Atlas Systems
Technological Feasibility

October 24th, 2025

Project Sponsor: Morgan W. Boatman - Missions and Madness, Winter Communication LLC
Faculty Mentor: Ogonna Eli
Team Members:
John Zeledon (Team Lead)
Tristen Calder (Release Manager)
Hunter Beach (Architect)
Mitchell Morris (Recorder)

Table of Contents

1. Introduction

For the past few years, our capstone client, **Morgan Boatman**, has been running games of his creation, *Missions and Madness,* for the residents of Flagstaff. The game is a real-life adventure and training game that blends mental challenges, physical activity, and social interaction to build awareness and teamwork among users. Right now, the game is run with a booklet and some dice, similar to a tabletop game, but Morgan wants to improve and extend his game **by making a mobile app** to make the game more accessible to more people! Last year, he started up a capstone with our successor team, *Task Masters,* and together they built the backbone for the app; however, there is still technological work to be done before the app is complete.

The main issues identified by Morgan are the absence of GPS-based navigation, a rough user interface, a lack of deployment on the App Store, bugs, and the lack of a global database of Points of Interest (POIs) to start missions from. Once these issues are addressed, Morgan feels ready to release the app as a final product, completing the goal of making an app for *Missions and Madness*.

Since we want to be certain we can design a good product for Morgan, our team, *Atlas Systems*, is taking on the task of writing this **Tech Feasibility** document, to:

(1) Clearly lay out the technological challenges we need to overcome
(2) Analyze different options for technologies to choose from, and pick the best
(3) Offer examples on how this can be implemented into the app.

All of this document is for the greater purpose of offering the best application possible for our customer, as the more work we do ahead of time, ensuring we can solve the technological challenges, the better product we can make. To begin this document, we will start by defining the technological challenges we need to solve

2. Technological Challenges

In order to deliver the best version of *Missions and Madness* this year, we need to analyze our main technological challenges. While some choices from last year's *Task Master* Team, such as using ***Unity*** as the main game engine, still influence our project, our focus this year shifts toward improving and expanding the system. Specifically, our work centers on redesigning the Mission and Madness and mobile experience by implementing ***GPS-integrated*** mapping, which can be accessed through different ***Application Programming Interfaces (APIs)***, improving ***UI*** consistency, confirming the possibility of ***Deployment*** and ***Testing*** on multiple platforms and appstores, developing a scalable ***cloud-backed database*** for mission information, **a**nd the implementation of ***Points Of Interest(POIs)*** via a public ***Database***. In addition, we need to make the game accessible to new players to allow low-friction adoption of the application

The main technological challenges we've identified include:

GPS Integration:
We need GPS to actually tie the game to the real world to show the player where they are and where their goal is. Without it, the missions don't make sense. As we already know, we are working with ***Unity***, so we must find a GPS that works within it. The app must track players in real time, but also avoid draining the phone battery. Getting permissions and making it work on both Android and iPhone are the main things we have to solve.

API Management:
APIs are how the phone and map services will talk to each other, so they have to be simple and secure. If requests are slow or error-prone, the game will feel laggy or show wrong locations. Good API design keeps the gameplay responsive and makes future changes easier.

UI Design:
The UI should make the game straightforward for new players with clear labels, readable buttons, and a consistent look. It needs to work on different screen sizes and be easy to update as we iterate. A messy UI will confuse players and hurt retention.

Deployment:
We want actual users to install the app, so building and submitting to Google Play and the App Store matters. The release process must be repeatable and account for signing, privacy strings, and store rules. Doing this right avoids wasted time during review and keeps the project on schedule.

Testing:
Testing catches bugs before testers or players do. Using Unity's test tools plus a couple of real-device checks gives us good coverage without getting bogged down. Regular smoke tests make sure the location, maps, and the UI keep working after changes.

Cloud Database:
We need a cloud database so mission and location data can scale beyond a single device. A Postgres instance with geospatial support fits our needs and lets us run SQL queries for routing and POIs. It also prepares the app for more users and future features.

Database for Points of Interest:
Linking to POI sources (like Wikidata/Overpass) lets the game show interesting local places automatically, which can make the game spread beyond Flagstaff. It also means less manual data entry and more variety for players in different towns. We'll favor sources that give useful descriptions and that work in smaller cities like Flagstaff.

3. Technology Analysis

To accomplish a task such as this one, which requires the combination of the skill sets of multiple people and a wide range of different technologies, it is important to choose the best option at every step. Here, we will analyze all of our options for each category of the project and choose the technology that fits us and our project the best. Each technological challenge requires careful evaluation of potential frameworks, APIs, and libraries. The following subsections will analyze alternatives for each major area and provide a rationale for the final technology choices.

3.1 GPS Integration

Introduction:

Unity, being primarily a game engine, does not have built-in ***GPS*** services as robust as those on native mobile platforms. This introduces challenges such as obtaining permission to access location data, ensuring cross-platform compatibility (Android/iOS), and maintaining performance without draining battery life. To get a better understanding of the technical challenges we face, we will research how best to implement a map with a live display of several locations, including the user's location, which will be constantly changing, as well as the relevant window and map scale.

Desired Characteristics:

1. **Cross-Platform Support:** We want a GPS integration in Unity that can work on both Apple and Android phones, as we wish to deploy to everyone possible. It would also be ideal if it worked in the same way.
2. **Ease of Integration:** We want something that easily plugs into Unity and doesn't have much difficult learning. This would make development easier for our team as well as whatever team takes over after us.
3. **Accuracy and Performance:** As the play will be moving, we want the GPS to update quickly and accurately display where the player currently is, what direction they are going, and to what POI they are heading towards. This is a lot of data, and if it's confusing, players could get lost. It is also good to make sure the GPS doesn't strain the device battery.
4. **Licensing Cost:** We want something low-cost, as it would be more beneficial to Morgan in the long run.
5. **Community Support:** Anything with community support, to help us quickly understand how to best implement it, would be preferred.

Alternatives:

1. **Unity's Built-in LocationService:** This is Unity's native API for accessing device GPS coordinates. It is lightweight and built directly into Unity's Input. location class. It has been supported since early Unity versions and is frequently used in mobile location-based games and AR projects.
2. **AR Foundation with Geospatial APIs:** This combines Unity's AR Foundation with ARCore/ARKit Geospatial APIs. It provides high accuracy by fusing GPS with visual positioning. This method is more modern and powerful but requires newer devices and higher processing power.
3. **Third-Party Plugin:** Several open-source or low-cost plugins exist that wrap native Android/iOS location APIs for Unity. These typically add convenience functions (background tracking, error handling) and can be faster to implement.

Analysis:

To choose the best solution for our team, we tested how well each of the ***Alternatives*** satisfied our ***desired characteristics,*** and wrote the table below:

Characteristic	**Unity LocationService**	**AR Foundation Geospatial**	**Third-Party Plugin**
Cross-platform support	5	5	5
Ease of integration	5	2 (Complex setup)	5
Accuracy and performance	2 (GPS only)	5 (Visual + GPS fusion)	0
Licensing cost	5 (Free)	5 (Free)	3 (Some paid)
Community support	5	0	0
Total:	**4.4**	**3.4**	**2.6**

Table 1: Analysis of Unity GPS Technology

Chosen Approach:

Testing with small prototypes indicated that Unity's LocationService offers sufficient accuracy for a prototype with minimal setup, while AR Foundation's geospatial mode is overkill and not supported on all devices.

The team chose **Unity's LocationService** because it is free, stable, easy to use, and performs well enough for the project's GPS-based location tracking. Although it lacks the centimeter-level precision of AR Geospatial APIs, its simplicity and reliability make it ideal for our capstone phase.

Feasibility:

Initial feasibility testing will involve a small Unity demo that requests GPS permissions, retrieves latitude and longitude, and displays the coordinates in real time on screen. Later stages will expand this to transmit user coordinates to Geoapify and visualize nearby locations on a dynamic map. Testing will include accuracy validation, responsiveness when moving, and comparison across devices (Android and iPhone).

To test this, we'll make a small Unity demo that asks for GPS permissions, gets the user's coordinates, and displays them live on screen. Later, we'll connect that data to Geoapify so players can see nearby points of interest on a dynamic map. We'll also test it on both Android and iPhone to make sure accuracy and update speed are good enough for gameplay.

3.2 API Integration

Introduction:

A big challenge for our project is getting the app to figure out where the user is in real time and show that correctly inside Unity. Since *Missions and Madness* depend on players moving to real-world locations to complete missions, we need accurate GPS data that updates as they move. Unity makes this tricky because it's mainly a game engine, not a mobile navigation platform, so it doesn't have strong built-in GPS tools like Android or IOS do.

This means we have to solve a few issues: getting permission from users to access their location, keeping it working across both Android and IOS, and making sure it doesn't drain the battery too fast. Figuring out these problems early helps us design a GPS that fits the app's needs without adding unnecessary complexity.

Desired Characteristics:

To handle this, we plan to use Unity's built-in Location service feature, which gives us access to the phone's GPS coordinates directly from Unity. It's free, easy to set up, and works on both platforms, which makes it ideal for a student project like ours. The main things we need to implement are location permission handling, live position tracking, and integration with external API services, while keeping in mind battery life and performance constraints of a wide array of mobile devices.

These GPS features are what make the live map possible — the part of the app that lets players see their current position, routes, and mission areas in real time. Without reliable GPS, the game wouldn't really work the way it's supposed to.

When comparing different options, we looked at Unity's LocationService, AR Foundation with Geospatial APIs, and some third-party GPS plugins. AR Foundation was very accurate but overkill for our needs and required newer devices. Third-party plugins had extra features but added cost and setup time. Unity's built-in system ended up being the best fit because it's free, simple, and accurate enough for what we're doing.

We are looking for a few key qualities in a mapping API that will allow us to deliver a quality product to the client with minimal friction in the programming stage of the project.

1. **Licensing Cost:** We are looking for a solution that will be free or cheap on an ongoing basis, and if a completely free option is unavailable, we would like an API with a free tier for development and testing
2. **Detailed Accuracy:** The game is played in the real world, and we do not want to mislead people or send them anywhere they are unwelcome due to inaccuracy or low-quality maps
3. **Integration with Unity:** We are locked into Unity and need a technology that is not difficult or outright impossible to implement on Unity, or that will cause problems with deployment to mobile devices on multiple platforms

4. **Performance:** On mobile devices, we need to balance performance with battery life, keeping in mind some phones are substantially more powerful than others.
5. **Documentation:** We need an API with clear, concise documentation that is up to date and accurate

These features will directly support the “exploration and navigation” functionality of the final product. Next we looked into what alternatives there are to chose from for APIs to use.

Alternatives:

There are many different APIs, and through extensive research, here is what the team found.

1. **Geoapify API:** Geoapify provides APIs for geocoding, places, and routing, using OpenStreetMap data. It is free for limited use and has clear documentation. It’s commonly used for map visualization and route optimization in small projects.
2. **Google Maps Platform:** Google Maps offers powerful APIs for directions, places, and visualization. It’s mature, highly accurate, and widely supported, but its licensing cost increases rapidly with usage.
3. **Mapbox API:** Mapbox provides customizable vector maps and routing capabilities. It has strong Unity SDK support, but its free tier is limited. It’s often used in mobile apps with strong map visuals.

Analysis:

After finding alternatives, we analyzed how well each fit our Desired Characteristics, and sorted our findings below into a total average score.

Characteristic	**Geoapify**	**Google Maps**	**Mapbox**
Licensing Cost	5 (Free tier generous)	3 (Paid beyond small use)	0
Data accuracy	5	5	5
Integration with Unity	5 (Simple REST API)	5	0 (Unity SDK can be complex)
Performance	5	5	5
Documentation	5	5	5
Total:	**5**	**4.6**	**3**

Table 2: Analysis of GPS APIs Technology

Chosen Approach:

Our team leans towards **Geoapify** as it offers the right balance of cost, simplicity, and accuracy for educational and prototype use. Google Maps, while superior in global coverage and reliability, poses cost and key restrictions. Mapbox provides great visuals, but is heavier and more complex for Unity integration.

The team chose Geoapify API because it provides free access with strong documentation, making it ideal for quick integration and testing in Unity. It's sufficient for demonstrating dynamic map updates and routing features without requiring an expensive enterprise license.

Proving Feasibility:

Initial feasibility testing will involve sending HTTP requests from Unity to Geoapify's Places API using sample coordinates and displaying the results as text or markers in the Unity scene.
Next, the app will visualize routes using the Directions API and integrate map tiles for visualization. The demo will verify response speed, data accuracy, and how well the API integrates with the Unity rendering system.

3.3 UI Design

Introduction:

A core technological issue that Morgan himself has pointed us to focus on is building an initiative and a visually consistent user interface that communicates the gameplay flow to players and facilitates setup during play. The following are a few identified ways to do this.

Desired Characteristics:

1. **Explainability, Accessibility, and Tutorials:** We want to guide new players through the game, making the game explainable enough that new users can understand, as well as offering tutorial modes to help slowly introduce the player to the game flow.
2. **Modularity and Iterability:** We want to have things designed modular and iterable, so once we make one UI, we can quickly change and improve it without breaking many things
3. **Theme and Consistency:** We want a consistent UI across the whole game. Possibly with an artistic theme seen throughout the game.
4. **Performance Efficiency:** The UI must remain smooth without slowing gameplay or draining battery life.

Alternatives:

Within Unity, several different alternatives or options can be used to help with UI design and consistency:

1. **Unity UI (uGUI)**: One option is the built-in canvas-based UI system. It offers some basic tools likeanchors, layouts, and event systems. It is moderately hard to learn with some trial and error, and strugles to be modular, and consistant
2. **UI Toolkit (UITK)**: Another option Unity's newer web-style framework using XML and C#, offering reusable stylesheets and improved performance for complex UIs. It is somewhat easy to learn, and more modular than the prior option. It also can have good consistency, the main catch is it sometimes has slow performance
3. **Third-Party Frameworks (e.g., TextMeshPro)**: There are also Community packages that add animation tools, menu transitions, and theme management. They vary amongst themselves, but most are bulky and somewhat difficult to learn.
4. **Custom Prefab Library**: the last option is our team can manually define reusable prefabs (buttons, modals, labels) which enforce consistent styles. It will take a little learning, but offers a very modular, consistent and performant approach.

Analysis:

Seen below are our comparisons on how each ***Alternative*** a UI choice satisfied the ***Desired Characteristics*** requirements above.

Criteria	Unity UI	UI Toolkit	Third-Party	Prefabs
Explainability / Accessibility / Tutorials	3	4	3	3
Modularity and Iteratibility	3	5	4	5
Theme and Consistency	4	5	4	5
Performance Efficiency	5	3	2	5
Total	**3.75**	**4**	**3.25**	**4.5**

Table 3: Analysis of Unity UI Technology

While each had its pros and cons, Unity UI, UI Toolkit and Third-Party tools all had the same general score on how well they satisfy our characteristics, but Prefabs stood out on top. For satisfying our constraints, given we take a little extra time to set them up.

Chosen Approach:

Based on these results, our team plans to build a **Custom Prefab Library** within Unity's UI system. This allows us to design reusable, themed UI elements (buttons, text boxes, tutorial panels, and status indicators) that can be quickly instantiated and modified across scenes. This method balances Unity's native reliability with our need for modularity and visual consistency, while keeping us independent of third-party frameworks that may break with updates.

Proving Feasibility:

Seen below is a screenshot of a prior prefab from *Task Masters*. In Unity, you can orient things such as buttons or lists in a "blueprint" format, where you can change styles to make ALL objects of that type match throughout the program. We will likely aim for a more appealing UI, but this shows that a Prefab system is viable:

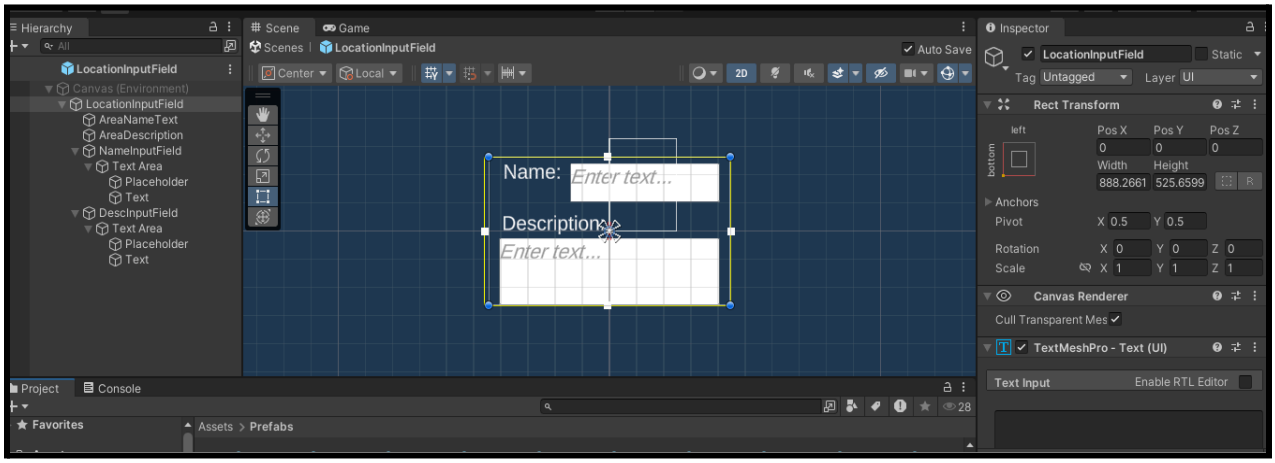

Fig. 1: A Prefab test within Unity.

3.4 Deployment

Introduction:

We would like to deliver a first-time mobile release to Apple App Store (iOS) and Google Play (Android) from a single Unity codebase.

Risks: store policy compliance (location/privacy), signing/provisioning, and timing of review/approvals. Additionally, iOS builds require macOS + Xcode, a potential obstacle, but we have team members with Mac computers, so we will perform iOS builds on those machines to overcome this constraint.

Desired characteristics:

1. Budget-friendly: no paid software or services required (unfortunately, not possible)
2. Deterministic: reproducible local builds; keystore/cert management documented.
3. Policy-compliant: location permission strings + privacy disclosures.
4. Timeline-safe: private beta path (invite links) before final release.

Alternatives:

1. **Local/manual release flow (chosen):** Unity builds on local computers; Xcode (iOS) + Play Console (Android); private betas via TestFlight / Play Closed Testing.
2. **Lightweight CI for build/test only:** optional GitHub Actions to run tests and produce artifacts; still upload manually.

3. **Cloud build services**: fastest to scale, but adds cost/overhead; not needed for this project.

Costs (hard requirements):

1. **Apple Developer Program: $99/year** (required to sign and submit iOS apps, and to use TestFlight)
2. **Google Play Developer account: $25 one-time** registration.

Characteristic	**Local/Manual**	**Light CI + Manual**	**Cloud Build**
Licensing / Fees	5	5	1 (tiers)
Setup Speed	5	0	0
Repeatability	4 (checklists)	5	5
Student Constraints	5	0	0
Policy Compliance Help	4 (manual checks)	5	4
Private Beta Support	5	5	5
Total:	**4.7**	**3.3**	**2.5**

Table 4: Analysis of Deployment Methods

Analysis:

1. **Local/manual** hits cost and simplicity goals; relies on our Mac users for iOS signing/Archive; easy to document.
2. **Light CI** helps catch regressions and produces zips, but it isn't necessary, especially because we are likely only deploying once or twice.
3. **Cloud** is overkill for our budget/timeline.

Chosen approach:

For Deployment, we have chosen to build the Unity game for Apple iOS and Google Android on local machines with no cloud integration whatsoever. For the actual deployment, we need to sign the apps and submit to those companies for review and potential release to app stores.

Proving Feasibility:

Unfortunately, feasibility cannot be proven in any meaningful way for this section. The only way to test or prove feasibility would be to try to deploy the app for real on both platforms, which would cost $125 and waste the time of Apple and Google employees, as well as cause a

multi-month-long hassle for us. Additionally, this is a “could have” item for our project and is unlikely to occur at all.

3.5 Testing

Introduction:

We need confidence that GPS, mapping, and routing work without committing to heavy, ongoing test infrastructure. Our goal is targeted, high-value tests that fit our bandwidth and budget.

For a testing system, we have a few requirements. The biggest requirement would be that the technologies we use are free and run locally. We would also like a solution that is easy and painless to set up and to use on an ongoing basis. Of course, we are not looking to compromise on effectiveness or quality, and the solutions must also either test our API or as a minimum spec, not interfere.

Desired Characteristics:

We need a few key attributes in a testing technology or testing pipeline. First and foremost is licensing cost, we would like the testing to be monetarily free as well as low time cost on an ongoing basis. We would also like something that runs local and offline as that simplifies testing and insulates us from issues such as AWS or other cloud outages. We need a solution that is easy to setup as well as easy to maintain on an ongoing basis. However, we will not sacrifice quality, we need a realistic and effective set of tests that work well with Unity and our chosen APIs.
Alternatives:

1. **Unity Test Framework (UTF) only (chosen):** Edit Mode (pure C# logic) + Play Mode (engine/coroutines).
2. **UTF + tiny device smoke tests:** a handful of manual checks on one Android + one iPhone to ensure no major bugs exist before further testing.
3. **Full UI/E2E automation:** not feasible for this project (too expensive and complex for our needs).

Item	UTF only	UTF + AltTester	Full UI/E2E
Licensing cost	5 (free)	5 (free)	3 (free, high time cost)
Local / runs offline	5	5	3 (partial)
Setup speed	5 (fast)	3 (moderate)	1 (slow)
Maintenance burden	4 (low)	2 (medium; selectors drift)	1 (high)
Unity engine fit	5 (strong)	4 (good)	1 (fragile)

UI/device realism	2(limited)	4 (good)	5 (excellent)
HTTP/GPS mockability	5	5	5
Total:	**4.4**	**4**	**2.7**

Table 5: Analysis of Testing Technology
3 technology options compared across all desired characteristics with a 1-5 ranking and and average for easy decision making

Analysis

1. **UTF** gives the best ROI: fast Edit-mode tests for coordinate math, request building, and parsing, Play-mode for marker rendering/rate-limit backoff.
2. **Device smoke tests** catch permission and rendering bugs. We will keep it small and perform smoke tests on physical devices after a major change in feature implementation
3. **Full UI automation** is out of scope.

Chosen approach:

UTF-centric testing plus minimal device smoke testing broad device testing or long-running automation is overkill and unfeasible for us.

Proving feasibility:

To prove feasibility for program testing, we will test-run the testing pipeline and perform edit-mode tests, play-mode tests, and then practice exporting to physical Android and iOS devices and ensure there are no severe issues that would create hiccups for the users or any issues when moving from simulated devices to physical ones.

3.6 Cloud Database

Introduction

Our project needs a cloud-hosted database that's affordable, secure, and easy for a small team to manage. The database has to handle geospatial queries for missions and locations, connect smoothly with our Node and Express backend, and scale as more users test the app. After looking at different options, we decided that running PostgreSQL on a Hostwinds VPS makes the most sense. It gives us full control over how the database runs, keeps costs predictable, and matches the tech stack we're already using. We plan to set it up using Docker with regular backups, simple monitoring, and a firewall for protection. This setup should be reliable while also giving us a chance to learn about managing real infrastructure.

Desired Characteristics

The main goals for this setup are keeping costs low, maintaining full access to configurations, and making sure the system stays secure. Having root access means we can install extensions, view logs, and tune performance as needed. Security is a big part of the plan too—we'll use private networking, TLS encryption, and role-based permissions to protect user data. The system also needs to perform well on SSD storage and have the option to scale up if the app grows. Another important point is that PostgreSQL supports tools like PostGIS and pgRouting, which can help us add map and routing features later on without needing to change databases.

Alternatives

Before choosing Hostwinds, we looked at other hosting options. AWS RDS was very reliable but too expensive and complicated for a student project. Services like Render, Railway, and DigitalOcean were simpler but didn't allow enough control over extensions, and their free tiers shut off when inactive. Firebase and Firestore didn't work well because they use a NoSQL model, which doesn't fit our relational and geospatial data. Supabase had some great developer tools but higher costs and potential platform lock-in. In the end, Hostwinds gave us the best mix of control, price, and flexibility while still letting us learn important DevOps and database skills.

Feasibility

We plan to run an Ubuntu VPS with Dockerized PostgreSQL 16 on SSD storage. A firewall will only allow SSH access from team IPs and Postgres connections from our app server. Backups will run nightly with pg_dump, and we'll take weekly snapshots that can be restored if needed. For monitoring, we'll use lightweight tools like Uptime Kuma or Node Exporter. Database migrations will be handled using Prisma or Knex, and we'll create standard tables for users, missions, locations, and submissions to keep everything organized.

To make sure the system is stable, we'll bind Postgres to a private interface, use SSH keys for access, and limit user roles. We'll test performance by simulating around 100–300 users and keeping response times under 250 milliseconds. Backup and restore drills will help us confirm data safety, and we'll verify that ports and permissions are properly configured.

Overall, Hostwinds gives us predictable pricing, complete control, and a realistic setup we can learn from. It's simple to manage but still powerful enough for our app's mission and location data. Once it's set up, we'll connect the app, test performance, and make any adjustments needed before beta testing.

3.7 Database for Points of Interest

Introduction:

The final core feature of *Missions and Madness* is the ability to automatically populate the in-game map with real-world Points of Interest (POIs), which can be anything from historical statues to interesting stores to nature trails. The main idea behind a POI is that it's something worth going to, that the player can gain joy from exploring and learning about, to engage with the world and community around them. In prior versions, there was a static database; however, Morgan has noted wanting to scale this elsewhere and anywhere.

The core challenge for our team is finding an external POI data source that is easy to query, relevant to both tourists and locals, and reliable across small and large towns alike, returning descriptive information on as many points as possible. The chosen solution should also be developer-friendly, to make our lives easier, and be integrable with Unity. It would also be good to have the solution be free for our project to reduce the strain on Morgan.

Desired Characteristics:

Based on the points we just spoke on in the introduction, an ideal POI API for *Missions and Madness* should meet the following desired characteristics:

1. **Query Ease:**
   The API should support queries that are as developer-friendly as possible so our team can best understand experimenting and writing with them for when players want to adjust their searches for POIs (i.e., different radii, looking for different attractions). Readability and simplicity also go far in long-term maintenance after our time on this project

2. **Many Points Available:**
   The Database must include a sufficient number of points across a wide variety of locations and contexts. A good way to test this is to use a smaller town like Flagstaff as a baseline, realizing there would be more points in larger cities. So, if we can ensure a database works with Flagstaff, it’s a step in the right direction.

3. **Relevant Points:**
   The data should include places that are meaningful for camplay, such as scenic viewpoints, museums, historical landmarks, and parks, rather than arbitrary business listings. This requirement could change in towns smaller than Flagstaff, as the POIs would become even more limited; however, we will still keep in mind looking for content that appeals to both new visitors and residents.

4. **Point Information:**
   Ideally, the API should provide additional information or metadata on the points (i.e., description, or history) in order to help with the game's educational aspects, where the players can learn more about the local history or culture.

5. **Financial Ease:**
   For future product sustainability, the API must be free for educational and small-scale production use, without strict rate limits or paid licensing requirements.

6. **Unity Integration:**
   Finally, the API should be integratable with Unity, likely through its HTTP request system, which would allow us as developers to *get* the POI information into Unity itself. This part is the most important, as it is essential for dynamically populating paps and generating in-game missions in real time. Considering this is the final step and a notable amount of work, we will cover it in ***Feasibility*** as it's only really realistic to test it on databases we feel confident in using. It will still be shown on the table, but will be covered in Section ***3.7***.

Alternatives:

Through our team's in-depth research, we identified four databases that meet our desired characteristics. These are **OverpassTurbo, Wikidata**, **OpenPlaques**, and **Geoapify**. It's not feasible to describe each, as they all do the same thing (i.e., retrieve points of interest in a JSON format); instead, it's best to compare them through the lens of An***alysis.***

Analysis:

Seen below are the ***Analyses*** of how well our ***Alternatives*** satisfy our ***Desired Requirements.***

Criteria	**OverpassTurbo**	**WikiData**	**OpenPlaques**	**Geoapify**
Query Ease	3	5	0	3
Many Points Available	5	4	0	2
Relevant Points	4	5	3	3
Point Information	3	5	4	2
Finacial Ease	5	5	5	5
Unity Integration	5	5	0	0
Total Feasibility Score:	**4.16**	**4.84**	**2**	**2.5**

Table 6: Analysis of POI Database Technology

1. **OverpassTurbo:**
   OverpassTurbo has a logical syntax; however, the queries have a tendency to fail if they return too much information, which makes testing an issue. It has a *Massive* number of points returned in Flagstaff, 28000; however, many of them are lackluster (i.e., a bunch, a bathroom, a security camera). There is also limited point information. Financially, there are no limits, though there could be rate limits to keep in mind. Overall, our team felt this was a reasonable database, so we tested it in Unity in the ***Feasibility*** section.

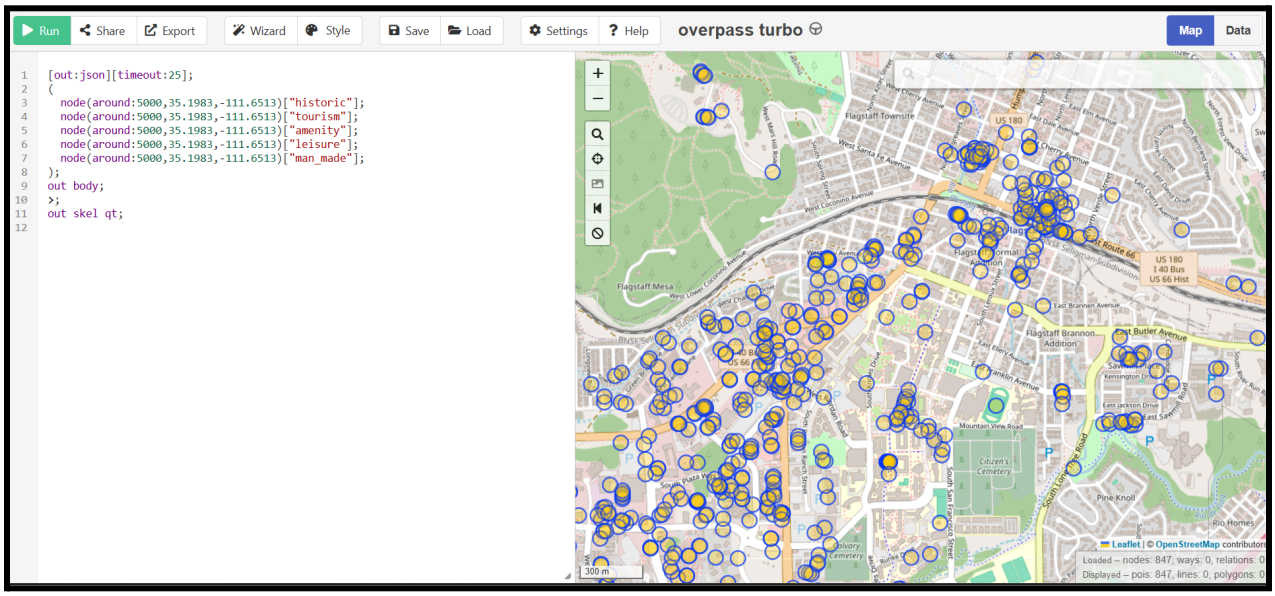

Fig. 2: A preliminary test of OverpassTurbo

2. **Wikidata:**
   Wikidata also has a logical syntax, and it's simple to limit queries so they don't fail or repeat values. It offers a good number of points in Flagstaff, though notably fewer than ***OverpassTurbo***, still enough for a small city. Its points are deeply relevant to the type of POI Morgan desires (i.e., historical, architectural learning points), and they have a fair amount of information with some images and even Wikipedia pages connected to 50% of the data points! Financially, there are no limits, though there could be rate limits to keep in mind. Overall, our team felt this was a reasonable database, so we tested it in Unity in the ***Feasibility*** section.

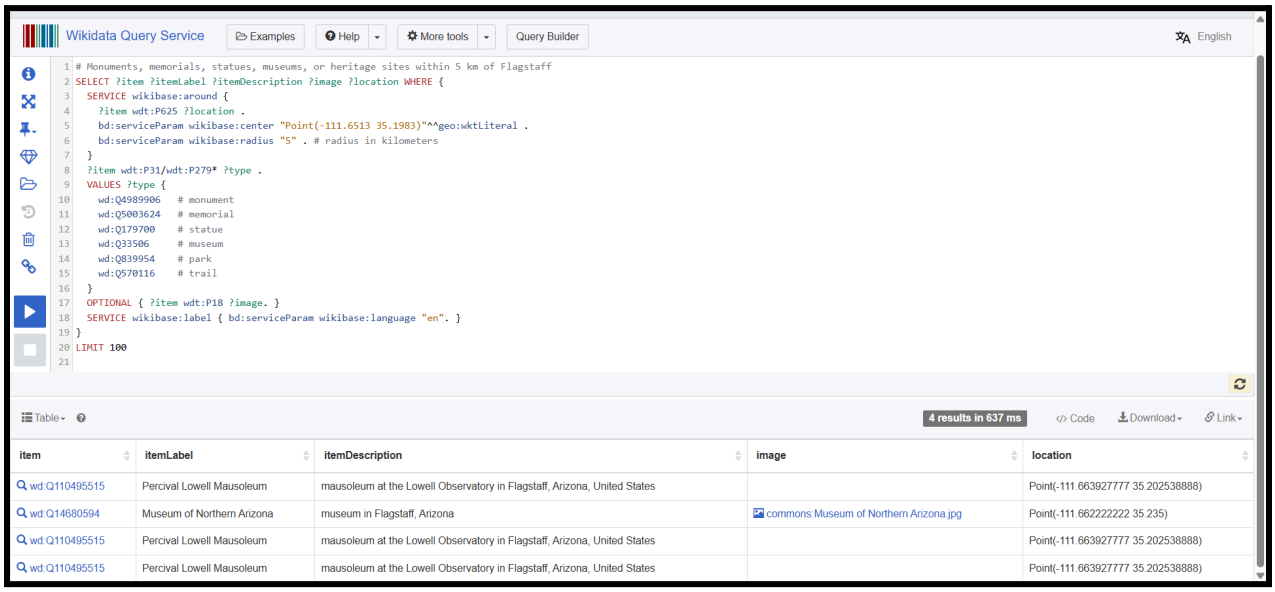

Fig. 3: A preliminary test of Wikidata

3. **OpenPlaques:**
   OpenPlaques has deeply limited filters, with no in built query system to sort data. While it was advertised to have points all over the US, there actually seem to be none in Flagstaff. This database features mainly plaques that are of interest to Morgan, though it is limited. It offers good historical information on the plaque. Financially, there are no limits. Overall, for completely failing the first two ***Desired Characteristics***, this data source is *not feasible.*

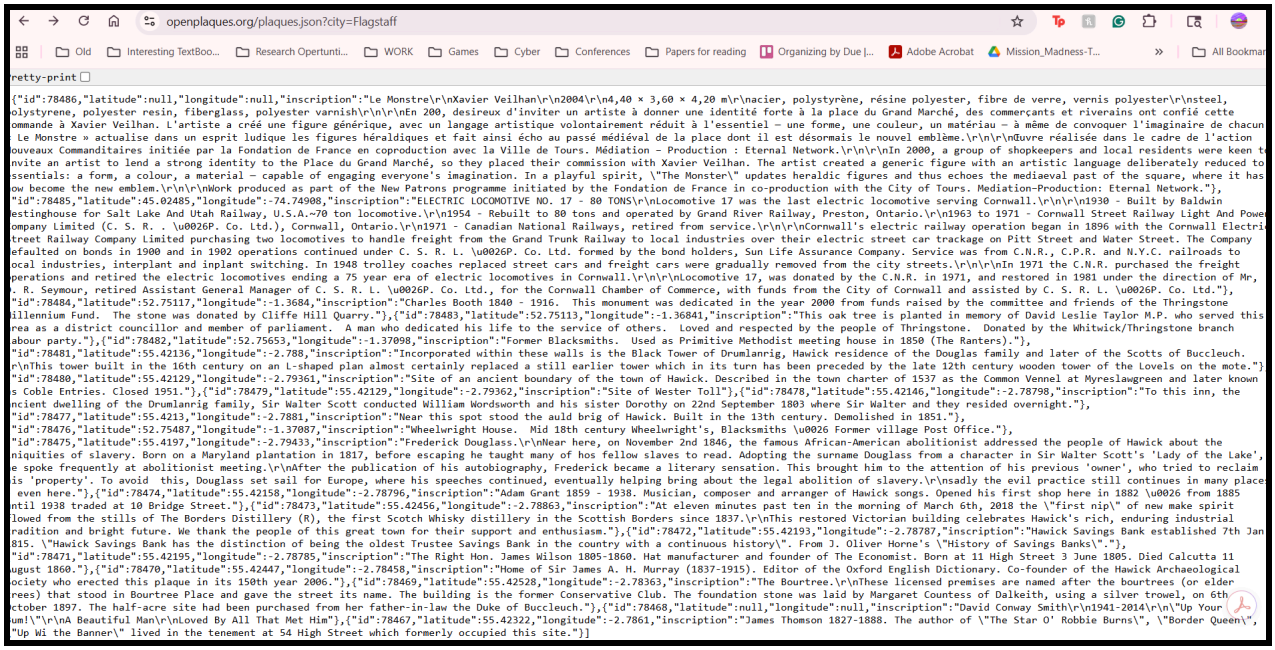

Fig. 4: A preliminary test of OpenPlaques

4. **Geoapify:**

Geoapify has somewhat of a limited query section, which was difficult for our developers to use. We also found that the number of points, the relevance of points, and information on points in the Flagstaff area were all lacking. Financially, there are no limits; however, since it offers a worse approach than our first two alternatives, our team will label this database as *not feasible.*

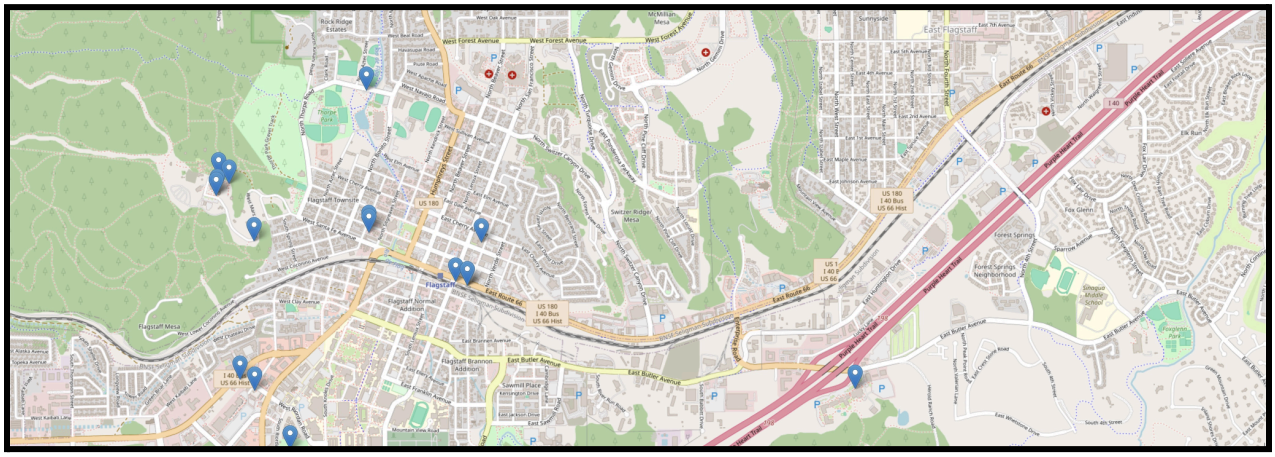

Fig. 5: A preliminary test of Geoapify

Chosen Approach:

Given what our team learned in our ***Analysis***, we think that ***OverpassTurbo*** and ***Wikidata*** both offer great options, so we will use a combination of both moving forward, favoring ***Wikidata*** as it has valuable points in short numbers, and using ***OverpassTurbo***'s points as backup for more rural areas where ***Wikidata*** may come up short.

Feasibility:

Moving forward from what our team decided in the ***Chosen Approach*** section**,** we next went to prove that using these two databases is feasible for Unity. We built a test project and used Unity's HTTPS query system to see if we could bring the JSON data into Unity, and we were successful in both! Seen below are both ***OverpassTurbo*** and ***Wikidata*** accessible in Unity.

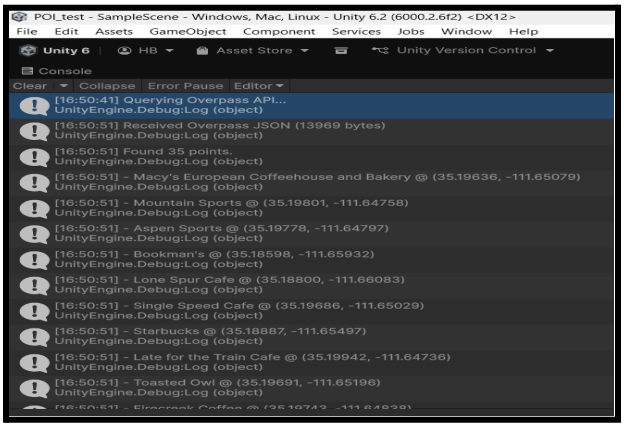

Fig. 5: OverpassTurbo Unity Test

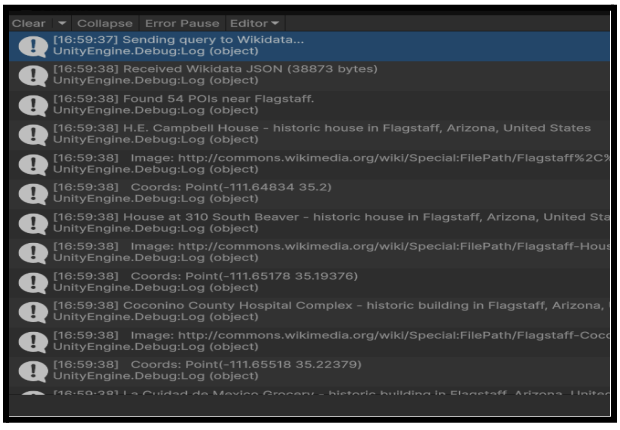

Fig. 6: Wikidata Unity Test

4. Technology Integration

Overall, through our research, we seek to create an architecture that successfully integrates all major components into a unified, production-ready system through the following plan. The final implementation features a Unity-based mobile client built with C#, providing an interactive user interface and real-time GPS tracking through Unity's LocationService. This client communicates with a Node.js and Express backend, which manages authentication, mission logic, and external API requests. The backend connects to a PostgreSQL database hosted on a secure Hostwinds VPS, enabling persistent mission data storage, user progress tracking, and future scalability.

The Geoapify API powers the live map and route visualization, providing dynamic updates as users move through mission areas, and POIs can be retrieved via Wikidata or OverpassTurbo, whichever works best for the player's location. Together, these technologies create a robust and modular system that delivers real-time navigation, smooth gameplay performance, and reliable data synchronization between devices.

This integration reflects a complete alignment between the technical design proposed in the feasibility phase and the final implementation outcomes, confirming both the project's feasibility and successful execution.

5. Conclusion

This document shows that our team's plan for *Missions and Madness* is realistic and doable within the time, tools, and budget we have. By using Unity with Node.js, Express, PostgreSQL, and Geoapify, we can make the app handle GPS tracking, map updates, and player progress across different devices without major issues.

Hosting the database on Hostwinds gives us full control and keeps things cheap enough for a student project while still letting us learn how to manage backups, security, and scaling. It's a setup that fits our current needs but can also grow later if the app expands.

From here, our main focus will be testing Unity builds with working GPS maps, and POI Databases improving the UI and user flow with help from our client, and making sure the database and APIs run smoothly under stress. After that, we plan to release small beta versions for feedback and prepare the app for store deployment when it's ready.

Overall, our approach keeps things realistic and hands-on while still pushing the project toward a complete, working version of *Missions and Madness* that players can actually enjoy using, and Morgan can enjoy publishing.