ChatGPT

Project Overview

We plan to build a **JAMstack-based educational website** focusing initially on IB subjects (Analysis & Approaches HL, Physics HL, Chemistry HL). The site will start as a fast **static landing page** introducing our platform, with a waitlist signup and contact form. Behind the scenes we'll integrate cloud services for dynamic features: e.g. **Firebase** for hosting, auth, and database, and **Stripe** for payment subscriptions. The goal is to outline each component and process – from frontend to backend – with an eye on scalability and future growth.

Technology Stack & Architecture

- **Static Frontend:** Use a modern static site framework (e.g. Next.js, Gatsby, or Hugo) to generate pre-built HTML/CSS/JS. Static hosting on a **CDN** (such as Firebase Hosting, Netlify, or Vercel) ensures **global distribution and fast loading**, since "static files scale effortlessly across global CDN networks" [1]. This CDN-backed approach gives excellent performance and security (no server-side page rendering) [2] [1].
- **Client-Side Dynamics (JAMstack):** Any interactive features (forms, dashboards) run in the browser using JavaScript. We'll call APIs for dynamic data. This follows the **JAMstack pattern**: pre-built Markup + client-side JavaScript + external APIs [3]. For example, the waitlist form and contact form will POST to a backend API (Firebase function) without needing a traditional server.
- **Firebase Backend:** Use Firebase for authentication, database, and serverless functions. Specifically:
- **Authentication:** Use Firebase Auth (email/password or OAuth) to manage user accounts. Admin users (the 3 founding team members) get elevated roles.
- **Database:** Use Cloud Firestore (NoSQL) to store dynamic data: user profiles, invite codes, subscriber status, and collected emails/messages. Firestore is fully managed and "auto-scales" globally [4], which means as user/data volume grows, it expands seamlessly without manual sharding.
- **Cloud Functions:** Use Firebase Cloud Functions to implement any server-side logic – e.g. processing form submissions, handling Stripe webhooks, or generating invite codes. These are event-driven and scale with demand. (For example, a function can listen to Stripe payment events and update user subscription status in Firestore.)
- **Hosting:** Deploy the static site on Firebase Hosting (or a similar provider). Hosting serves the pre-built files over a CDN and can easily integrate with the Cloud Functions.
- **Stripe Integration:** Use Stripe for handling subscriptions. We'll configure Stripe Products and Plans for our paid tiers. Payment processing should happen on the server-side for security. Firebase has a guided tutorial: you can "process payments with Firebase and Stripe without building your own server infrastructure" [5]. In practice, the frontend will collect payment details (via Stripe Elements or Checkout) and send a token to a Cloud Function which then creates the subscription using Stripe's secret API key [6] [5]. The secret key is kept only on the server (Cloud Function) for safety [6].
- **Invitation/Beta Access:** We want an invite-code system for early testers. We can generate secure random codes and store them in Firestore. A good practice is to structure them by invitee email, so only the intended user can redeem it [7]. For example:

```
inviteCodes {
  invitee: {
    userB_email: { code: XYZ123, inviterId: abc, generator: "system" }
  }
}
```

Security rules in Firestore will ensure only the user owning the invite code (or its creator) can use/read it [7]. During registration, we'll check the provided code against this collection before creating the account. This ensures uninvited users cannot sign up.

Initial Landing Page

We will create a simple yet engaging **landing page** to introduce the platform. Key elements:
- **Headline & Description:** Briefly explain the platform's purpose (e.g. "Free IB HL Resources and Tutoring").
- **Email Waitlist Form:** A signup form to collect emails of interested users. The form will use a bit of client-side JS to POST submissions to Firebase (e.g. a Cloud Function or directly to Firestore). As one source suggests, a static page can "use Firebase to not only collect emails but even feedback" [8]. On submit, we push the email into Firestore (under a waitlist collection). Firebase's free tier even provides **100 MB of storage** – enough for a small waitlist and messages [9].
- **Contact Us Form:** Simple form (name/email/message) that similarly saves messages to Firestore or triggers an email. The Firebase landing-page model uses JS to push contact entries into the database [10].
- **Newsletter/Automation:** Optionally integrate a mailing service (like Mailchimp) via API, but initially Firestore storage suffices.
- **Meta & SEO:** Include meta tags, fast images, and ensure the static page is lightweight. Performance is crucial: static pages have no runtime DB queries, so "faster load times" and better Core Web Vitals [11].

Example of a static landing page design (modern CSS, minimal JS). Static pages served via CDN enable blazing-fast load times and scale globally [1] [11].

Core Features & User Workflow

1. **User Accounts & Roles:**
2. **Students:** Eventually, students will create accounts (email/password or Google) to access subject content. Initially, we may restrict signups via invite codes or admin approval. Once live, we'll implement normal signup.
3. **Administrators:** The three founding team members have admin privileges. We can mark them by a field in Firestore (e.g. `role: "admin"`) or use Firebase custom claims. Admins can manage content, view signups, and generate invite codes.
4. **Subject Content Management:**
5. Create pages or a small CMS for each subject (AA HL, Physics HL, Chem HL) with topics, notes, or videos. Use a clear route structure (e.g. `/subjects/physics-hl/`). This can be done with the static site generator (Markdown files for each topic) or fetched from Firestore if content is dynamic. We will **prepare placeholders** for other subjects (AA SL, AI SL/HL, CS HL/SL, etc.) so the site can expand.
6. Each subject section will check if the user has subscribed or has an invite code before showing premium content. Unauthenticated users might see teaser or blocked message.
7. **Subscription Gatekeeping:**

8. Integrate with Stripe to lock premium content. When a user subscribes successfully, the Cloud Function updates their Firestore profile (e.g. `activeSubscription: true`). Frontend pages check this flag.
9. Use Stripe Webhooks via Cloud Functions: upon successful payment, mark user as "paid." On cancellation, remove access.
10. **Invite Code Access:**
11. For beta, implement the invite code check at signup (via a Cloud Function or during client-side registration). Only accounts with a valid code can be created. The earlier [33 † L225-L233] structure ensures codes are one-time use or tied to emails.
12. **Email & Analytics:**
13. Send welcome emails on signup (via Firebase Extensions or a Cloud Function with SendGrid).
14. Integrate Google Analytics or Firebase Analytics to monitor traffic/performance.

Scalability & Performance

- **Global CDN:** Hosting the static files on a CDN (Firebase Hosting or similar) means the site will automatically scale to thousands of visitors. No extra servers needed as traffic grows. Static delivery "eliminates multiple points of potential failure" and loads from edge caches [1] [11].
- **Database Scaling:** Firestore auto-scales as data grows [4]. We should follow Firestore best practices (no super-large documents, use indexes) for efficiency [4]. Data like waitlist or user profiles will grow slowly.
- **Cloud Functions:** These scale horizontally. We need to monitor quotas (Firebase has limits, but Blaze pay-as-you-go plan will cover small usage initially). If traffic spikes, functions can scale, though cold starts may occur. We should keep functions lightweight.
- **Optimize Assets:** Minify JS/CSS, compress images. Use responsive images. Since performance is key, this is standard static-site practice. (See [14 † L269-L278] on performance and web vitals.)
- **Monitoring:** Use Firebase Performance Monitoring and Logging to track slow queries or function errors.

Development Process & Tools

- **Version Control & CI/CD:** Use GitHub (or similar) for code. Set up a CI pipeline (GitHub Actions) that runs on push to main: build the static site, run tests/lint, then deploy to Firebase Hosting. The Sealos guide even shows a GitHub Action example for static sites [12].
- **Environment Management:** Keep API keys (Stripe secret key, Firebase credentials) in secure env vars. Stripe's secret key is only used in Cloud Functions (server-side). The publishable key is in the frontend code. Never expose secret keys in client JS [6].
- **Local Development:** Use the Firebase emulator suite for testing functions and Firestore locally.
- **Iterative Launch:**
- Phase 1: Deploy static landing page, waitlist & contact via Firebase. Open invite-only beta signups.
- Phase 2: Add authentication and an admin interface (even a simple script) to generate invite codes and review submissions.
- Phase 3: Implement Stripe subscription flow for initial subjects, protect content pages.
- Phase 4: Expand subjects, refine UI/UX, optimize and monitor.
- **Security:** Static sites inherently reduce attack surface (no databases or exec at request time [2]). Still, enforce Firebase Security Rules: e.g. only authenticated users update their own data, only admins edit content. Use HTTPS everywhere (Firebase Hosting provides SSL by default).

Team Roles

With 3 admins initially, we can divide responsibilities: e.g. one handles frontend/content design, another backend/infrastructure (Firebase/Stripe), and another marketing/community. Regular planning sessions should track tasks (e.g. using Kanban or GitHub issues).

Summary

By using a **JAMstack + Firebase + Stripe** approach, we can quickly build a responsive, secure, and scalable site. The initial static landing page collects interest and contact info, while Firebase and Stripe manage auth, database, and payments behind the scenes [5] [8]. Static hosting ensures we can handle high traffic easily (static files "scale effortlessly" on a CDN [1]). As we grow, this architecture allows adding more subjects, increasing users, and upgrading backend capacity without major rework. All security-sensitive logic (invite validation, payment processing) is handled in Firebase Cloud Functions on the server side. This plan covers the end-to-end flow from user landing on the page, to signing up (potentially via invite code), to subscribing via Stripe, ensuring a smooth and maintainable development path.

Sources: Technical references and best practices from Firebase and JAMstack documentation [5] [11] [8] [7], plus community advice on static sites and invite code management.

---

[1] [2] [3] [11] [12] What Is a Static Site? Complete Guide to Static Website Development 2025 | Sealos Blog
https://sealos.io/blog/what-is-a-static-site

[4] Scaling Firebase - Practical considerations and limitations
https://ably.com/topic/scaling-firebase-realtime-database

[5] Process payments with Firebase
https://firebase.google.com/docs/tutorials/payments-stripe

[6] Is it possible to use Stripe for subscriptions with static website + per user storage? - Stack Overflow
https://stackoverflow.com/questions/77952339/is-it-possible-to-use-stripe-for-subscriptions-with-static-website-per-user-st

[7] angularjs - Manage invite codes for registration with Firebase - Stack Overflow
https://stackoverflow.com/questions/28252862/manage-invite-codes-for-registration-with-firebase

[8] [9] [10] Startup Landing Pages with Firebase
https://www.fizerkhan.com/blog/posts/startup-landing-pages-with-firebase