ChatGPT

Recommended Tech Stack for ARK: Survival Ascended Breeding Manager

Frontend (Next.js UI & Visualization)

- **Next.js (React 18)** – Continue using Next.js for the breeding manager's frontend. It provides server-side rendering and a seamless way to integrate an API without leaving the app [1], which suits your needs for dynamic stat displays and interactive tools. Next.js's routing and built-in APIs make it easy to develop a cohesive frontend+backend on the same project [1], and it can be deployed on Vercel for scalability and ease of hosting.
- **UI Component Library (Chakra UI or Material UI)** – Leverage a modern React component library for consistent, responsive UI. *Chakra UI* is a popular choice because it is "simple, modular, and accessible" [2], offering a set of pre-built components (menus, modals, forms, etc.) that can be easily themed for ARK's style. This speeds up development of the dashboard and ensures good accessibility out-of-the-box [2]. Alternatively, **Material-UI (MUI)** provides a comprehensive suite of polished components following Material Design, useful if you want a more structured design system for the app [3] [4]. Both libraries are well-maintained and integrate smoothly with Next.js.
- **Charting & Data Visualization (Recharts or Chart.js)** – For breeding statistics (e.g. trait distributions, lineage charts, mutation counts), use a dedicated charting library. **Recharts** is a React-based chart library built on D3 that makes it painless to create interactive charts as reusable components [5]. It supports responsive SVG charts (line graphs for stats over generations, bar charts for stat comparisons, etc.) with a declarative API [5]. This will help visualize dino stats, ancestry trees, and progress in a user-friendly way. If you need simpler graphs, **Chart.js** (with a React wrapper like *react-chartjs-2*) is another reliable choice for quick plots, though Recharts offers more flexibility for custom stat visuals.
- **File Upload & Sync UX** – Provide a smooth experience for uploading save files and managing sync status. Implement a drag-and-drop file uploader (e.g. using **React Dropzone** or a UI library's Dropzone component) so single-player users can easily upload their `.ark` save file. This improves UX by allowing drag-and-drop, file type validation, and progress feedback [6]. Next.js can handle the upload via an API route that stores the file or processes it. For server sync, include UI cues for status (e.g. a "Last sync time" indicator or a connect button for server admins). Ensure the frontend handles large file uploads gracefully – possibly by uploading to cloud storage or chunking if needed. The key is to make both manual uploads and automatic syncs intuitive: clear buttons or drop areas for file import, and a dashboard section showing incoming data from servers in real-time.

Backend/API (Server-Side Logic & Hosting)

- **Node.js (TypeScript) with Next.js API Routes or Express** – Given that the frontend is Next.js, using Node/TypeScript on the backend keeps the stack unified. Next.js API Routes allow you to write serverless endpoints directly in the Next app [1] – ideal for lightweight tasks like handling form submissions or triggering save parsing. For more complex processing, you might create a separate Express or **NestJS** service in Node. TypeScript is recommended to maintain type safety across the app. The familiarity of Node will speed up development of HTTP endpoints for mods

and any user account logic. If you prefer an all-in-one approach, Next.js API routes can handle the mod webhooks and file uploads initially, then scale out if needed.

- **Python FastAPI (for Save Processing Microservice)** – To leverage the mature ASA parsing libraries (see below), consider a small Python microservice. **FastAPI** is a high-performance Python web framework that is great for building RESTful APIs. It offers data validation and auto-generated docs, and its async support yields performance on par with Node [7]. You could have a FastAPI worker service that accepts uploaded files or data and uses Python libraries to parse stats, then returns JSON to the Node/Next API. This separation keeps the Next.js server responsive (offloading CPU-heavy parsing to Python) and takes advantage of FastAPI's ease of integrating Python code [7]. FastAPI's design ("convention over configuration" with flexibility) means minimal boilerplate to set up parsing endpoints [7].
- **Hosting Considerations** – For the Next.js frontend (and any Node API routes), **Vercel** is a natural choice (optimized for Next.js deployment, auto-scaling serverless functions, etc.). Vercel can host your SSR front-end and lightweight API routes for general use. However, long-running parsing tasks might hit serverless time limits, so deploy the parsing backend on a dedicated host or container platform. For example, use **AWS EC2 or ECS/Fargate**, **DigitalOcean App Platform**, or **Railway** to run a Docker container for the FastAPI service (or Node service) continuously. This backend would expose endpoints for uploading saves or receiving JSON from mods. Ensure CORS and auth are configured so the Next.js app (on Vercel domain) can communicate with the API service. As the user base grows, you can scale horizontally (multiple containers or serverless functions) behind a load balancer. This two-tier architecture (Next.js on Vercel + a dedicated API service) provides flexibility and scalability for many players.

Save File Parsing (`.ark`, `.arkprofile`, `.arktribe`)

- **ArkParse (Python)** – The most robust open-source library for ARK: Survival Ascended saves right now is **ArkParse** (available via PyPI as `ark-asa-parser`). ArkParse is a Python library specifically designed for ASA that can read, analyze, and even modify save files via an intuitive API [8]. It builds upon prior ARK parsing efforts and is actively maintained with ASA support. Using ArkParse, you can extract detailed info about creatures, players, tribes, and structures from `.ark` world saves, as well as from `.arkprofile` (player) and `.arktribe` files [8]. This is ideal for a breeding app because it gives you structured access to dino stats, ancestry, and mutations without having to reverse-engineer the binary format yourself. For example, ArkParse's Dino API lets you query creature stats and mutations directly [8], and it provides JSON export functionality for easy integration.
- **Integration Strategy** – To integrate ArkParse, you could run a **Python worker service** that handles parsing jobs. When a user uploads a save or when a server pushes new data, the Node backend can hand off the file (or filepath) to the Python service (via an HTTP request or task queue). The Python service would use ArkParse to parse the file and return structured JSON (e.g. list of creatures with their stats, or specific breeding-relevant info). This separation keeps your Next.js backend lean. You can also call Python scripts from Node (using a process spawn), but a web API or message queue (like Redis or RabbitMQ with a Celery worker) is cleaner for scale. If performance becomes critical, ArkParse could potentially be pre-loaded in a long-running process to avoid startup cost on each parse.
- **Alternative Libraries** – There is also a community Java-based library by Kakoen (`ark-sa-save-tools`) for ASA, but it's in an **unstable, incomplete state** and intended only for very experienced developers at the moment [9]. Another route is the older **ARK Savegame Toolkit** (originally in Java, with a C# port) used for Survival Evolved; however, it may not fully support ASA's changes out-of-the-box. Given ASA's recent release, the **Python ArkParse** library is currently the most feature-complete and actively maintained solution tailored to ASA. If you have expertise in Rust, you might consider writing a Rust parser (for performance and even

WebAssembly compatibility), but that would be a significant effort. In the short term, leaning on ArkParse's proven capabilities is the fastest path to support ASA save files.

- **Full-Stack Node Option** – If keeping everything in Node is a priority, you would likely need to use a lower-level approach or a Node library if one exists. Currently, ASA-specific Node libraries are scarce, so you might end up invoking an external tool. For example, you could use Qowyn's **ark-tools** as a CLI (it outputs JSON from save files) and call it from Node, or use the C# toolkit via a small .NET service. These add complexity in deployment. Thus, the recommended approach is the Python route for now, given its maturity for ASA. Over time, if ASA's formats stabilize and demand grows, a Node/TypeScript parsing library may emerge, at which point you could migrate to a full TS stack.

Live Sync from Mods/Server (Realtime Data Ingestion)

- **Unreal Engine 5 Blueprint HTTP** – To automatically sync live creature data from a dedicated server, one approach is writing an ASA mod that uses UE5's HTTP capabilities to POST data. Unreal Engine has an **"HTTP Blueprint"** plugin (currently experimental) that, when enabled, allows Blueprints to send web requests [10] [11]. You can create a server-side Blueprint (or use a *BlueprintCallable* C++ function) to gather dino stats periodically and send a JSON payload to your backend's endpoint (e.g. an endpoint like `/api/sync`). The Blueprint HTTP API is simple but note that it's still experimental, so you'd need to test thoroughly. This method is attractive because it can be done entirely with the ASA Dev Kit: minimal C++ needed if the blueprint nodes suffice.
- **ARK Server API (C++ Plugin)** – For a more powerful and headless solution, consider using the **ARK Server API** framework. The Server API is an open-source project that lets you write C++ plugins which run directly on the dedicated server (without requiring players to install a mod) [12]. It's widely used by server owners and supports ASA as well as older ARK versions [12]. Using this, you can write a plugin that hooks into creature tamed/spawn events or runs on a timer, and it can gather all necessary data (dino stats, player info) then send an HTTP POST to your web service. The advantage here is that it's server-side only (transparent to players) and likely more performant/robust. The plugin can use C++ networking (WinHTTP or libcurl) or even an embedded HTTP client library to communicate. Since ServerAPI is a well-known framework, you'll find community examples for things like sending data to webhooks. This route requires C++ knowledge and compiling against the ASA ServerAPI SDK, but yields a stable long-term solution.
- **Endpoint Design & Security** – Whichever method (Blueprint or C++ plugin) you choose, design a secure HTTP endpoint on your backend for data ingestion. Use HTTPS and require an **authentication token** or API key with each request (for example, generate a unique token for each server and have the mod include it in a header or URL parameter). That way, only authorized servers can push data, and you prevent random actors from hitting your endpoint. Implement basic **rate limiting** on the endpoint: for instance, if the mod sends data too frequently (e.g. multiple times per second), queue or drop some requests to avoid overload. A good practice is to have the mod send updates at a reasonable interval (maybe every few minutes for full dumps, or instantaneous for specific events only). The backend can respond with acknowledgment or instructions (you could even have the mod adjust frequency based on response).
- **Data Format & Validation** – Have the mod send JSON payloads in a predefined schema (e.g., an array of creatures with their properties, or diffs since last update). On the server, validate this data rigorously: check for expected field types/ranges (e.g. level >= 1, reasonable stat values) and sanitize any strings (to prevent any injection issues, though likely minimal in this context). By validating, you both protect against malformed data and ensure that only data from the actual game (not a tampered request) gets stored. Logging these sync requests (with timestamps and maybe partial data) is useful for debugging sync issues or detecting abuse.

- **Best Practices** – Document to your users (or server admins) that the mod/plugin will transmit game data externally. This transparency is important for player trust. Encourage using a separate **limited-access API user** or key for each server. Also, consider building a retry mechanism: if the backend is temporarily unreachable, the mod could queue data and retry a bit later to avoid data loss. By following these practices, the live sync feature will be robust, secure, and respectful of players' data.

Database & Storage (Creatures, Profiles, Tribes, Mutations)

- **PostgreSQL (Relational DB + JSON)** – A relational database is a strong choice for organizing creature data and enabling fast queries for the web dashboard. PostgreSQL in particular offers excellent performance, reliability, and a mix of relational and semi-structured capabilities [13]. You can design a schema with tables for **Creatures**, **Players**, **Tribes**, etc., capturing relationships (e.g. creature belongs to a tribe and an owner player). This makes it easy to do joins – for example, find all creatures owned by a certain player or all breeding pairs in a tribe. Postgres also supports JSONB columns, so you could store the raw stats blob or ancestry tree as JSON while still indexing key fields (like species, level, mutation count). This hybrid approach gives you **flexibility** for evolving data structures and **fast filtering** using indexes and SQL queries. For instance, you might index `species` and `level` columns on the Creatures table to quickly answer queries like "show all Rexes sorted by health stat". PostgreSQL has been used in countless data-heavy applications and will scale well for this scenario [13].
- **NoSQL Alternative (MongoDB or Document DB)** – If you prefer a schema-less design or the data model is highly flexible, a NoSQL document store is an option. Many game backends use document databases because they allow varying structures and can be more natural for hierarchical game data. In fact, non-relational databases can be **very flexible for games and often perform well** in those scenarios [14]. Using MongoDB, you could store each creature or even an entire save-state as a document. This would easily accommodate nested structures like a creature's full stat breakdown, ancestory, and inventory in one record. Queries for specific criteria (e.g. filter by species and mutation count) would require indexing those fields, which Mongo supports. The trade-off is that implementing cross-entity relationships (like linking a creature document to a player or tribe) is less straightforward than in SQL – you might duplicate some data or do application-side joins. **Recommendation:** if your data model is fairly structured (which ARK data tends to be: creatures always have the same stat fields, etc.), lean towards PostgreSQL for its robust querying and the ability to use strict relations *plus* JSON for any unstructured parts. You can always migrate or supplement with a NoSQL store later if you find the need for more flexibility.
- **Storage Format** – Regardless of DB engine, consider storing **snapshots vs current state**. For a web dashboard, you likely want the current state of the world (latest creature stats). Each sync or upload can upsert the records (update existing creatures or insert new ones). However, it might be valuable to keep historical data for trends – for example, tracking a dino's stat progression or how many creatures were on the server over time. In that case, you might maintain a history table or use a time-series database for metrics. Initially, focus on the current state for simplicity. Use transactions when updating multiple tables (e.g. adding a new creature and linking it to a tribe) to keep data consistent. Also, take advantage of **indexes** heavily: creature species, name, owner, and tribe are likely query filters, so index those for quick lookups. Postgres can even do GIN indexes on JSON fields if you query deep nested stats. By designing the DB with both **flexibility** (JSON fields or a schema that can evolve) and **indexability**, you ensure the breeding manager can handle complex filtering (like "show all male Rexes with melee stat > X and mutation count < Y") swiftly on the dashboard.

Security & Player Consent

- **Authentication & API Tokens** – **Secure all data endpoints** to prevent unauthorized access or tampering. For the mod syncing endpoint, require a secret token in the HTTP header or payload that only your server and the mod/plugin know. This could be a server-specific API key that the admin generates on your web app and configures into the mod. The backend should check this token on every request and reject any missing or invalid tokens. This ensures only approved servers send data. Similarly, if you implement user login (e.g. a player logs in to view their dinos), use a proper authentication system (NextAuth or JWT-based auth). Tie the data access to identity: for example, a server admin account can see all data for their server, whereas an individual player might only see data related to their own characters if you expose such functionality.
- **Authorization & Consent** – Since the app deals with potentially sensitive player data (dino ownership, player profiles, etc.), make sure you have **player consent** where appropriate. If a server admin is using the tool, ideally players on that server should be informed (perhaps via server rules or the mod description) that their game data is being uploaded to an external service. For deeper consent, you could implement a system where players authenticate (maybe via Steam OpenID or an in-game command) to "claim" their profile on the web app – this way they explicitly opt in to viewing their data. In any case, **never display private player info** (like Steam IDs or personal profiles) to the public without permission. Only the server owner or the player themselves should see detailed stats. Aggregate data (like total creatures) is fine, but be cautious with anything that could be considered personal.
- **Transport Security (HTTPS)** – All communication between the mod and your backend, and between users and the Next.js app, **must be over HTTPS**. This prevents eavesdropping or man-in-the-middle attacks, which is especially important if auth tokens or any personal data are in transit. If hosting the backend on your own, use Let's Encrypt or a managed certificate. Vercel and most cloud platforms make HTTPS straightforward – ensure it's enforced.
- **Rate Limiting & Abuse Prevention** – Implement basic rate limiting on APIs to prevent abuse or accidents. For example, if a buggy mod started spamming your endpoint, the backend should throttle it (you can use an in-memory rate limiter or a service like Cloudflare in front). This protects your server and database from overload. Also, validate input sizes – a .ark save can be tens of MBs; if someone tried to upload an absurdly large file or malicious data, have caps in place (and return a clear error). Use streaming or background processing for large files to avoid tying up server threads.
- **Data Protection** – Store any sensitive keys or tokens securely. For instance, if you have a database of server API keys, hash them or store them in a secure secrets vault. Regularly audit your database for any sensitive fields (like player names or IDs) and consider if they need encryption at rest. Back up your database periodically, but also **purge data on request** – if a server or player wants their data removed, you should comply (especially if you plan to make this a public service, you'll need a privacy policy).
- **Transparency with Users** – Clearly communicate how the data is used. If you provide a UI or documentation, explain that the tool reads game save files or live data and list what it stores (creature stats, player/tribe names, etc.). This will build trust and also cover you from a compliance perspective. You may also implement an **opt-out** or deletion mechanism – e.g. if a player doesn't want their data shown, a server admin could flag them to be excluded from sync (the plugin could then skip those players' data). While this might not be strictly required, it's a good community-friendly feature.
- **Testing & Hardening** – Before going live, test the security by simulating what could go wrong: invalid files, hostile JSON from a fake source, many requests at once, etc. Use tools or even attempt a small **penetration test** on your API (there are linters and scanners for common API vulns). Since you're handling game files, also be mindful of **path traversal or injection** – e.g. don't blindly use a filename given by a user to read files on your server. Use safe file handling

(store uploads in a separate directory, never in a path that could execute code). By taking these precautions, you'll create a trustworthy platform that players and admins can adopt with confidence.

In summary, the recommended stack is a **Next.js** front-end with a rich React component library (for productive UI development) and a robust charting library for visualizing breeding stats. On the backend, use **Node/TypeScript** for the web API (especially convenient if integrated with Next.js), but delegate the heavy lifting of save parsing to a dedicated service using **Python + ArkParse** – this taps into the latest ASA parsing capabilities [8] without reinventing the wheel. For live data, leverage ASA's modding APIs: either Blueprint HTTP calls or the **ServerAPI C++ plugin** approach to push data securely to the backend. Store the data in a **PostgreSQL database** for reliable relational queries and fast filters, possibly mixing in JSON for flexibility. Throughout all components, enforce strong security practices (HTTPS, auth tokens, validation) and respect player consent and privacy. This stack will be flexible enough to handle both manual file uploads and automated sync, will play nicely with Next.js, and will scale as your user base grows – all while providing a smooth, feature-rich experience for ARK breeders.

Sources:

- Next.js API Routes (Node.js serverless functions) [1]
- Chakra UI – accessible React component library [2]
- Recharts – composable React + D3 chart library [5]
- Drag-and-drop file upload UX (Supabase Dropzone example) [6]
- FastAPI framework benefits (Python, type validation, performance) [7]
- PostgreSQL reliability and familiarity [13]
- ArkParse library for ASA save files (Python) [8]
- Kakoen's ASA save tools (Java, experimental state) [9]
- Unreal Engine Blueprint HTTP plugin (enable Experimental HTTP) [10] [11]
- ARK ServerAPI for server-side plugins (open-source, supported for ASA) [15] [12]
- NoSQL flexibility for game data (general context) [14]

---

[1] Pages Router: API Routes | Next.js
https://nextjs.org/learn/pages-router/api-routes

[2] [3] [4] Best UI Libraries for Developers (trust me, I'm a dev) - DEV Community
https://dev.to/bibschan/best-ui-libraries-for-developers-trust-me-im-a-dev-2f2i

[5] GitHub - recharts/recharts: Redefined chart library built with React and D3
https://github.com/recharts/recharts

[6] Dropzone (File Upload)
https://supabase.com/ui/docs/nextjs/dropzone

[7] [13] Full Stack Next.js, FastAPI, PostgreSQL Tutorial - Travis Luong
https://www.travisluong.com/full-stack-next-js-fastapi-postgresql-tutorial/

[8] GitHub - VincentHenauGithub/ark-save-parser: A python library to parse ark saves
https://github.com/VincentHenauGithub/ark-save-parser

[9] GitHub - Kakoen/ark-sa-save-tools: Parses Ark: Survival Ascended save files
https://github.com/Kakoen/ark-sa-save-tools

[10] [11] How to send an HTTP Request from an Unreal Blueprint? - Game Development Stack Exchange
https://gamedev.stackexchange.com/questions/208929/how-to-send-an-http-request-from-an-unreal-blueprint

12 15 Official - Ark: Survival Ascended - ServerAPI (Crossplay Supported) | Game Servers Hub
https://gameservershub.com/forums/resources/ark-survival-ascended-serverapi-crossplay-supported.683/

14 architecture - noSQL - Is it a valid option for web based game? - Game Development Stack Exchange
https://gamedev.stackexchange.com/questions/5316/nosql-is-it-a-valid-option-for-web-based-game