ArtKnights

Tech Feasibility

24 October 2025

<u>ArtKnights</u>

Project Sponsor: Andres Sepulveda Morales

Team Faculty Mentor: Bailey Hall

Team Members:
Chandler Silk *(Team Lead)*
Kalyana B R Pabbisetty
Anthony Birk
Ashrith Paranam

Table of Contents

Introduction

Context: Millions of artists have had their personal artwork stolen and used to train AI models without credit or compensation. As artificial intelligence continues to evolve at an unprecedented pace, it becomes increasingly difficult for artists to protect their creative work. Many artists are not equipped with the technical knowledge or tools needed to guard against automated scraping systems that harvest art from the internet. The growing prevalence of this issue threatens not only the livelihood of individual artists but also the integrity of creative industries as a whole. If left unaddressed, it could lead to a future where artists have little control over how their work is used or monetized.

Problem: To push back against this trend, some artists have turned to tools like Nightshade, which attempt to confuse AI training systems. These systems rely on pattern recognition and tagging to learn from data. Nightshade poisons that process by making AI models associate incorrect subjects with specific tags, which in turn leads to distorted outputs. While helpful, this solution is limited. It cannot fully prevent targeted scraping or stop curated datasets from cherry-picking unprotected content. Its primary value lies in disrupting large-scale scraping, not in offering individualized protection.

Our sponsor, Andres Sepulveda Morales, is deeply invested in defending artists' rights. As the founder of Red Mage Creative Technologies and Fort Collins AI for Everyone, he has been working to educate artists about the risks posed by AI tools and the available defenses. However, his outreach efforts are constrained by scale. As a one-person advocate, his ability to reach artists outside his local community is limited. Without a scalable solution, his mission to protect artists from AI exploitation remains difficult to achieve.

Goal: To address this need, we are building ArtGuard, a web-based platform that informs and empowers artists to defend their work from unauthorized use in AI training. ArtGuard will serve as both an educational hub and a reporting tool. Its primary purpose is to expose which websites or services may be harvesting artwork without consent and to provide resources that help artists take action.

The first core feature of ArtGuard will be an educational page filled with curated articles, research, and guides that explain how AI scraping works and what artists can do to safeguard their creations. This will serve as a central resource for learning and awareness.

The second feature is a user-driven reporting system. Artists will be able to submit reports identifying platforms they believe have misused their artwork. They can also upload visual comparisons between their original pieces and the suspected AI outputs. These reports will populate a dynamic graph that tracks submission frequency by platform, along with the number of cases that include visual evidence. A public comment section beneath each entry will allow artists to share experiences and insights.

We also plan to release a browser extension that displays trust scores and user reviews for websites as they are visited. This feature will let artists assess platform safety in real time, without needing to consult the ArtGuard website directly.

Impact: ArtGuard is designed to support and amplify the work of artists like Andres Sepulveda Morales by scaling education, awareness, and community reporting efforts. It turns a one-person mission into a collaborative platform that helps artists make informed decisions, recognize risks, and take preventive action. By providing accessible tools and reliable information, ArtGuard will reduce the power imbalance between artists and AI developers, fostering a safer and more transparent creative ecosystem.

Now that we have established the context, problem, goals, and expected impact of the ArtGuard project, it is important to evaluate the technological feasibility of turning this concept into a fully functional solution. At this stage in the project, our focus is on identifying the key technical challenges, exploring alternative approaches, and determining the most suitable technologies for implementation.

In this Technological Feasibility Analysis document, we begin by outlining the major technological challenges that will shape our design and development process. Following this, we analyze each challenge in detail by examining possible technologies, comparing alternatives, and explaining the rationale behind our chosen solutions. The analysis is structured into major subsections such as Programming Setup, Web Hosting, User Report Submission, Discussion Forum, Database and Data Management, and Data Visualization. We then move to the Technology Integration section which illustrates how the chosen frameworks, hosting platforms, databases, and visualization tools work together to form a unified system architecture. Finally, the Conclusion summarizes the overall feasibility findings, highlighting how our chosen stack and design decisions collectively support ArtGuard's mission to empower artists and ensure long-term system reliability. Together, these sections provide a structured roadmap for transforming ArtGuard's vision into a robust, scalable, and secure system.

Technological Challenges

The ArtGuard project faces several major technological challenges that will influence its development and long-term sustainability. Each of these areas requires thoughtful consideration to ensure the system is efficient, secure, and scalable.

- **Web Framework Selection**

  One of the first and most important challenges is selecting an appropriate web framework that can handle routing, rendering, and server-side logic effectively. Choosing between frameworks such as Next.js or Express.js involves balancing performance, scalability, and integration with databases and APIs. The decision also requires evaluating static versus dynamic rendering approaches to manage server load efficiently while providing responsive user interactions.

- **Website and Database Hosting**

  Another key challenge is determining a reliable and cost-effective hosting solution for both the website and database. Hosting platforms such as AWS, Google Cloud, or Microsoft Azure each offer distinct advantages, but the choice must support both the web application and PostgreSQL database seamlessly. Factors such as uptime, scalability, deployment simplicity, and long-term cost must be weighed carefully to ensure consistent platform accessibility for users.

- **User Forum / Discussion System**

  ArtGuard's success depends heavily on fostering community engagement among artists. Implementing a user discussion forum is therefore a significant technical undertaking. We must decide whether to integrate an existing platform like Discourse or build a custom forum from scratch. The forum needs to include moderation tools, spam prevention mechanisms, and authentication integration, which introduce additional layers of complexity in both setup and maintenance.

- **Data Visualization (Graphs of Reports)**

  A major challenge lies in visualizing the data collected from user reports in a clear and insightful manner. We need to implement dashboards and graphs that display report trends and statistics using tools such as Chart.js or Recharts. The challenge here is ensuring the data visualizations update in real time while maintaining fast performance and a visually accessible layout for all users.

- **Database and Data Management**

  Managing and structuring the large amount of user-generated data is another critical aspect. We plan to use a relational database such as PostgreSQL to store user accounts, reports, and forum data. The challenge lies in maintaining consistency, data security, and query efficiency while handling potentially large datasets. Proper schema design and optimized queries are essential to achieving this balance.

- **User Report Submission System**

  Finally, creating a secure and user-friendly submission system for reporting AI artwork misuse presents its own challenges. The form must handle file uploads, data validation, and error checking efficiently, while also preventing spam and duplicate entries. Since these reports are central to ArtGuard's mission, the system must be both reliable and scalable, capable of managing growth in user participation over time.

Having outlined the major technological challenges, we now analyze each in detail to identify the most feasible solutions.

Technology Analysis

Programming Setup

Introduction: The core of this project is a website which allows us to aid in the education and interaction of artists who are seeking to better understand the effects AI can and will have on them. The site can be broken into three categories: articles, reports, and forums. The articles section will have articles written about the pressing topics facing artists with respect to AI, and help them understand what's going on in a way that they can understand. It will also help with keeping them updated, since the world of generative AI changes quickly. The second portion is a report section where users can report and score various websites based on how well the website protects their art. This allows users to quickly identify which websites protect their art, and which ones do little about it. This also allows for changes when updates to these websites occur. Lastly, there will be a forum section for users to openly discuss different topics.

Desired Characteristics: We need a backend setup to support all 3 use cases. We are looking for an option that is low-cost, responsive, and straightforward to develop on. In addition to these core characteristics, we also need to consider security, feature fit, scalability, and maturity. Since our site has several different parts, whatever we choose will need to handle all 3. This will likely lead to us needing to find the best fit solution for the entire project, even if another setup option looks more appealing in two out of the three areas.

Alternatives: First up for analysis is the programming setup. This lays the basis for how we will build the entire site. Through research of community discussion and industry articles, we found the following alternatives:

- **Option 1 - HTML/CSS/JS:** The traditional "triple stack" of web development languages, this setup is easy to use. It is also very familiar to us, having been used in several of our courses here at NAU. They have a long history of development and support.
- **Option 2 - React/Next.js with Typescript + Sanity + Discourse:** A more modern setup to the classic three, this setup gives us a rich set of features to work with. Next.js gives us rendering tools. Sanity gives us a suite of tools for articles and content management. Discourse handles the forums independently.
- **Option 3 - Astro + Discourse:** Astro uses static files deployed by the CDN, which makes the articles section lightning fast. Discourse manages the forum side independently.

Analysis: Each of these solutions comes with their own set of positives and negatives. Finding a solution that works for all 3 of our use cases makes it more difficult to decide on a definitive option. To find the best fit, we can evaluate them based on our characteristics. The information and ratings provided were found through documentation, user discussions, and video reviews.

Cost: HTML/CSS/JS has no licenses and is cheap to host. Next.js and Discourse are open source, and Sanity has a free tier, but may require a small plan in the future. Static hosting for the SSG is cheap, but the API service for the ratings may cost money.

Performance: For performance, Astro is the fastest due to its static nature. Next.js and Sanity are also very quick, only refreshing what they need to. HTML/CSS/JS can be fast, but we would have to develop our own caching and pipelines.

Ease of Use: HTML/CSS/JS is the setup that the team is most experienced in, however it would require us to create a lot of functions ourselves. Comparatively, Next.js has file-based routing and built-in patterns for things like caching. Articles are easy to handle on Astro, but we would have to integrate the API ourselves.

Maturity: Next.js has a large community and ecosystem, and has many resources to seek out for feature implementation. Astro has content collections to pull from when needed. HTML/CSS/JS, comparatively have a mature community, but has a lot less resources for "drop-in" features.

Security: On the forum side, Discourse provides anti-spam systems. Next.js allows for secure logins and connection to Discourse through that. Since Astro is a static system, it is secure by nature.

Feature Fit (Articles/Ratings/Forums): For Next.js and Astro, Discourse provides excellent support for the forums. With Next.js, Sanity provides a UI for creation and editing, and Next.js provides good API tooling and database integrations. Astro provides strong functionality for static articles, Comparatively, HTML/CSS/JS has no built-in functionality. This makes it fine for static articles, but makes it difficult to edit, and the ratings system must be built from scratch.

Scalability: HTML/CSS/JS require us to tweak the custom backend in order to handle larger usage. Next.js works well with serverless/containers. Astro is easily scalable for articles, but would require work on the ratings system. Discourse handles scale adjustment nicely.

Criteria	HTML/CSS/JS	Next.Js + Sanity + Discourse	Astro + Discourse
Cost	✅Good	✅Good	✅Good
Performance	✅Good	✅Excellent (Image Optimization, Edge Caching)	✅Excellent (Static articles + CDN)
Ease of Use	⚠️Moderate (Requires lots of custom work)	✅Excellent (File-based routing)	✅Good (Both ratings and forums need to be separate)
Maturity	⚠️Moderate (Less drop-in guidance)	✅Excellent	✅Good
Security	⚠️Moderate (Relies on host)	✅Excellent	✅Excellent (Static site)
Feature Fit (Articles, Ratings, Forums)	⚠️Moderate (Needs CMS for workflow) ⚠️Moderate ❌Poor (Hand coding unreasonable, or need external service)	✅Excellent (CMS+ISR) ✅Excellent (API routes, database integration) ✅Excellent	✅Excellent ✅Good (Requires bespoke API) ✅Excellent
Scalability	✅Good	✅Excellent (Works with serverless, containers)	✅Good

Chosen Approach: Although we are more experienced with HTML/CSS/JS, it seems that to best cover our goals for this project, we would need to go with something more modern with greater functionality. As shown in the table, the two modern options are fairly close, but one has the edge for our use cases. For this, we will choose **Next.js + CMS + Discourse**. This allows us to create a responsive web app that can combine our three main use cases. Next.js provides hybrid rendering for both SSG/ISR and SSR, the CMS will handle the articles and workflow, and Discourse can handle all of the forum needs.

Feasibility: We've created a simple site that is able to host articles and display ratings, and has a link to the forum page. We'll need to create the API in order to have ratings to display, as well as fleshing out our forums using Discourse.

Web Hosting

Introduction: We've laid out how we plan on structuring the web app. However, this web app is not for our use only. We need to be able to allow other users to access our web app for it to properly function. For this, we need to find somewhere to host the website. We need a host that supports our chosen technologies, provides a secure connection to protect the users, and gives us the ability to reach the people who would benefit most from our web app.

Desired Characteristics: Our criteria for our hosting options is similar to the criteria for our programming setup. The options need to be low-cost, high performance, and straightforward to host our site on. In addition, having robust support to resolve issues and a strong security setup are very important. Lastly, it should easily support our choice of Next.js + Sanity + Discourse for the programming setup.

Alternatives: Our hosting alternatives are well-established. We found them through looking through discussion forums and web articles, and are as follows:

- **Option 1 - AWS:** AWS is one of the leading hosting platforms on the web. It gives us the tools to utilize the programming we discussed earlier, as well as being adaptable to any changes we should make and any scaling up we would have to do.
- **Option 2 - Google Cloud:** Google CLoud uses containers with automatic scaling and can be paired with other services. It has a strong development platform as well.
- **Option 3 - Microsoft Azure:** Azure is a great fit for web apps and APIs, and supports a large amount of functionality.

Analysis: All three of our options provide a rich suite of features, with each being backed by one of the major tech companies in today's world. Therefore, the scores can be much closer. We analyzed each alternative based on user experiences and community discussion:

Cost: AWS only makes us pay for what we use, and Google Cloud only charges when our containers handle requests. Microsoft Azure is cheap too, but can incur costs when idle.

Performance: Google Cloud uses autoscaling containers that react to traffic. AWS uses atomic deploys that clears the cache so users can see the newest pages quickly. Microsoft Azure has solid deployment, but isn't as responsive as the other options.

Ease of Use: Microsoft Azure is good for .NET/Node/containers, but requires more setup for API projects. Google Cloud has revisions and rollbacks built in. AWS can connect to a GitHub repository, and supports both SSR frameworks and static sites, as well as built in support for Next.js.

Maturity: Documentation and community support for all three options are extensive.

Security: All three options use the security protocols of major established tech companies.

Feature Fit: AWS contains support for Next.js, API for the ratings, and Discourse. Google Cloud can host Next.js and Discourse on containers, and provides API support. Microsoft Azure provides support for the programming setup, but not as robust as the other options.

Criteria	AWS	Google Cloud	Microsoft Azure
Cost	✅Excellent	✅Excellent	✅Good (can be pricier based on tier)
Performance	✅Excellent (quick cache clears for faster page loading)	✅Excellent (Good edge speed)	✅Good (Solid PaaS with CDN)
Ease of Use	✅Excellent (instant cache validations)	✅Excellent (revisions, rollbacks)	✅Good (more setup required)
Maturity	✅Excellent	✅Excellent	✅Excellent
Security	✅Excellent	✅Excellent (Private networking)	✅Excellent
Feature Fit (Articles, Ratings, Forums)	✅Excellent (Integration for Next.js, Sanity, and discours)	✅Excellent (Containerized Next.js and Discourse)	✅Good (less integration)

Chosen Approach: As shown above, all three options are very relevant to our use cases. However, Microsoft Azure falls a little short of the others on that criteria. From there, both **AWS** and Google Cloud are relatively equal. With this, we will choose **AWS**, since it offers integrated support for some parts of our coding setup.

Feasibility: Once we have developed the site prototype, we will go to **AWS** to host it. This will not be immediately necessary, but we have managed **AWS** hosting before for another course, and setting it up is straightforward using the tools Amazon provides.

User Report Submission

Introduction: A core functionality of ArtGuard is the User Report Submission system, which allows artists to report cases where their artwork may have been misused or used without consent in AI training datasets. This feature forms the foundation of our data collection process and directly influences other components, such as analytics and community discussions. It will enable users to submit detailed reports that include the suspected website or service, a brief description of the issue, and optional visual evidence for verification. By making this process simple and efficient, we aim to empower artists to take an active role in safeguarding their work while contributing valuable information to the platform's shared database.

The report submission system will include form(s), and file uploads. These reports will be stored in a secure database. Moreover, this data will feed directly into the Data Visualization component of ArtGuard, where aggregated information will be used to generate graphs and insights about report trends, which helps artists to identify the most-affected websites. This integration ensures that every user submission not only helps identify art misuse but also contributes to a broader understanding of AI-related risks to artists.

Desired Characteristics: The report submission system should be user-friendly, ensuring that artists can quickly and accurately report issues easily. Security is a top priority, all submitted data, especially user-uploaded images, must be encrypted and securely stored. We should also consider scalability, allowing the system to handle a growing number of submissions over time. Eventually, we plan to provide data validation to prevent spam or duplicate reports, and it should remain cost-effective.

Alternatives: We explored four possible and widely used approaches for implementing the User Report Submission system.

- **Option 1 - Node.js with Express:** This is a lightweight, and flexible framework that integrates smoothly with Next.js and PostgreSQL. It supports efficient form handling, routing, and file uploads using middleware like Multer. Its strong community support, scalability, and open-source nature make it the most practical choice for our system.
- **Option 2 - Flask (Python):** This is a simple Python framework suitable for small-scale API development. While it performs well for lightweight applications, integrating Flask with a JavaScript-based frontend like Next.js would require additional setup and limit asynchronous compatibility.
- **Option 3 - PHP:** This is a traditional option widely used for form submissions and server-side scripting. However, it is less effective for modern API-based systems, and its integration with Next.js would be inefficient and complex.
- **Option 4 - Custom API within Next.js:** Next.js provides built-in API routes for handling submissions directly within the app. This simplifies deployment but restricts

advanced backend features such as file uploads and middleware handling, making it less ideal for our long-term scalability needs.

Analysis: We evaluated four potential approaches and tools for building the User Report Submission system based on the following criteria.

Integration: Integration evaluates how easily each framework connects with ArtGuard's existing frontend, database, and overall architecture. Node.js with Express integrates efficiently with the ArtGuard architecture because both frontend and backend are JavaScript-based, allowing seamless data transfer through RESTful APIs. Flask provides straightforward integration but requires additional middleware to coordinate with React-based interfaces. PHP's integration is less direct, as it operates in a different environment and may need adapters for asynchronous operations. A Custom API with Next.js integrates natively within the existing stack, enabling smooth communication between client and server components and reducing dependency complexity.

Form Handling & Validation: This criterion is used to assess the framework's ability to process, validate, and manage user input securely and efficiently. Node.js with Express offers robust libraries such as express-validator for real-time input validation and error handling, ensuring secure and reliable data collection. Flask provides built-in form support through WTForms, which is functional but less dynamic for large-scale web apps. PHP can perform validation but often relies on manual scripting, increasing potential for errors. A Custom API with Next.js supports modern form libraries like react-hook-form and integrates client-side and server-side validation, improving user experience and minimizing invalid submissions.

File Upload & Storage: This examines how effectively each option supports uploading, storing, and managing report files within the system. Node.js provides efficient file management using tools like Multer, allowing quick file upload and integration with cloud storage like AWS. Flask supports file uploads through extensions like Flask-Uploads, which work well for smaller systems but need more setup for scaling. PHP supports file handling natively but it lacks performance optimization for concurrent uploads. A Custom API with Next.js allows flexible integration with cloud services via APIs, providing secure and scalable storage management consistent with the ArtGuard backend.

Security: This is used to assess the strength of built-in protections against threats such as data breaches, injection attacks, and unauthorized access .Node.js ensures strong protection through middleware such as Helmet.js and secure token-based authentication, supported by extensive security documentation and community updates. Flask provides good protection with libraries like Flask-Security, though additional configuration is needed for large-scale deployments. PHP includes standard security measures, but its legacy structure makes it more prone to vulnerabilities if not configured carefully. A Custom API built within Next.js supports secure API routes and server-side session handling, making it inherently safer and aligned with the overall system's authentication model.

Performance & Scalability: This determines how well each framework handles concurrent requests and adapts to increasing system load. Node.js's non-blocking I/O model supports concurrent requests efficiently, proven in production systems such as Netflix and LinkedIn. Flask delivers stable performance for moderate traffic but requires additional tools like Gunicorn or Nginx for scaling. PHP is dependable but synchronous, limiting scalability. A Custom API with Next.js provides high performance due to server-side rendering and edge deployment options, ensuring fast response times and easy scalability as user load increases.

Cost: Cost considers setup, and maintenance expenses. Node.js, Flask, and PHP are all open-source frameworks with extensive community support, keeping development costs minimal. Node.js and Flask also have vast libraries that reduce time spent on custom implementations. A Custom API with Next.js introduces slightly higher costs because of the development effort required to build from scratch.

The following table shows the feature evaluation and overall comparison of the options considered for the User Report Submission feature based on the selected criteria.

Criteria	Node.js + Express	Flask (Python)	PHP	Custom API with Next.js
Integration	✅ Excellent	⚠️ Moderate	⚠️ Moderate	✅ Excellent
Form Handling & Validation	✅ Excellent	✅ Good	⚠️ Moderate	✅ Excellent
File Upload & Storage	✅ Excellent	✅ Good	⚠️ Basic	✅ Excellent
Security	✅ Excellent	✅ Good	⚠️ Basic	✅ Excellent
Performance & Scalability	✅ Excellent	⚠️ Moderate	⚠️ Moderate	✅ Excellent
Cost	✅ Free (Open Source)	✅ Free (Open Source)	✅ Free	⚠️ Moderate

Chosen Approach: Based on our analysis, we decided to use **Node.js** with **Express** for implementing the User Report Submission system. This approach offers the best integration with our existing stack, particularly with the **Next.js** and PostgreSQL. Since it is free and open source, it keeps the costs low.

Feasibility: Some of our team members have worked with **Node.js + Express** on several small projects before, which has provided us with valuable experience and confidence in moving forward with this approach.

User Discussion Forum

Introduction: One of the most defining features of ArtGuard is to foster collaboration among artists through a dedicated user discussion forum. This forum aims to create an interactive community where artists can share experiences, and exchange knowledge. It will serve as a space for users to comment on reported cases, discuss preventative techniques, stay informed about updates, and many more.

In this forum, we want to include features like user threaded conversations, "like" buttons, replies, and post tagging to improve user interaction. Because user engagement and moderation are critical to the success of this feature, the forum must balance usability, and strong administrative control.

Desired Characteristics: The discussion forum should integrate seamlessly with our existing stack, **Next.js** for the main web app, a headless CMS for article publishing, and PostgreSQL. Moderation tools are also under consideration which is needed to manage discussions and filter spam. Overall, the forum should be secure, offering data protection, and cost-efficient, ideally open-source and easy to maintain as the community expands.

Alternatives: The following are the options we explored four potential solutions for implementing the User Discussion Forum.

- **Option 1 - Discourse:** A popular open-source discussion platform. It supports SSO integration, threaded conversations, real-time notifications, and moderation tools. Discourse is highly scalable, customizable, and has a strong community, making it a robust choice for long-term use.
- **Option 2 - NodeBB:** We considered Node.js-based forum framework as it is lightweight and integrates well with modern web stacks, but customization and setup require additional configuration compared to Discourse.
- **Option 3 - phpBB:** As PHP-based forum platform known for reliability and moderation features. However, it lacks a modern interface and API support, making integration with frameworks like Next.js more challenging and less efficient.
- **Option 4 - Custom Forum:** Building a fully custom forum is also an option. However, it would require significant development time, introduce security and maintenance challenges, and delay other core project features.

Analysis: We evaluated the four considered tools based on the following chosen criteria to identify the most suitable option for ArtGuard's discussion forum.

Integration: Integration evaluates how smoothly each forum option connects with ArtGuard's existing technologies such as Next.js, PostgreSQL, and the headless CMS. Discourse supports Single Sign-On (SSO) and API connections with external web apps, including Next.js. Numerous tutorials on Meta Discourse confirm seamless authentication sharing between Discourse and React-based frameworks. NodeBB also works well with Next.js but needs manual authentication setup. phpBB uses older PHP methods that complicate integration, while a custom forum would require developing all endpoints and authentication from scratch.

Moderation Tools: Moderation capabilities are crucial for managing user content and preventing spam. Discourse again has a built-in trust system, automatic spam detection, and moderation dashboard, all of which are supported by its extensive documentation. NodeBB includes basic moderation tools but depends heavily on community plugins for advanced controls. phpBB has a mature and time-tested moderation system, but it lacks automation. A custom forum would need these tools built manually, increasing workload and maintenance.

Performance & Scalability: This criterion refers to how efficiently the website handles user traffic and data growth. Discourse uses Redis caching and supports thousands of concurrent users. This capability is documented on the official Discourse Meta site. NodeBB performs well but may require additional Redis clustering for heavy workloads. phpBB provides consistent performance for smaller communities but lacks the distributed processing capacity needed for large scale use. A custom forum's scalability would depend entirely on design optimization and available infrastructure, requiring extensive testing and tuning.

Feature Implementation: This criterion examines the range and completeness of the features provided by each platform. Discourse includes all major features like threaded replies, notifications, tagging, and search by default that are the core features for our ArtGuard's web app. NodeBB offers similar features but adds maintenance challenges. phpBB provides basic forum functionality but lacks real-time interactions. The custom forum was rated because implementing these features would require substantial development time.

Customization Options: This determines how easily we can adapt the forum's design and workflows to match ArtGuard's branding. Discourse supports theme customization, CSS overrides, and plugin APIs that allow full design control. NodeBB being highly customizable but requiring deeper JavaScript expertise for advanced changes. phpBB has rigid PHP templates, which make visual or structural modifications difficult. A custom forum is flexible but would require high expertise, entail a heavy workload and higher maintenance effort.

Security: Security measures protection against vulnerabilities. Discourse with frequent updates, OAuth2 authentication, and robust permissions model. It is trusted by high-security communities, reinforcing its reliability. NodeBB because of its active development and secure core, though plugin dependencies may create exposure. phpBB being stable but historically prone to vulnerabilities common to PHP. The custom forum security would depend entirely on our team's coding and monitoring diligence.

Cost: Cost reflects price, hosting, and maintenance effort. Discourse, NodeBB, and phpBB are all open-source and free to use, with expenses limited to server hosting. Discourse and NodeBB were rated excellent for affordability, while phpBB is also good but requires more setup time. The custom forum would involve extensive development and long-term support.

The following table shows the feature evaluation and overall comparison of the options considered for the User Discussion Forum feature based on the selected criteria.

Criteria	Discourse	NodeBB	phpBB	Custom Forum
Integration	✅ Excellent	✅ Good	⚠️ Limited	⚠️ Complex
Moderation Tools	✅ Excellent	✅ Good	✅ Excellent	⚠️ Complex
Performance & Scalability	✅ Excellent	✅ Good	⚠️ Moderate	⚠️ Complex
Feature Implementation	✅ Excellent	✅ Good	⚠️ Moderate	❌ Difficult
Customization Options	✅ Excellent	✅ Good	⚠️ Limited	✅ Excellent
Security	✅ Excellent	✅ Good	⚠️ Basic	⚠️ Complex
Cost	✅ Free (Open Source)	✅ Free (Open Source)	✅ Free	❌ High

Chosen Approach: Based on the overall analysis, we decided to go with **Discourse** for our discussion forum, as it meets all our needs like seamless integration, strong moderation capabilities, high scalability and others. Most importantly, it's completely free and open source, despite offering an impressive range of advanced features. Moreover, **Discourse** also has an active community and supports integration with Next.js, which aligns perfectly with our architecture.

Feasibility: To confirm the feasibility of using **Discourse**, I explored multiple demonstrations and community setup guides. I watched a detailed YouTube tutorial on integrating **Discourse** with a Next.js web app, which illustrated how session tokens can be shared securely between both systems. It felt so achievable within our team's skillset. Importantly, as mentioned before, we also developed a sample website using a blog-style forum as a practice implementation to test our approach.

Database and Data Management

Introduction: One of the most critical technological components of this project is the database system that will store and manage user-submitted reports. Since the website functions as a platform for artists to report and track unauthorized use of their artwork on AI training websites, it must handle large volumes of data efficiently and securely. Every report submitted by a user will contain important information such as the artist's username, URLs of suspected misuse, timestamps, and user credentials. This data needs to be stored in a structured, reliable manner that supports easy retrieval, search, and cross-referencing. Furthermore, because the integrity of user-submitted information directly impacts the credibility of the entire site, the chosen database solution must ensure data accuracy, prevent duplication, and maintain a high level of consistency between related data entities such as users, reports, and source websites. Therefore, selecting the right database technology is a foundational design decision that will significantly influence the system's overall scalability, performance, and long-term maintainability.

Desired Characteristics: Our desired characteristics include: Data integrity and relational consistency, Scalability, Security, Ease of integration, Query flexibility, and Low Cost. For Data integrity and relational consistency we need to make sure that the technology ensures reports, users and the website are linked correctly. For Scalability it should be able to handle growth in both users and reports as the site gains more users. For Security it should have the ability to protect or encrypt user data. It should also be easy to use and work with Node.js, Express, and EJS as we have previously decided to use those programs. It should also support complex queries. Finally it should also be low cost or ideally open source.

Alternatives: First up for which database we want to use, we discussed and researched forums which database systems are more commonly used and have features we plan to use, we found the following:

- **Option 1 - PostgreSQL:** For our project we could use PostgreSQL which has an open-source relational database; strong support for complex queries and data integrity. It is widely used in production systems. Supports JSON, indexing, and extensions ideal for unstructured data. Integrates well with Node.js via libraries like pg.

- **Option 2 - MySQL:** We could also use MySQL which is also open-source and relational.It has good performance, large user base, and strong hosting support. However it is slightly less feature-rich for complex queries and JSON handling compared to PostgreSQL.
- **Option 3 - MongoDB:** Another option we have is MongoDB which has a NoSQL document database, great for flexible, schema-less data. Easier to scale horizontally, but weaker for relational consistency and is common in Node.js apps, and integrates with Mongoose ORM.

Analysis: We researched documentation and user threads discussing each program and their strengths and weaknesses.

Data Integrity and Relationships: We found that PostgreSQL is excellent at handling data integrity and complex relationships as it fully supports foreign keys, constraints, and multi-table joins. MySQL supports the same features however they are not as strict or advanced as PostgreSQL's which is why it's rated good. MongoDB is a NoSQL database so it does not support joins and instead relies on document embedding or manual referencing which limits its ability to handle traditional data.

Scalability: PostgreSQL scales well vertically and supports horizontal scaling which is the method we will primarily use making it strong. MySQL performs reliably with vertical scaling; however its horizontal scaling options are extremely limited and require additional programs. MongoDB is designed with horizontal scalability in mind which makes it a good option for our project.

Security: PostgreSQL includes advanced security features such as role-based access control, SSL encryption and support for further extensions to heighten security. MySQL offers a good level of security with solid access controls and encryption, although it requires more setup and implementation to match PostgreSQL. MongoDB also provides good security features but the advanced protections are dependent on the edition used.

Integration with Node.js: PostgreSQL integrates extremely well with Node.js as it has the popular and well documented pg library allowing seamless integration. MySQL has Node.js support as well but it is more commonly used with PHP which would be an additional program to use. MongoDB integrates very well with Node.js with native support using Mongoose that helps simplify data modeling.

Query Flexibility: PostgreSQL offers excellent query flexibility as it supports standard SQL and JSON functions giving us many options to query data. MySQL has decent SQL capabilities but lacks a lot of the JSON and advanced query features found in PostgreSQL. MongoDB uses an aggregation pipeline for queries, which overall is good and flexible within its environment.

Cost: PostgreSQL and MySQL are both completely free and open source which is ideal as we want to keep costs to a minimum. MongoDB offers a free tier however to access its full features we will need to use a monthly subscription.

Criteria	PostgreSQL	MySQL	MongoDB
Data integrity & relationships	✅Excellent	✅ Good	⚠️ Weak (no joins)
Scalability	✅ Strong (vertical, some horizontal)	⚠️ Moderate	✅ Good
Security	✅ Advanced (roles, SSL, extensions)	✅ Good	✅ Good
Integration with Node.js	✅ Native support (pg library)	⚠️ Native support with addition of PHP	✅ Native support (Mongoose)
Query flexibility	✅ Excellent (SQL + JSON)	⚠️ Moderate	✅ Good (aggregate pipeline)
Cost	✅ Free and open source	✅ Free and open source	❌ free tier but pay for full

Chosen Approach: Our chosen approach is **PostgreSQL** because it ranks highest for the characteristics relevant in our project. It fits because it has strong relational features, secure and well supported by Node.js, handles structured data with flexibility and is open source. A key consideration for this project is protecting user information, especially account passwords and identifying data. **PostgreSQL** supports strong integration with Node.js authentication libraries and hashing tools such as bcrypt, enabling secure storage of password hashes rather than plain text values. This helps create a safer and more trustworthy website for users. This makes it the best choice for our project.

Feasibility: To prove feasibility we created a sample website using a blog forum as we plan to do as a feature for our project and we were able to get it to work, as it could create, read, update, and delete. The next steps would be testing indexing and performance for search through larger datasets, implementing a simple authentication and data validation and simulating a large load and seeing how it performs.

Data Visualization

Introduction: To effectively present and analyze the information collected from user reports, our website needs a strong data visualization component. Raw data is often complex and difficult to interpret, so transforming it into visual representations such as graphs, charts, and dashboards helps users and administrators quickly identify trends, patterns, and statistics. The data visualization system will play a crucial role in improving decision-making, transparency, and the overall usability of the platform by presenting meaningful insights in an intuitive, interactive format.

Desired Characteristics: We need a data visualization approach that is responsive, interactive, and dynamic. It should automatically update as new user reports are submitted, ensuring real-time accuracy. Additionally, it must integrate seamlessly with our backend and database while maintaining high performance and minimal latency. The visualizations should be responsive, and adjust smoothly across devices and screen sizes. They should be interactive and allow users to hover, filter, or click for deeper insights. It should update dynamically in real-time when new data is submitted, and be designed with accessibility in mind for all users. Lastly, it should be aesthetically clean and easy to integrate, being compatible with modern frameworks like Next.js and our backend stack.

Alternatives: Below are three potential visualization approaches and libraries that can be integrated into our project:

- **Option 1 – Chart.js:**
  A lightweight and beginner-friendly library for 2D charts. Ideal for dashboards showing report counts, user activity, and time-based trends. I chose Chart.js because I've used it before in one of my projects, and it's very easy to set up for clean, responsive dashboards.
- **Option 2 – D3.js:**
  A powerful, low-level visualization library offering complete control over interactivity, animations, and custom chart design. Great for complex, data-driven visual storytelling. We picked D3.js because I've seen tutorials and examples on YouTube showing how powerful it is for creating interactive and customized visualizations.
- **Option 3 – Google Charts:**
  Cloud-based visualization API offering built-in charts, good performance, and easy embedding with auto-scaling features.I chose Google Charts because I've seen it used in several web projects online, and it's simple to embed with great performance for quick data visualization.

Analysis: Each visualization tool offers distinct advantages. The right choice depends on our desired level of customization, performance, and integration ease with our frontend framework.

Cost: All three libraries Chart.js, D3.js, and Google Charts are excellent choices in terms of cost. Chart.js and D3.js are completely free and open-source, giving full flexibility without licensing fees. Google Charts is also free for standard use through its API, making all three options cost-effective and ideal for a student or project environment.

Performance: Chart.js is known for its lightweight structure and fast rendering, making it efficient for small to medium datasets. D3.js performs well with large or complex data visualizations but can be resource-heavy due to its detailed DOM operations. Google Charts delivers excellent performance since it is cloud-optimized, efficiently handling data rendering through Google's servers.

Ease of Use: Chart.js offers a simple API and straightforward setup, allowing quick chart creation with minimal code. D3.js provides deeper control but has a steeper learning curve, requiring understanding of JavaScript, SVGs, and data binding. Google Charts is very beginner-friendly with ready-made chart types and can be embedded easily, making it great for fast implementation.

Interactivity: Chart.js includes built-in animations, hover effects, and tooltips, offering good interactivity for most visual needs. D3.js stands out in this area, allowing complete customization and dynamic user interactions. Google Charts provides moderate interactivity, though most features are predefined and less customizable compared to D3.js.

Integration with Next.js: Chart.js integrates very well with Next.js using React wrappers such as *react-chartjs-2*, making it easy to use within React-based applications. D3.js also works with Next.js but requires more manual setup to manage components and rendering. Google Charts integrates smoothly using its embeddable API or React packages, ensuring compatibility with Next.js projects.

Aesthetic Quality: Chart.js produces modern, visually appealing charts right out of the box. D3.js can achieve stunning visuals but requires manual design and configuration. Google Charts offers clean, professional-looking visuals with built-in themes, though customization options are somewhat limited.

Scalability: Chart.js handles moderate datasets effectively, making it suitable for dashboards and general reporting tools. D3.js is built for high scalability, capable of managing large, dynamic, and real-time datasets. Google Charts also scales well thanks to server-side rendering and cloud-based data handling.

Criteria	Chart.js	D3.js	Google Charts
Cost	✅ Excellent (Free, open-source)	✅ Excellent (Free, open-source)	✅ Excellent (Free for standard use)
Performance	✅ Good (Lightweight and fast)	⚠️ Moderate (Can be heavy for complex visuals)	✅ Excellent (Cloud-optimized)
Ease of Use	✅ Excellent (Simple API, easy setup)	⚠️ Moderate (Steeper learning curve)	✅ Excellent (Quick to embed)
Interactivity	✅ Good (Built-in animations and tooltips)	✅ Excellent (Fully customizable interactivity)	⚠️ Moderate (Limited customization)
Integration with Next.js	✅ Excellent (React wrappers available)	✅ Good (Requires manual setup)	✅ Excellent (Embeddable via API)
Aesthetic Quality	✅ Excellent (Modern chart styles)	✅ Good (Requires manual setup)	⚠️ Good (Predefined styling options)
Scalability	✅ Good (Handles moderate datasets)	✅ Excellent (Designed for large, dynamic datasets)	✅ Excellent (Server-side rendering support)

Chosen Approach: While D3.js provides the highest flexibility and depth, it requires more complex implementation and setup. **Chart.js** strikes an optimal balance between simplicity, responsiveness, and integration ease with our frontend framework (Next.js). It is lightweight, developer-friendly, and sufficient for the scope of our project, supporting bar, line, and pie charts effectively. Therefore, **Chart.js** is the chosen visualization library for this project.

Feasibility: Implementing **Chart.js** within our Next.js environment is straightforward and highly feasible. The library can easily pull real-time data from our backend API and render visual updates on dashboards without page reloads. Its React integration libraries (like react-chartjs-2) simplify component embedding. We can quickly set up visualizations for metrics such as: number of user reports submitted per day/week, categories of reported issues, and user engagement levels over time. After evaluating individual solutions, it is essential to understand how these technologies interact as a unified system.

Technology Integration

We have spent most of this document analyzing the technological options available to us. We have explored the pros and cons of each alternative, and have made choices on the preliminary systems we will use. We have chosen the systems carefully so that we can combine them into an overall system architecture.

When a user accesses our web app, they will be greeted by the 3 uses of our site: articles, ratings, and discussion. Data is rendered in the web app by **Next.js**, and then pushed to the **AWS** container for display. User reports are submitted using **Node.js with Express**, and stored in **Postgres.** Then, **Chart.js** takes the data from the database and displays it on the web app. Should the user click on the forums link, they will be sent to the forums, which are handled by **Discourse**. Their login on the web app allows the forums to recognize the user without having to login again. We can also include functionality to notify the web app of a high activity topic on the forums, which can notify users on the web app of a discussion happening about the site they're viewing. Figure 1 below shows how our chosen systems interact with each other as the system architecture:

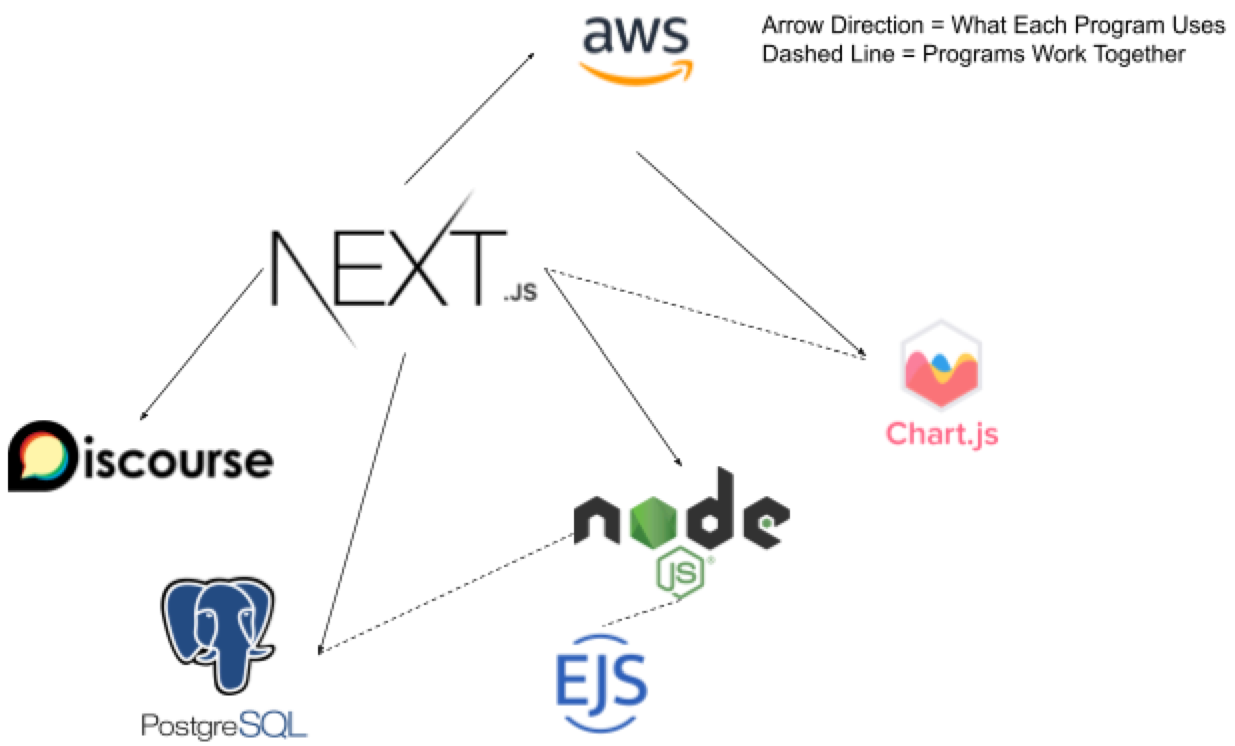
Arrow Direction = What Each Program Uses
Dashed Line = Programs Work Together
aws
NEXT.JS
Chart.js
Discourse
node JS
PostgreSQL
EJS

Figure 1: Diagram of the system architecture

These systems form our system architecture, and provide us with a solid foundation on which we can build our project. The features provided by each will allow us to maximize our goals.

Conclusion

The ArtGuard project demonstrates that it is technologically feasible to create a secure, interactive, and user-friendly platform that empowers artists to protect their work from unauthorized AI use. Through carefully structured, data-driven, and comparative analysis of each major design challenge, we identified reliable solutions that align with our performance, scalability, and usability goals. By selecting PostgreSQL for robust data storage, Next.js + CMS + Discourse and AWS for backend development, and modern front-end tools such as Next.js, Discourse, and Chart.js, we established a well-integrated technology stack capable of supporting all critical functions of the system.

Each core feature, such as the **visual design**, **user discussion forum**, **data visualization dashboard**, and **report submission system**, plays an essential role in ensuring that users can engage meaningfully with the platform while maintaining trust and transparency. Our technology choices were made through a structured evaluation process, ensuring that they are scalable, secure, and cost-effective for long-term deployment.

Moving forward, our next steps will include developing a prototype that integrates all these technologies, performing real-world testing with sample data, and refining usability based on user feedback. The outcome will be a comprehensive and reliable web application that not only educates artists but also builds a community dedicated to protecting creative integrity in the age of AI. With a clear technical foundation and a strong design rationale, ArtGuard is well-positioned for successful implementation and future expansion.