject levels uploaded by users

2.4 Non-functional requirements

- Usability: The system interface must be simple and intuitive to facilitate navigation, management, and uploading of levels and completions.
- Clarity of rules: The system must make the rules governing the validity of completions and uploaded levels easy to understand.
- Data integrity: Moderation operations should be automated as much as possible. Fields dependent on other data should update automatically in the event of changes.
- Performance: The site must guarantee fast loading times, even on pages that require complex processing.

2.5 Constraints and assumptions

It has been assumed that every user has a Discord account, to simplify *onboarding*, given that the entire community is currently based on this platform.
This choice also allows us to take advantage of the one-to-one mapping between Discord accounts and site accounts, avoiding the need to implement a separate authentication system.

Given the relatively low number of active users and the low probability of exponential growth, the system can be hosted on a simple VPS, without the need for more complex cloud infrastructure.

Finally, although the community currently manages only two lists, the owners have expressed their intention to expand in the future to accommodate users interested in high-quality levels, without necessarily aiming for the most difficult levels. The design will therefore need to allow for the possibility of managing multiple lists.

3 Technologies used

Once the requirements had been analyzed, it was necessary to choose which technologies to use to create the site. The project was created with a stack of modern, cutting-edge technologies. The entire project is designed to be scalable and easily maintainable.

3.1 Front-end: React + Next.js

Although *front-end* technologies change very often to adapt to new paradigms and architectures, the React library continues to be the most popular due to its intuitive use and simplicity. Although many sites use React, it is not a complete technology like Angular or Vue. At its core, it is just a rendering library to make parts of pages responsive to state changes.

For this reason, many frameworks have been created that use React as their main building block, and I had to choose which one to use. Among all the options, I identified the two best: Vite and Next.js.

Vite is not so much a framework as a tool for developing in a fast and responsive environment. It is agnostic with regard to the technologies used, to the extent that it encourages developers to create frameworks using it as a basis, and it integrates very smoothly with React. It uses a tool called *esbuild* to compile code in real time and allow changes to be seen immediately during development. Applications created with Vite are mainly designed as *single-page apps,* where all rendering takes place on the client side after an initial request to the server. Although Vite supports *server-side rendering,* it can only be configured via third-party plugins.

Next.js, on the other hand, is the most modern framework for creating applications in React, also recommended by the library itself. It has many pre-installed features; one of the most convenient is the routing system. In Vite, routing must be configured manually using external libraries, while in Next.js it happens automatically simply by creating the files in the right place. Next.js has a folder called app in which each subfolder corresponds to a page on the site. For example, the folder app/level-lists/first-list will correspond to the page https://esempio.com/liste-livelli/prima-lista.

The most important feature that Next.js offers *out-of-the-box*, and which sets it apart from tools such as Vite, is its advanced support for *server-side rendering* in all its variants. Although it is possible to configure SSR in Vite as mentioned above, the process is less straightforward and requires more configuration. Vite remains an excellent choice for developing *single-page apps* where *rendering* takes place mainly on *the client* side*,* with all the advantages and disadvantages that this approach entails.

The main advantage of *client-side rendering* (CSR) is the speed of the application and navigation: normally, when navigating to a new page, the browser makes an HTTP request

request to load the HTML content of the new destination. With CSR, on the other hand, you always have at least one template for the page you want to navigate to, and you can display it immediately, giving the impression of high navigation speed. With this technique, data is loaded with an HTTP request that provides it in a more streamlined format such as JSON, thus increasing overall speed as there will be less data to transfer. While the data is being transferred, a loading screen can be displayed to provide visual feedback to the user, improving the user experience compared to what would happen in a non-SPA application, where the user would remain on the same page waiting for it to load or on a blank page.

Another advantage of this type of application is that, since the code needed to render the entire site is downloaded on the first request, it can be stored in the device's cache. This allows the site to be displayed even when the user is offline. This approach gave rise to the concept of *progressive web apps:* web applications that work partially even without a connection and can be installed locally, almost as if they were native applications, on both phones and computers. Although they offer fewer features than native apps, they are easy for anyone to install and do not require paid publication on proprietary stores such as the iOS App Store or the Android Google Play Store.

However, this type of approach comes at a cost. The user is forced to download the entire source code of the entire application, even when a significant portion of it is not actually used. This is not a big problem if you access the site via a home connection, but it can result in significant mobile data consumption if you connect from a smartphone or tablet on the go. At first glance, the unused code may seem negligible, but in reality, if the application grows and makes use of many frameworks and libraries, each with its own dependencies, the amount of data to be downloaded becomes significant.

Another significant disadvantage concerns *search engine optimization* (SEO). Although search engines are evolving, many *crawlers* still only analyze a page's initial HTTP response to decide how to index it. For a *server-rendered* application, this makes no difference, as the response already contains the complete HTML. In contrast, a *single-page app* returns only an HTML skeleton, delegating the rest of the content to JavaScript code, which not all crawlers can interpret correctly. This may be a minor issue for applications that do not require indexing, such as web apps for drawing or creating diagrams. But when it comes to landing pages or product pages, this limitation can severely penalize online visibility. Although search engines have started to execute JavaScript in order to better read SPAs, they still clearly prefer pages that are already rendered on the server side[1][2].

The fact that the *client* has to generate the entire HTML from JavaScript code also has a negative impact on performance, particularly on two key metrics: *first contentful paint* (FCP)[15] and *largest contentful paint* (LCP)[16]. These measure, respectively, the time taken from page load to the display of the first content, and the time taken to display the largest visual element. In an SPA, these times are typically longer, as the JavaScript must first be downloaded, analyzed, and executed to generate the interface.

All these problems are addressed and solved by Next.js, which is a modern framework for React development capable of sending server-side rendered pages to the client. Next.js supports different approaches to server-side rendering thanks to React *server components*, a type of component introduced in version 19 of React.

Server components are components in which HTML is generated directly on the server, and only the final result is included in the HTTP response sent to the client. This significantly improves search engine optimization, as crawlers immediately receive HTML that is already structured and readable. It also reduces both FCP and LCP, as the browser can start rendering the received HTML immediately, without waiting for JavaScript to be processed.

Another advantage of this approach concerns heavy libraries. For example, a *syntax highlighting* library can be heavy to load in terms of both time and system resources. Thanks to React *server components,* it is possible to completely avoid transferring this library to the client if it is used exclusively on the server. This further improves site performance by reducing the time required to *parse* JavaScript code and decreasing the amount of data transferred during navigation. Next.js, leveraging *server components,* offers three main modes of *server-side rendering.*

The first is *static site generation* (SSG), in which all pages are generated at build time. Through the optional *getStaticProps* function declared within a page, it is possible to determine all the necessary *props* (properties) and generate a static version of the page itself for each one[5]. This approach is the least flexible of those offered by *server-side rendering,* but it is very useful and used for static pages such as *landing pages.*

The second, more flexible mode is *incremental static generation* (ISG). Its power lies in the application of the *stale-while-revalidate* technique, which allows pages to be generated statically, as in SSG, but not at *build time*, rather on the first request. In practice, the server initially responds with a static version of the page and continues to do so for a configurable period. After this time, the page will be updated in the background upon the next request. During the update, the server will continue to return the previous version until the new one is ready. This system is useful for pages where it is not necessary to have perfectly updated content at all times.

The third mode is traditional *server-side rendering* (SSR), where all pages are generated dynamically on each request. This is the default behavior in Next.js and is used when it is essential to provide the user with constantly updated data. However, it is preferable to avoid it where not strictly necessary, as it involves greater latency: the server must perform all necessary operations (including any API calls) before it can return the HTML of the page. This type of rendering is similar to that used in the traditional web of the 2010s, for example with pages generated in PHP. However, the use of React allows for optimizations that improve the experience for both the developer and the user and search engines.

One of the key optimizations is offered by React's Suspense component. This allows you to "pause" the rendering of a component until it is ready, displaying fallback content in the meantime. It is particularly useful when, for example, a component needs to complete an asynchronous operation such as an HTTP request. In this case, the server can quickly return the main sections of the page while loading the suspended components in the background. This mechanism reduces user-perceived latency, improves *user experience, and* positively affects SEO performance. Once ready, suspended components will be *streamed to the client.* From a technical standpoint, the Suspense component immediately sends the fallback as soon as one of its children returns a Promise. Once the Promise is resolved, the content is streamed.

However, server components have a limitation: they cannot handle any kind of interactivity, which is the sole responsibility of the client. To handle reactive behavior or internal state, *client components* must be used[20]. These work exactly like classic React components, executed in the browser: they are dynamic and handle state and reactions to events through JavaScript. The crucial difference from React 19, however, is that *client components* are also initially rendered by the server when used with SSR, so they do not introduce any initial performance penalties[21]. During server-side rendering, each state will be considered in the initial value with which it was defined.

To make the interface responsive on *the client*, a step called *hydration* is required. The HTML returned by the server does not contain any information about the application's reactive logic: it is only static output. Along with the HTML, the client receives the code for *the client components* (including children and libraries used), as well as the entire React component tree for the page. Once both are received, React rebuilds the tree and associates each DOM node with its logical counterpart, allowing events and *handlers* to be bound correctly. At the end of this process, the page will be interactive just like in a classic React application, combining the advantages of *server-side rendering* with *client-side* reactivity.

The *hydration* process, however, is extremely delicate. It can fail if the HTML received does not match the expected React component tree. This generates a *hydration* error, which can compromise the correct functioning of some DOM subtrees. Such errors can also arise if the HTML is syntactically correct but semantically incorrect. A classic example is the unauthorized use of a *div* tag inside a p tag, which does not comply with the expected DOM structure.

3.2 Back-end: Python + PostgreSQL

For the database, I chose a relational solution, ideal for representing data that was originally organized in a Google Sheets table in a structured way, with strong relationships between users, maps created, completions, and other related entities. Specifically, I opted for PostgreSQL, a mature, *open-source* database with a very extensive ecosystem that provides numerous extensions. Over the years, PostgreSQL has established itself as one of the best SQL database management systems, thanks to its optimizations and numerous features, making it not only suitable for this project but also highly configurable, scalable, and efficient.

Although Next.js is a *full-stack* framework, I decided to separate the *back-end* from *the front-end* for several reasons.

The first is related to a design choice: to maintain a clear distinction between API and client. Integrating both into the same project can lead to greater complexity over time, especially if you take full advantage of the *full-stack* features offered by Next.js, which allow the use of functions designed for the API even in *server components*. Although this is convenient initially, as the project grows, it can compromise the separation of responsibilities, so I wanted to maintain a clear separation.

The second reason is technical: the *back-end* in Next.js is not designed to be persistent, and its architecture is based more on being a *function as a service*. This makes it difficult to implement global or singleton objects, which are essential for managing database connections, for example. In PostgreSQL, which imposes a maximum number of simultaneous connections[26], a single shared object is usually used to manage connection *pooling*. The lack of support for persistent objects in the serverless context of Next.js means that each request establishes a new connection, making the system inefficient and unscalable in the event of high traffic. This problem is much less prevalent in cloud-native solutions such as Firebase, which are designed to natively handle a high number of connections[27], but it is relevant for a single instance of PostgreSQL.

Not having chosen Next.js for the *back-end*, I opted for Python. Although not the most powerful language available, it is adequate for this project, as it will not have to handle particularly intense operations: the heaviest processing will in fact be delegated to the database as much as possible. Python stands out for its high productivity, syntactic clarity, and the availability of a vast ecosystem of libraries, currently over 650,000[28]. I considered two main frameworks, Django and Flask, but ultimately decided to write a custom framework using the aiohttp library directly.

This choice stems from the fact that both Django and Flask are designed as synchronous frameworks. Flask offers some support for asynchrony, but handles it suboptimally, creating a separate *event loop* for each asynchronous function instead of sharing a single one, thus negating the benefits of this approach[4]. aiohttp, on the other hand, is a natively asynchronous library that allows you to write efficient HTTP servers by taking full advantage of the potential of asynchrony. With minimal initial configuration, it was possible to create a simple, high-performance, tailor-made framework with all the features necessary to meet the application's requirements.

3.3 Testing

A fundamental aspect of any project is the testing phase. For the front-end, the Cypress framework was chosen, which, while providing both single component and end-to-end testing, does not support server component testing, given their relatively recent entry into the React ecosystem. Currently, no *front-end* testing framework seems to offer specific support for server components, so application validation will be done through *end-to-end* testing, performed on a production *build* of the site.

Regarding the *back-end*, although Python has the unittest library pre-installed, pytest was preferred as the testing framework due to its modernity compared to unittest, its larger and more active ecosystem, and superior support for asynchrony via community plug-ins.

4 Proposed architecture

The site was designed to be as easy to maintain as possible, which is why we chose a clear separation of responsibilities between the various components. We adopted the *model-controller-view* model, in which the API part handles the *controller* component, including all data authentication and validation logic.

However, part of *the controller* logic also resides in the database in the form of automatic actions. A representative example of this choice is the automatic saving of the name of the verifier of a level that has not yet been verified, when the completion uploaded by a user is rejected or not added manually. Another control logic implemented directly in the database concerns the calculation of rankings: to ensure their accuracy, these are calculated dynamically each time they are requested. Since this is a multi-step operation, we chose to expose this calculation via *views or* functions. For the rest, the database plays the role of *model* exclusively.

The last component of the model is represented by *the front-end* and integration with Discord, both of which act as *views*. Although there is minimal validation on *the view side*, its sole purpose is to improve the user experience, for example by providing more descriptive error messages. The entire application has been designed according to the principle that all checks must be performed on the server side, as the validity of messages received from the client cannot be trusted.

This architecture contributes significantly to the maintainability of the project: the clear separation of responsibilities makes it possible to know precisely where an action is performed and to quickly identify any errors. To further increase maintainability, communication between the various components has been structured so that only the *back-end* can interface directly with the database. Although it would be technically possible for other components, such as Discord integration, to establish a server-side connection and access data, this approach would make it difficult to accurately trace the origin of a transaction in the database. It would also require updating the code in multiple separate projects in the event of changes, increasing the risk of human error. By forcing the transition to a single entity responsible for data management, we ensure that in the event of problems, there is only one point to check and update interactions with the database.

4.1 Architecture components

The final architecture is divided into the following macro-components:

- The application clients are the browser, when the user accesses the site, or Discord, when using integration via the platform.
- The front-end includes both the site interface developed with Next.js and the interactions available on Discord. The latter are implemented in Python using the discord.py library, chosen both for my familiarity with the language and for the popularity of the library itself, second only to discord.js.
- The back-end is developed in Python using aiohttp.
- The database is PostgreSQL.
- Cloudflare is used as a CDN to improve the overall speed of the application and as a registrar for domain management.

Communication between components takes place via HTTPS protocol, with the exception of the database, which uses its own protocol. To improve performance and reduce response times, two levels of *caching* are implemented. The *stale-while-revalidate* strategy is used between *the front-end* and *back-end*, while a *cache-first* strategy is used in communication between *the back-end* and external services.

Each component of the application runs within its own Docker container, ensuring isolation, portability, and ease of management. External access to services is managed via nginx, configured as *a reverse proxy*, which correctly directs requests to the appropriate containers. Only the *front-end and back-end* containers are publicly exposed, while those related to the database and Discord integration remain accessible only within the internal network, as they do not require direct interaction with the outside world.

This modular division not only promotes the maintainability and scalability of the project, but also allows for more precise control over the responsibilities of each component and the interactions between them.

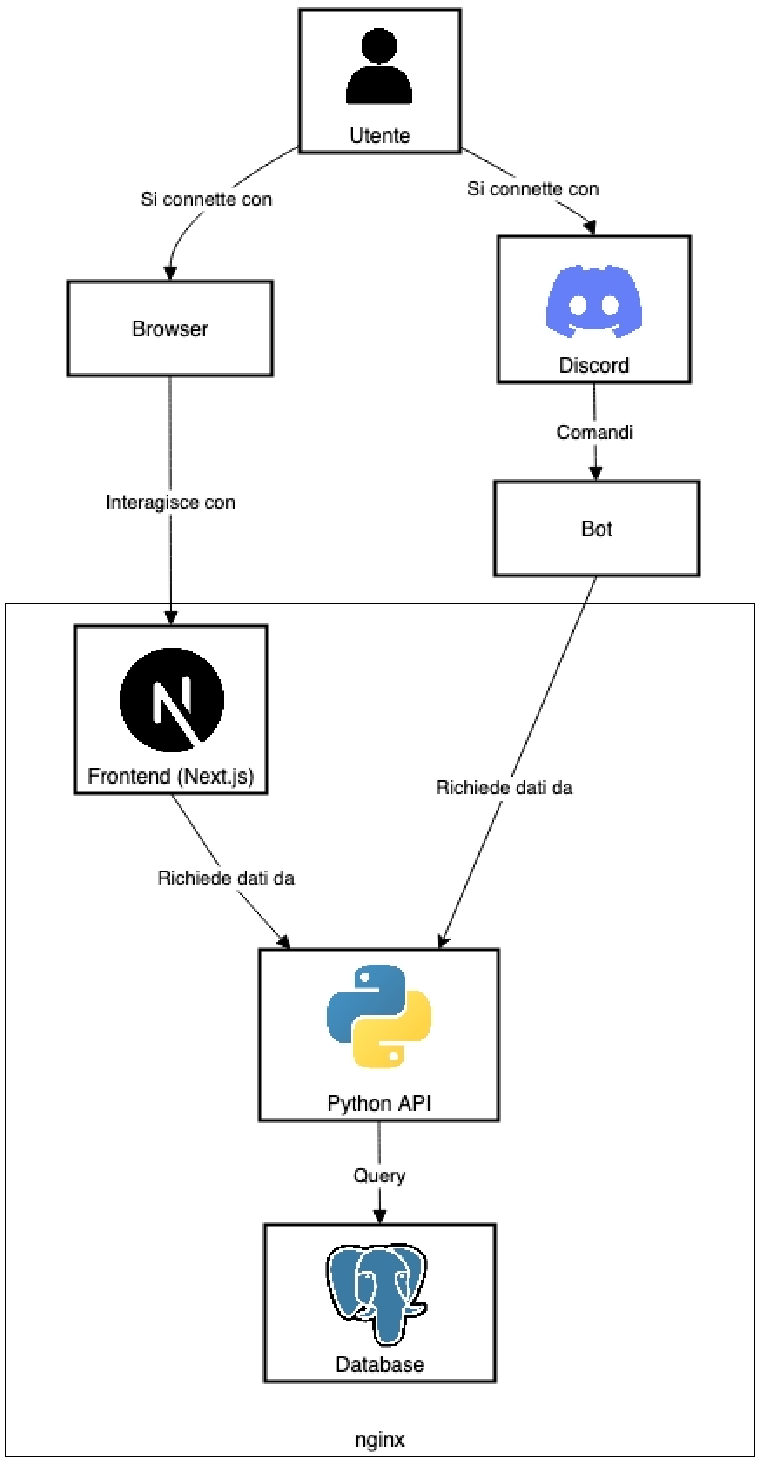


Fig.1 Project architecture

4.2 Security

As defined in the requirements analysis phase, the Discord API was adopted for user authentication. Users can log in to the website using their Discord credentials; once authenticated, an *access token* is returned that allows the application to obtain user data. Access authorization to resources is managed directly by the application, according to a role-based model: each unique user identifier (corresponding to their Discord ID) is associated with one or more roles, and each role has certain permissions linked to specific lists of resources.

When a user wants to access a protected resource, they include their *access token* in the HTTP request (specifically in *the* "Authorization" *header*). The API makes a request to Discord to obtain the user's identity, specifically their ID, and based on this check, determines what permissions they have and whether they can access the resource. To improve efficiency and avoid exceeding Discord's *rate limits*, responses to these requests are cached with a *time to live* of five minutes.

However, this process is not applicable to Discord integration. The normal authorization flow involves using an *access token* to validate the user's identity with the API, but in the case of integration, for example when a user uses a command directly within Discord, *the access token* is not available, nor are any other credentials. In these cases, Discord itself sends the user data. Since the request comes from the official platform, it can be assumed with certainty that the user's identity is authentic, thus eliminating the need for *back-end* authentication verification.

To handle these special cases, dedicated *endpoints* have been implemented, accessible exclusively from the Discord integration, which do not check user authentication but assume that it has already been performed by the integration. To ensure message security and sender identity integrity, the integration adds a cryptographic signature generated with its own private key to each request. The *backend* then verifies the validity of this signature using the corresponding public key and checks that the message content has not been altered. If successful, the request is accepted. Authorization remains the responsibility of the API: the integration communicates the user's identity, but cannot determine whether or not the user has the necessary permissions.

Finally, in terms of security, the application is protected against common attacks such as XSS, which is automatically mitigated by React thanks to input sanitization, except in cases where the *dangerouslySetInnerHTML* property is explicitly used (not used in this project). Every variable used by React is treated as a *TextNode,* a native JavaScript API in browsers that allows you to safely sanitize any text and insert it into the DOM. *SQL injection* attacks are also prevented through the use of parameterized queries, which prevent arbitrary code from being executed within database requests.

4.3 Data modeling

One of the features planned for the future will be the ability to view a *snapshot* of each list at any past *timestamp*, and the data has been modeled from the outset to support this requirement. To allow the display of a *snapshot* of the database at any arbitrary time, it was necessary to design some main tables, in particular those relating to lists and rankings, with a *versioning* system. The chosen implementation involves separating the data into two distinct tables: the first contains untracked fields (such as the name of a level), while the second contains all the information that needs to be versioned. There is a one-to-many relationship between these two tables.

The table that tracks versions includes a column for the creation *timestamp* of each record. Whenever the data for a resource needs to be updated, the existing row is not modified, but a new row is inserted with the new data and a more recent *timestamp*. This approach makes it easy to reconstruct the status of the resource at a specific point in time: simply filter the records with a creation date earlier than or equal to the desired time, and select the most recent record for each resource. PostgreSQL facilitates this operation with the *SELECT DISTINCT ON(col1, col2, ...)* command, which returns the first *record* for each unique combination of specified columns, combined with *ORDER BY (*by resource ID and creation date in descending order) to take advantage of any *b-tree* indexes.

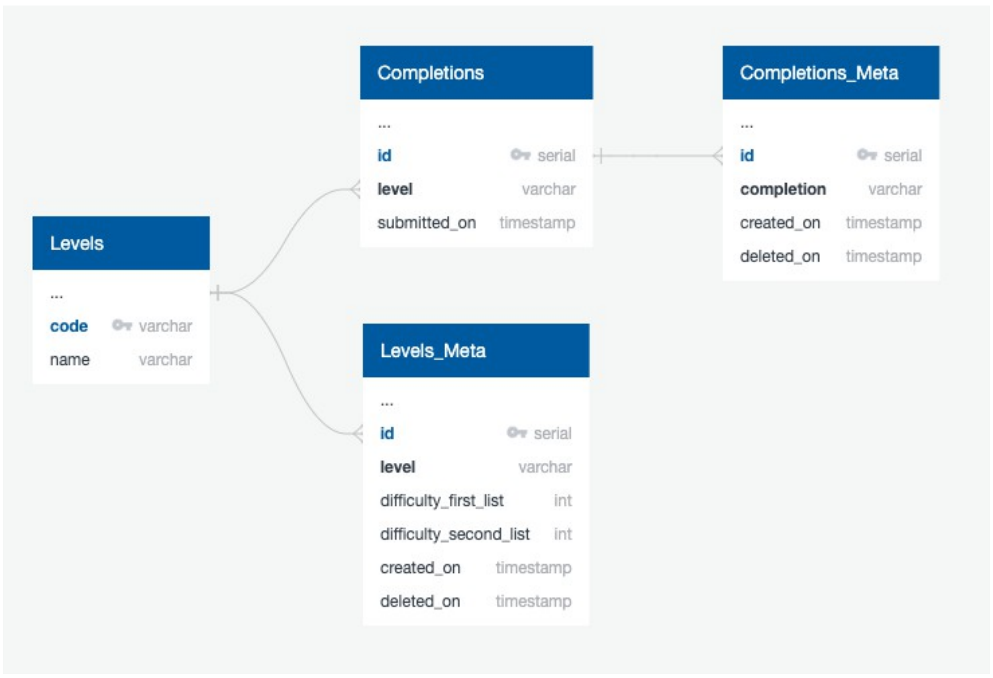


Fig.2 Minimal *versioning* schema for the *Levels* and *Completions* tables

In addition to these, the main tables include those for users and lists. Both are related to other fundamental entities, such as completions, levels, and level uploads. Furthermore, through a roles table, there is an indirect relationship between users and lists, which determines what permissions each user has in each list.

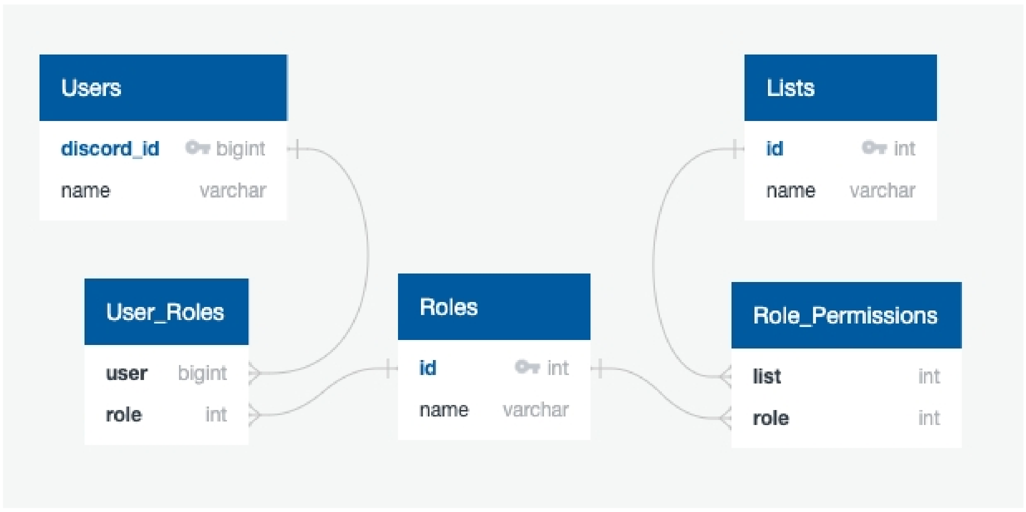


Fig.3 Relationship between users and lists through roles

When designing the schema, the principle of normalization was followed, with one exception: the table that manages the relationships between lists and levels. During the design phase, two alternatives were evaluated: a first normalized solution, which would have involved a many-to-many association table between list and level, and a second, denormalized solution, with custom columns for each list. Despite the advantages in terms of scalability and maintainability offered by a normalized structure, the second option was chosen.

The choice was motivated by the need for flexibility. Each list can have different rules for determining how levels are ordered or inserted. For example, the two initial lists in the project use simple integer values to define the position or difficulty of each level. However, in the future, other lists from different communities may require more complex criteria, such as conditional association with a foreign key that is only valid in certain cases. A generic, normalized structure would not allow this type of customization. The denormalized solution, although less elegant in theory, ensures greater flexibility in managing specific rules for each list.

4.4 Performance considerations

Although the project performs well overall in relation to its size, some components of the architecture could become potential bottlenecks over time.

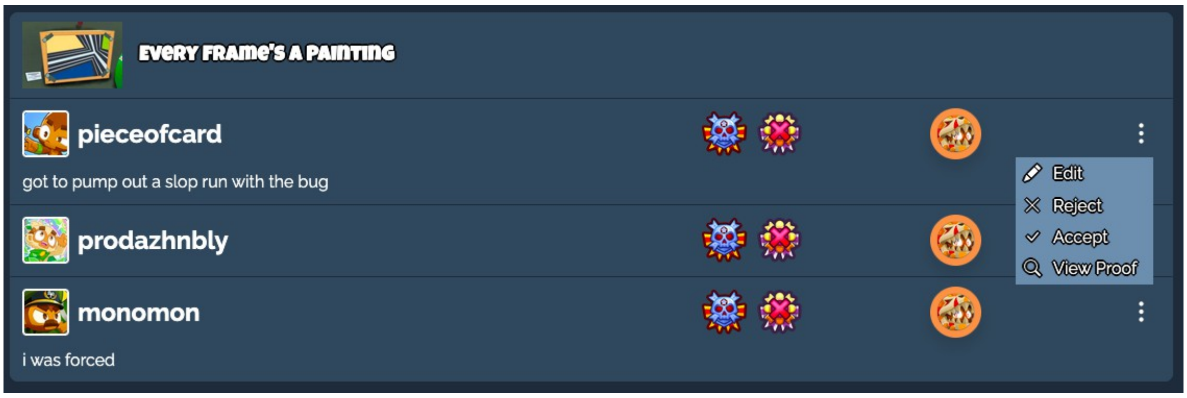
The first candidate in this regard is the database: currently, any change to the status of a level, completion, or configuration variable involves creating a new *record* rather than modifying the existing one. While this approach facilitates traceability and history management, it can also lead to the presence of numerous obsolete rows, which require explicit filtering in queries. Such filtering can become a costly operation, especially when it is a preliminary step to other complex operations, a clear example being the calculation of rankings. The problem can be addressed in several ways, which can also be combined. A first solution is to create a *B-tree* index on the level code and its creation date, making filtering and sorting on these fields much faster. An alternative is to create a materialized view that includes only the most recent data, although this solution loses the advantage offered by indexes. A compromise between the two options is to rely on indexes and, only when necessary, materialize a *common table expression* within a more complex query.

A second potential bottleneck is the Next.js server. This component handles both server-side rendering and image optimization, a feature included in the framework. Next.js provides a module that allows you to resize and convert any image to modern formats, such as WebP, to minimize the amount of data to download. It also requires you to specify the height and width of images in advance, eliminating cumulative layout shift, a metric that is also relevant for SEO. This metric measures how much a page's layout changes between the first render and the completion of loading: ideally, it should be zero. It is common to see high values for this metric when loading images without knowing their dimensions, as these can slightly change the layout. However, the image optimization service requires computational resources from the Next.js server, and in the event of a high number of simultaneous requests, it may become necessary to separate the server-side rendering and image optimization components to increase parallelism and improve the overall efficiency of the application.

Finally, the Python server does not appear to be a performance bottleneck, as its main role is limited to validating incoming requests and handling authentication and authorization.

4.5 Main interactions

The main interactions of this project were designed with the aim of simplifying the user experience for all categories involved: players, creators, and moderators. In particular, for moderators, the most frequent operation is the management of completions submitted by users, which can be accepted or rejected. To optimize this activity, a dedicated page has been created that collects all completions that each moderator has permission to intervene on.
Within this interface, each completion is accompanied by all relevant metadata and a drop-down menu that allows quick access to a series of common actions. Completions are also grouped by level, making the display clearer and more aesthetically pleasing.


Every Frame's a Painting

pieceofcard

got to pump out a slop run with the bug

prodazhnbly

monomon

i was forced

Fig.4 Completion management interface

For level creators, the main feature is obviously the ability to upload their own levels. To maximize accessibility, each list includes a button that allows the user to add a new level. Pressing the button displays a form in which it is possible to enter any notes related to the level and attach an image that certifies its completability.

For players, on the other hand, the main interaction consists of submitting their completions. On the page dedicated to each level, immediately below the main information (such as the creator's name or representative image), there is a button that allows the user to upload evidence proving that they have completed the level.

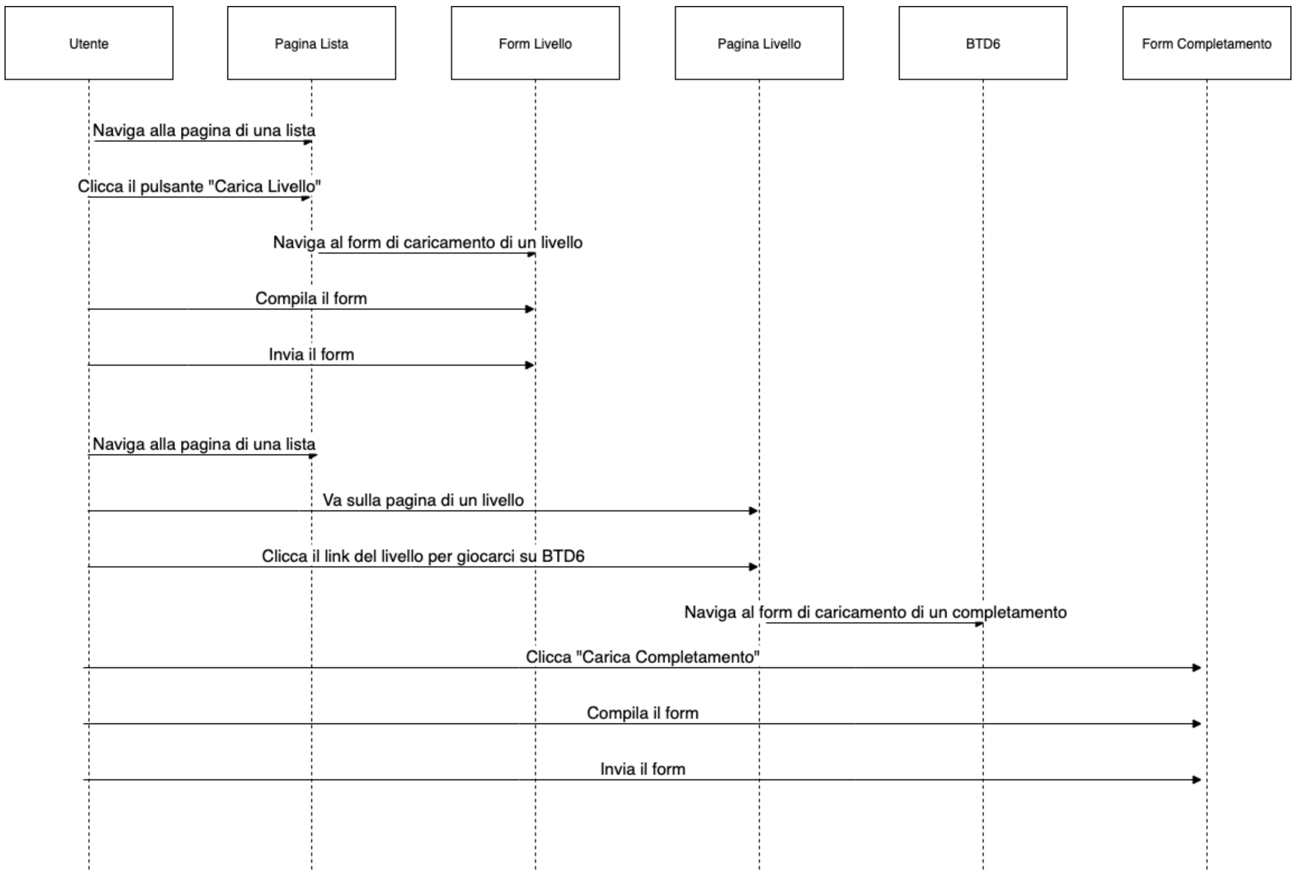

Utente
Pagina Lista
Form Livello
Pagina Livello
BTD6
Form Completamento
Naviga alla pagina di una lista
Clicca il pulsante "Carica Livello"
Naviga al form di caricamento di un livello
Compila il form
Invia il form
Naviga alla pagina di una lista
Va sulla pagina di un livello
Clicca il link del livello per giocarci su BTD6
Naviga al form di caricamento di un completamento
Clicca "Carica Completamento"
Compila il form
Invia il form

Fig.5 Interactions for uploading a level and a completion

5 Usage statistics

To analyze the use of the new platform, with the aim of improving user interaction, various data collection methods were employed.

The simplest method involved direct analysis of the application logs. In anticipation of a future expansion of the project that will allow users to view a snapshot of the lists and rankings at any time, every action performed by users is already tracked in the database along with a *timestamp*. This made it possible to obtain detailed and granular usage statistics.

The second method was based on the use of an *analytics* service integrated into the site. There are many options available, but the two most popular are Google Analytics and Plausible. Although Google Analytics offers an extremely comprehensive platform, it is too complex for the size and objectives of the project. In addition, the script required for integration into the site weighs in at around 100 kB, which is not insignificant. Added to this is the need for additional configurations to ensure GDPR compliance and user privacy protection.

For these reasons, Plausible was chosen, an open source alternative with more basic features but still sufficient to provide useful analytics. Plausible is designed to prioritize privacy: its integration does not require tracking *cookies* and does not record IP addresses. Furthermore, by adopting a *self-hosted* solution, the data collected is not shared with third parties, further improving the protection of user privacy. Finally, the Plausible script weighs only 1 kB, significantly less than Google Analytics and other similar solutions, with minimal impact on site performance.

The period considered for the final data analysis is from January 1, 2025, to April 1, 2025, i.e., four months after the initial release of the application in September. This time interval was chosen to observe user behavior and assess whether interaction with the site has actually improved compared to the initial system.

5.1 Traffic overview

The site averages about 55 unique visitors per day, each generating an average of 6 visits per session. This behavior reflects typical user flow: the user accesses the home page, navigates to the desired list page, and selects a level. On the level page, they find the relevant code, which can be used in the game to access it. Once the level is completed, the user returns to the site, goes to the page dedicated to uploading completions, and proceeds with the submission.

The *bounce rate* is quite low, at 25%, while the average session duration is around 6 minutes, which is higher than initial expectations. This could be due to the user's indecision about which level to play or the attractive presentation of the data. Before selecting the level to play, the user can view general statistics on other players' completions or rankings.

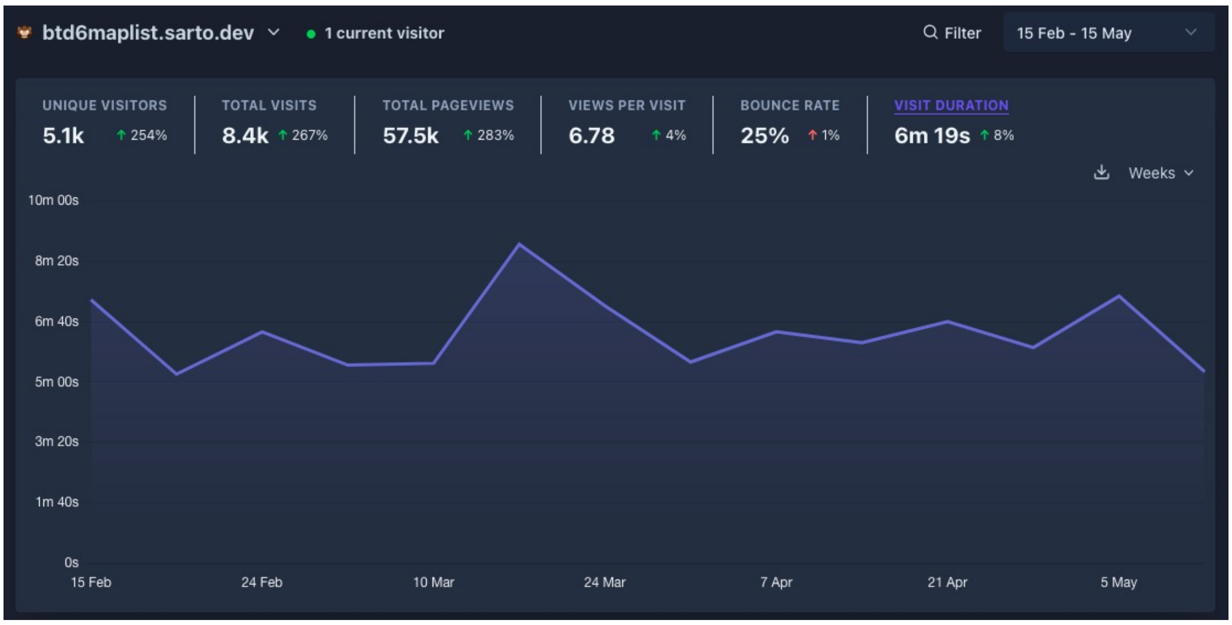


Fig.6 Plausible graphical interface showing various website statistics

Every month, around 70 users interact with the project: 60 of them upload completions, while 35 upload new levels, with some overlap between the two categories. This translates into a monthly average of 300 new completion uploads and 25 new level uploads, which are satisfactory numbers considering the high difficulty of the proposed levels and the strict criteria for accepting new levels. On average, 5 new levels are approved per month, selected from those submitted for both lists.

Comparing these data with those of the previous system, there has been a 50% increase in completion uploads, despite the number of players remaining unchanged. On the contrary, new level uploads decreased by 35% compared to the data collected during the site design phase. This decline is mainly due to the fact that, during the reference period, two map creation contests were organized: one promoted by the project community and one by an external community. Since support for contests is a planned but not yet implemented feature on the site, the entire submission upload process took place outside the platform. As a result, many level creators, busy participating in the contests, did not upload their levels in a way that could be tracked by the site.

The data also suggests that active contributors are heavily involved in the community: only 30% completed two levels or fewer during the analysis period, while 55% completed five levels or more.

5.2 Technical and demographic analysis

Data provided by Plausible shows that the countries with the highest number of visits to the site are English-speaking countries, particularly the United States, the United Kingdom, and Canada. This result is consistent with the demographics of the project, considering that the main language of communication is English. European countries also account for a significant proportion of users overall: although each European country individually contributes a limited number of visits, the continent as a whole accounts for a significant portion of traffic.

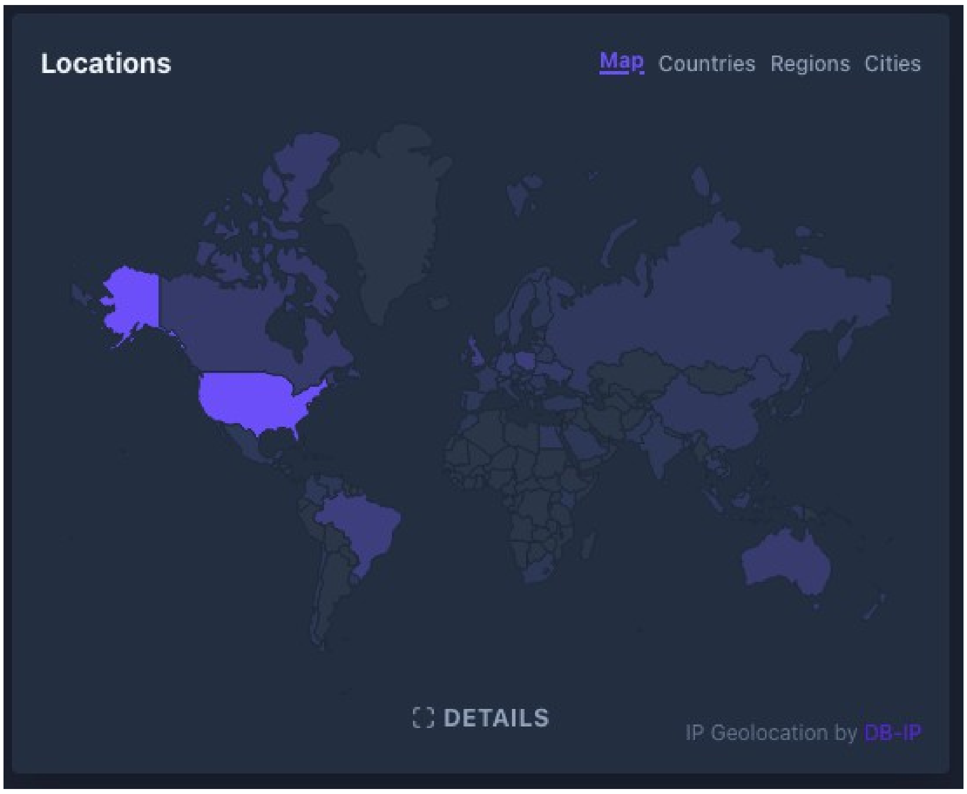
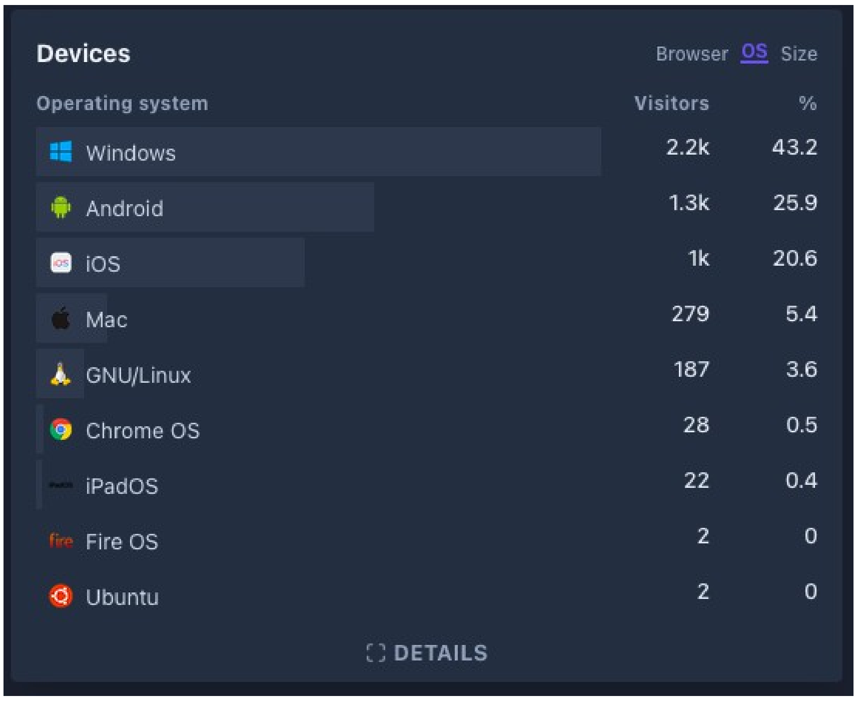


Fig.7 graph showing the distribution of site users

Mobile users account for about 45% of total traffic, while the other 55% comes almost entirely from desktops, with only 0.4% generated by tablets. These proportions reflect the demographics of the community: although the game is also available on smartphones, most users who tackle these levels tend to prefer using desktops, thanks to greater control and superior performance. In fact, the 50% share of desktop users is significantly higher than the average of around 35% found on other websites. On the level loading pages, 73% of accesses are from desktops, a significantly higher percentage than the average of 50% found for the rest of the site. This further confirms the tendency of the most engaged users to use the site from computers. Nevertheless, it is essential to ensure a high-quality user experience for the half of users who access the site from mobile devices, as for many this may be their first contact with the project, for example through links shared on mobile applications such as Reddit or Discord.


Devices

Browser OS Size

Operating system	Visitors	%
Windows	2.2k	43.2
Android	1.3k	25.9
iOS	1k	20.6
Mac	279	5.4
GNU/Linux	187	3.6
Chrome OS	28	0.5
iPadOS	22	0.4
Fire OS	2	0
Ubuntu	2	0

Fig.8 graph showing the operating systems used by users to visit the site

An interesting fact concerns the use of Discord integrations, which are very popular in the community for other aspects related to the game. Only 16% of users take advantage of these integrations to upload new levels or completions, while most prefer to use the dedicated pages directly on the site. Discord integrations, on the other hand, are used extensively by moderators. Each completion uploaded is automatically posted on Discord to be viewed by both moderators and other community members. Although moderators have the option to edit the data for each completion, in most cases no intervention is necessary, as the uploads are correct. To simplify this process, a button has been implemented via the Discord API that allows completions to be automatically approved without modification with a single click. This solution has proven effective in facilitating the work of moderators, who are already very active on the project's Discord channels and .

6 Future expansions

Although the project is complete and functional, other possible features have emerged that could improve the quality and usability of the site.

6.1 Level creation events

As mentioned in the analysis of usage statistics, communities regularly organize events dedicated to level creation. A preliminary study was therefore conducted on how these events are usually structured, in order to identify the features needed to support them within the project.

Each event will have a dedicated page where the creator can describe its details. This page will support the Markdown format for content formatting, a choice motivated by the simplicity and flexibility of the language: it is easy to use for the average user, particularly on Discord, where many of the Markdown formatting rules are already familiar, and it is easily convertible to HTML. To meet this requirement, a small content management system will be implemented. Each event will have a start and end date, will be associated with a list, and will be displayed on both the main page and the page of the list to which it belongs, once active.

The methods for users to submit levels will be configurable by administrators and event creators. Some events may allow only one level per participant to be uploaded, while others may allow unlimited submissions, but with restrictions on the frequency of uploads. In addition, it will also be necessary to manage the methods for selecting the winner: in some cases, the decision will be made by the staff, while in others it will be entrusted to the community vote once the submission period has ended. There are also hybrid methods, in which the community votes for the best levels and the staff chooses the winner from among them, or vice versa.

To make these events more accessible and encourage participation, the upload and voting process will also be integrated into the interface available on Discord.

6.2 Time machine

As mentioned above, both the database and the infrastructure have been designed to allow the status of lists, rankings, users, and, more generally, the entire site to be viewed at any point in the past. The last step is to make this functionality accessible via API and a graphical interface.

Users will be able to access a dedicated page where they can select a date of their choice to view the status of the project at that precise moment. From there, they will be able to browse the entire site in read-only mode, with a small button visible in the corner of the screen allowing them to restore the display to the current date and time.

6.3 Automatic identification of completion details

Although the process of uploading a completion is already very simple and the form is intuitive to use, there is room for further automation.

The game is based on completing levels by purchasing units that defeat enemies. Units can be bought and upgraded using money earned during the game. One of the criteria that awards extra points for completing a level is having spent less money than all other users for the same level. In the Bloons TD 6 community, this type of completion is known as *least cost CHIMPS (*abbreviated LCC). Currently, this data is entered manually by users or moderators who want to record the completion as a valid LCC.

Another criterion that can award additional points is the choice of units used: some levels are easier if you use certain units that are considered stronger. To prevent all users from completing a level using the same copied strategy, completing it without using these units guarantees extra points. As with LCCs, users must currently manually declare whether or not they used these units when entering the completion.

When uploading a completion, it is mandatory to provide a screenshot of the completed game, which moderators use to verify the validity of the completion. All the necessary information can be gleaned from this image: both the amount of money spent and the units actually used. A proposed long-term project involves the development of an artificial intelligence model capable of automatically detecting this information from a screenshot. Although this is only an idea at the moment, there are already several discussions underway on how to obtain the data needed to train such a model. The proposed solution is to organize one or more competitions in which participants will be required to upload a complete video of their game. In this way, a single completion would yield a large number of frames, which could be labeled manually and used to train the model.

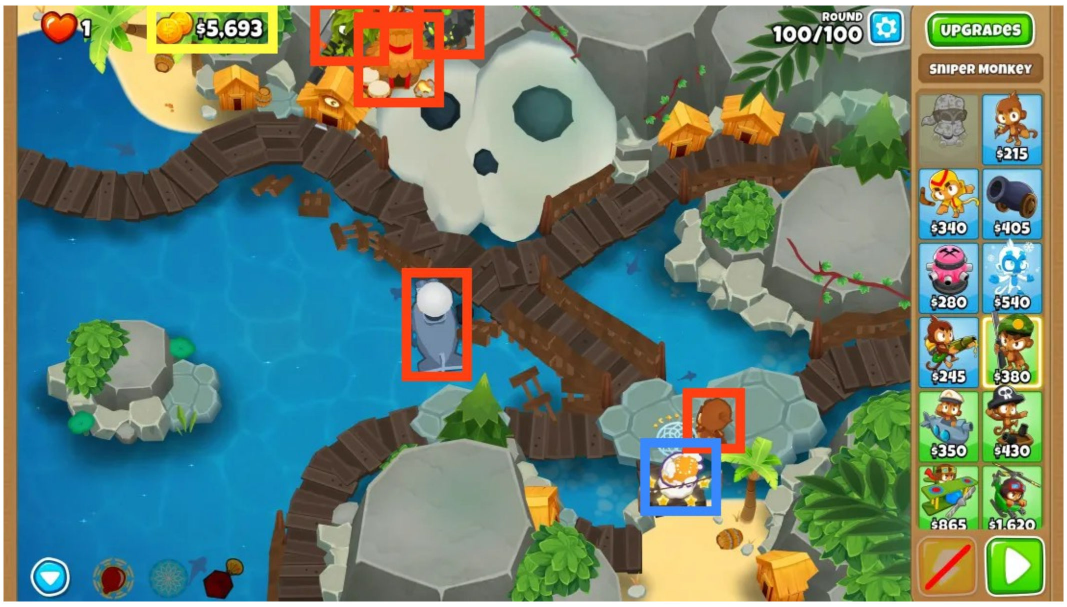


Fig.9 An image uploaded by a user to record completion. Labeled in yellow is the remaining money (from which the amount used can be calculated), in red are the normal units, and in blue are the prohibited ones.

7 Conclusion

The creation of BTD6 Maplist was a good learning experience, in which I was able to directly apply software engineering concepts to a case with real users and well-defined needs. The idea arose from a need that emerged within the community: the limitations of Discord management were becoming increasingly apparent, and there was room to build something more robust. What initially seemed like a simple bot became an opportunity to design a complete platform capable of responding to current needs.

From a technical standpoint, the project allowed me to consolidate many skills I had acquired over time and tackle new problems. I worked on designing a full-stack architecture, taking care of aspects such as security, database structure, APIs, and integration with external tools. Next.js gave me the opportunity to explore topics related to *server-side rendering and* the use of *server components,* while with Python I was able to build a back-end tailored to the project's needs. The entire development was accompanied by a testing phase, with appropriate tools for the front and back-end, in order to ensure a stable and maintainable base. From a systems perspective, I also had to deal with deployment pipelines and the management of nginx and Docker containers.

Beyond the technical aspects, the greatest satisfaction was seeing the platform actually being used by the community for which it was built. After launch, the number of completions uploaded increased, a sign that the new system has simplified the most common operations. Interaction with the interface proved to be better than previous methods, and use of the site replaced actions that previously went through Discord or Google Sheets.

In summary, BTD6 Maplist represented an opportunity to build something useful from a very specific context. It was a test bed for putting into practice an entire design and development process that goes beyond the technical side and also involves analyzing the real needs of users. Although there is room for improvement and new features, even this first phase alone has offered concrete insights into what it means to build software with a clear goal, and how valuable even a project developed for a small but active community can be.

Bibliography

[1] Ralf van Veen, Single-Page Applications (SPAs) in SEO, https://ralfvanveen.com/en/technical-seo/my-strategy-for-seo-for-single-page-applications-spas/
[2] Shahin Tamjidi, Comparison between SPA and MPA, Competition to get the best ranking on SEO
[3] https://developer.mozilla.org/en-US/docs/Web/API/Document/createTextNode
[4] https://github.com/pallets/flask/blob/a5f9742398c9429ef84ac8a57b0f3eb418394d9e/docs/async-await.rst#performance
[5] https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation
[6] https://react.dev/reference/rsc/server-components
[7] https://react.dev/reference/rsc/use-client
[8] https://steamdb.info/app/960090/charts/#3y (May 24, 2025)
[9] https://trends.stackoverflow.co/?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3 (07/06/2025)
[10] https://vite.dev/guide/philosophy.html#building-frameworks-on-top-of-vite (06/07/2025)
[11] https://vite.dev/guide/why.html (07/06/2025)
[12] https://react.dev/learn/creating-a-react-app#full-stack-frameworks (June 10, 2025)
[13] https://developer.mozilla.org/en-US/docs/Glossary/CSR (10/06/2025)
[14] https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (June 10, 2025)
[15] https://web.dev/fcp/ (10/06/2025)
[16] https://web.dev/lcp/ (June 10, 2025)
[17] https://nextjs.org/docs/pages/guides/incremental-static-regeneration (10/06/2025)
[18] https://react.dev/reference/react/Suspense#displaying-a-fallback-while-content-is-loading (07/20/2025)
[19] https://www.contentful.com/blog/what-is-react-suspense/ (July 20, 2025)
[20] https://react.dev/reference/rsc/server-components#adding-interactivity-to-server-components (07/20/2025)
[21] https://nextjs.org/docs/app/getting-started/server-and-client-components#on-the-server (07/20/2025)
[22] https://react.dev/reference/react-dom/client/hydrateRoot (07/20/2025)
[23] https://nextjs.org/docs/messages/react-hydration-error (07/20/2025)
[24] https://db-engines.com/en/ranking (07/20/2025)
[25] https://vercel.com/docs/functions#functions-lifecycle (07/20/2025)
[26] https://www.postgresql.org/docs/current/runtime-config-connection.html (07/20/2025)
[27] https://firebase.google.com/docs/database/usage/limits?hl=it (July 20, 2025)
[28] https://pypi.org/ (July 20, 2025)
[29] https://nextjs.org/docs/pages/guides/testing/cypress (July 20, 2025)
[30] https://www.jetbrains.com/guide/pytest/links/pytest-v-unittest/ (07/20/2025)
[31] https://pytest-asyncio.readthedocs.io/en/latest (July 20, 2025)