ChatGPT

Project Plan and Architecture

Tech stack & architecture: We will build a JAMstack-style static site (pre-built HTML/CSS/JS) hosted on Firebase Hosting, using client-side JavaScript and APIs for dynamic features [1] [2]. Firebase Hosting serves static content via a global CDN with SSL by default [3] [4], giving fast load times and easy scalability. We'll use Firebase Authentication (for user accounts), Cloud Firestore (NoSQL database), and Cloud Functions (serverless backend code) for custom logic. Static pages (e.g. subject content pages) are pre-rendered for performance [1] [5], while interactive features like subscriptions and invite-code checks will be handled by Firebase Cloud Functions and APIs. This decoupled setup (front end on CDN + backend logic via APIs) maximizes speed, security and scalability [1] [2].

- **Static content** (subject pages, landing page, FAQ, etc.) will be built with a static site generator (e.g. Next.js, Gatsby or similar) or plain HTML/CSS if simple. Large content (courses for Physics HL, Chemistry HL, etc.) can be maintained as Markdown or via a headless CMS, which lets editors update material without writing code [6] [7].
- **Dynamic features:** Firebase Cloud Functions will handle Stripe payment processing (no custom servers needed) and any backend logic (e.g. enforcing invite codes). As one guide notes, "integrating Stripe with Firebase means that we will create several Cloud Functions…including a webhook for accepting incoming Stripe data" [8]. This allows us to build a full subscription system using Firebase/Stripe out-of-the-box [9] [8].
- **Database:** Use Firestore to store user profiles, subscription status, subject progress (if any), and invite codes. Data is synced securely and scales with usage.
- **Tools & workflow:** We'll use Git for source control and set up a CI/CD pipeline (e.g. GitHub Actions) that triggers `firebase deploy` on merge. This automates building the static site and deploying to Firebase Hosting/Functions. Testing and linting can be part of the pipeline to ensure quality.

  Jamstack architecture: By pre-rendering pages and serving them via CDN, we avoid on-the-fly server rendering, making the site very fast and easy to scale [1] [5]. Any dynamic functionality (auth, payments, form submissions) is handled by JavaScript calls to Firebase APIs or third-party services. This "static front-end + APIs" approach lets us focus on features without managing servers.

Core Features and Content

- **Subject Content Pages:** The site will initially feature content for Physics HL and Chemistry HL (as those resources are ready). Each subject will have its own section with static pages (notes, videos, quizzes, etc.). We will design the content structure so that adding new subjects (e.g. Mathematics (AA SL), Biology (A/B SL/HL), Computer Science SL/HL, etc.) is straightforward. Essentially, subjects are modular sections of the static site.
- **User Authentication & Access:** Users can sign up and sign in via Firebase Auth (email/password or social login). During initial development, we'll require an **invite code** to sign up. Invite codes are securely stored in Firestore so only the intended user can use them [10]. For example, the database could have an `inviteCodes` collection keyed by the invited user's email, so that checking a code does not expose all codes [10]. Invite codes enable us to control who gets early access (e.g. beta testers). Admins can generate these codes.

- **Subscription System:** We will integrate Stripe for paid subscriptions. Using Firebase Functions, we'll create payment flows without any custom server. (Firebase's own docs show that "using Firebase features and Stripe, you can process payments in your web app without building your own server infrastructure" [9].) We'll set up Stripe products/prices (e.g. monthly/yearly plans) and a checkout page. On success, a Cloud Function webhook will update the user's Firestore record. The frontend can then enable premium content. We can also use the Stripe Customer Portal for subscription management.
- **Content Access:** Non-subscribers (or new sign-ups with invite code) will have limited access – e.g. only landing page or demo content. Subscribers get full access to all subject materials. Invite-code users could be automatically upgraded (e.g. marked as "premium tester") in Firestore upon entering a valid code.
- **Admin Panel:** The 3-person admin team can manage the site via Firebase Console or a custom admin interface. Roles might include: Content Lead (updates subject materials), Tech Lead (handles deploys, invites, Stripe), and Operations (handles marketing, user support). Admins will have elevated Firebase Auth roles or a flag in the database. They can create new subject pages (by updating the repository or CMS), view user/subscription stats in Firebase Analytics, and manage invite codes and subscriptions through Stripe's dashboard.

Landing Page with Waitlist and Contact

Figure: Example of a concise "coming soon" landing page with email sign-up form. The initial landing page will present a clear **value proposition** and invite visitors to join our email waitlist. Best practices suggest a simple hierarchy: a prominent headline and subheading explaining what the service is and why it's valuable [11] [12]. Below the header we'll add 3 bullet points or brief lines highlighting key benefits (e.g. "High-quality IB Physics & Chemistry resources," "Trusted by experienced tutors," "Personalized revision tools") [11]. A compelling image or graphic can reinforce the message. Finally, a bold **call-to-action** button (e.g. "Join Waitlist" or "Get Early Access") will trigger an email signup form.

In practice, the landing page will remove unnecessary navigation and focus only on conversion [12] [13]. It will include: - **Headline/Subheadline:** Describe the site's purpose (e.g. "Advanced IB Science Tutoring – Launching Soon").
- **Features/Benefits:** A few short bullet points or phrases that explain what's coming.
- **Email Waitlist Form:** An input for name/email and a submit button ("Notify Me" or similar). Joining the waitlist means we'll email updates or beta invitations. As marketing guidance notes, a good landing page "convinces [visitors] it's worth it to provide personal details" by clearly presenting the offer [12]. We might incentivize sign-ups by promising early access or discounts for subscribers. - **Contact Info:** A "Contact Us" link or form so interested users (or schools/teachers) can reach out with questions. This could simply be an email link or a basic form (also powered by Firebase Functions).

This landing page will be a static HTML page hosted alongside our app. Because it's the primary page during development, it should load extremely fast (static assets on CDN) and be mobile-responsive.

Waitlist & Contact: Submitted emails will be saved (e.g. in Firestore) for future contact or newsletter. We might integrate a lightweight email service or use Firebase itself to store/notify. The contact form (if used) will send messages to our admin email via a Cloud Function. These forms are straightforward and can be added without complex backend.

Admin Roles and Team

We will have 3 admins to start. Each admin account will be registered in Firebase Auth with an "admin" role flag. Typical responsibilities: - **Admin 1 (Content Lead):** Manages educational content (uploads notes, updates subject pages, adds new subjects). Uses code repository or CMS to deploy changes. - **Admin 2 (Tech Lead):** Manages Firebase project settings, deploys updates, sets up CI/CD, handles invites and technical integrations (Stripe, security rules). - **Admin 3 (Operations):** Oversees marketing (runs the waitlist campaigns), customer support (responds to "Contact Us" queries), and monitors site analytics. Also helps with user feedback and iteration.

The admin panel itself might just be the Firebase console and our own dashboards. For example, we can create a simple "Admin Dashboard" page (accessible only to admins) to view sign-ups, moderate content, or upload resources. This is optional to implement initially, but at minimum admins can use Firebase's built-in dashboards (Authentication, Firestore data, Hosting logs, etc.) [14].

Subscription and Invite System

- **Stripe Setup:** We'll create Stripe products (e.g. "Monthly Plan" and "Annual Plan"). Prices and plans are configured in Stripe's dashboard and referenced in our code. Using Firebase Functions, we'll expose the list of products/prices to the frontend so users can choose a plan [8]. A function (e.g. `createSubscriptionSession`) will handle Checkout sessions by calling the Stripe API and returning a secure payment URL.
- **Handling Webhooks:** A Firebase Function will listen to Stripe webhooks (payment success, subscription updates) and update our Firestore. For example, on successful payment it marks the user as active subscriber. This requires no traditional server – just deploy the function via Firebase CLI [9] [8].
- **Invite Codes:** Before the public launch, new users must enter a valid invite code to register. Invite codes (random strings) are stored in Firestore. One approach (from community guidance) is to store codes under an `inviteCodes` document keyed by the invited person's email, so that upon signup the app checks only the matching entry [10]. Security rules ensure one can't list all codes. After using a code, it can be marked "consumed" so it can't be reused. Admins generate and distribute codes to beta users.
- **Free Access for Invited Testers:** Initial testers (with codes) can gain provisional access to content even without paying, which helps us gather feedback. We may give them a temporary "premium" status in Firestore so they can explore the site.

Scalability & Performance

Because we're using static hosting and serverless backend, the site is inherently scalable:
- **Static files on CDN:** Each page is a pre-built HTML/JS file delivered from a CDN. Traffic spikes are handled by the CDN automatically, with minimal latency worldwide [5] [4]. Static assets can be aggressively cached and the site is safe from load spikes (no server bottleneck).
- **Serverless scaling:** Firebase Functions and Firestore scale on demand. We pay for usage, so initial costs are low, and as users grow, Google will automatically allocate more resources. There's no need to provision servers.
- **Cost efficiency:** Static sites and managed services mean we don't run expensive servers. As one source notes, static sites "require fewer server resources and less maintenance" [15]. Using Firebase's Blaze plan, we get generous free tiers and pay-per-use.
- **Performance:** Pre-rendered pages mean very fast first-load times, which improves SEO and retention

[16] [5]. We avoid database calls or heavy server logic on page load. Any dynamic call (like checking subscription) happens in the background via lightweight API requests.

Overall, this architecture is well-suited for growth: adding a new subject means creating new static pages (or adding CMS entries) and doesn't strain the infrastructure. The combination of CDN, Firebase, and Stripe has been proven to support production SaaS apps with minimal engineering overhead [9] [8].

Development Workflow

1. **Setup Firebase Project:** Create a new Firebase project, enable Hosting, Authentication, Firestore, and Functions. Upgrade to the pay-as-you-go plan for scalability.
2. **Design Landing Page:** Create a simple HTML/CSS design (or use a builder/template) for the landing page, following the layout discussed above [11] [12]. Implement the waitlist form (hooked to Firebase). Host this page at the root domain while the main site is under development.
3. **Implement Waitlist and Contact:** Add a form to save emails (via Firestore) and a contact form (via Firebase Function to send email or store message). Test these forms end-to-end.
4. **Build Static Site Framework:** Choose an SSG or static project structure. Set up the directory for subject pages. Scaffold pages for Physics HL and Chemistry HL content (even if initially mostly placeholder text). Ensure easy navigation between them.
5. **Auth & User Flow:** Implement Firebase Auth sign-up/login. During signup, require entry of a valid invite code (check Firestore). After login, track the user in a `users` collection.
6. **Stripe Integration:** Configure Stripe account (products/plans). Write Cloud Functions for creating checkout sessions and handling webhooks [8]. Create a "Subscribe" page where logged-in users can view plans and subscribe. Test subscriptions in Stripe's test mode.
7. **Access Control:** Secure routes/content. For example, use Firebase rules or app logic so that premium content is only visible to subscribed users or invited testers.
8. **Admin Dashboard (optional):** Create a protected admin page where admins can, say, view new signups or generate invite codes. Or simply use Firebase Console for these tasks.
9. **Testing & QA:** Throughout, test on different devices. Make sure content loads quickly (use Lighthouse audits). Test Stripe flows and form submissions.
10. **Deployment Pipeline:** Set up a CI/CD pipeline: on each push to the main branch, build the site and run `firebase deploy` (Hosting + Functions) automatically. Use Firebase's preview channels for testing pull requests if needed [17].

By following this plan and leveraging Firebase/Stripe services, we avoid reinventing core infrastructure. We focus on the **process**: design -> implement landing page -> build core features -> test -> iterate. Each component (Auth, DB, Payments, Hosting) is mostly turnkey, so we can "wire things up" rather than write servers from scratch. This modular approach ensures every piece is scalable, maintainable, and secure.

Sources: Best practices for static sites and landing pages [16] [11] [12], and official Firebase/Stripe guides on serverless payments [9] [8] inform this plan. These sources emphasize high performance, security, and ease of deployment for architectures like ours.

---

1 What Is Jamstack? Everything To Know | Naturaily
https://naturaily.com/blog/what-is-jamstack

2 5 6 7 15 16 What is a static website? Learn why it’ s perfect for speed and security | Contentstack
https://www.contentstack.com/blog/all-about-headless/what-is-a-static-website-learn-why-its-perfect-for-speed-and-security

3 4 14 Get started with Firebase Hosting
https://firebase.google.com/docs/hosting/quickstart

8 Implementing Stripe Subscriptions with Firebase Cloud Functions and Firestore - DEV Community
https://dev.to/thraizz/implementing-stripe-subscriptions-with-firebase-cloud-functions-and-firestore-4iji

9 Process payments with Firebase
https://firebase.google.com/docs/tutorials/payments-stripe

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

11 13 How To Design A High-Converting Waitlist Landing Page
https://www.getresponse.com/blog/waitlist-landing-page

12 What is a Landing Page & Why is it Important?
https://www.wsiworld.com/blog/the-importance-of-landing-pages

17 Deploy to live & preview channels via GitHub pull requests - Firebase
https://firebase.google.com/docs/hosting/github-integration