Tech Lead Focus Documents

Time Spent to Build: A 7-Hour Journey

This prototype was brought to life in approximately **7 hours**, broken down into distinct development phases.

- **Phase 1: Project Scaffolding & Database Design (1.5 hours)**
  - Initialized Next.js project with TypeScript and Tailwind CSS.
  - Designed and implemented the PostgreSQL schema using Prisma (`Lesson`, `Problem`, `User`, `UserProfile`, `UserStreak`, etc.).
  - Set up Hono for RPC-style API routes and established the NextAuth.js authentication flow.
- **Phase 2: Core API & Feature Implementation (3 hours)**
  - Built the core API endpoints for fetching lessons, submitting answers, and tracking progress.
  - Developed the idempotent answer submission logic to prevent duplicate XP awards.
  - Implemented the business logic for the XP system and daily streak tracking (`XpManager`, `Streak-Counter`).
- **Phase 3: UI/UX Development & Componentization (1.5 hours)**
  - Built the main UI pages: Dashboard, Lesson View, Profile, and Sign-In/Sign-Up.
  - Developed reusable components (`LessonCard`, `ProblemSolver`, `LevelProgressBar`) using shadcn/ui primitives.
  - Integrated React Query for seamless data fetching and state management on the client.
- **Phase 4: Refactoring, Documentation & Testing (1 hour)**
  - Refactored the project into a feature-sliced architecture (`/features`).

- Wrote comprehensive documentation, including onboarding guides and development guidelines.
- Added foundational unit and integration tests to ensure code reliability.

Team Development Strategy

Codebase Structure for Parallel Development

The current **feature-sliced architecture** is designed explicitly for team collaboration.

- **How it works:** All code related to a specific feature (e.g., `lessons`, `profile`, `leaderboard`) is co-located in its own directory under `/features`. This includes API endpoints, frontend hooks, and feature-specific components.
- **Benefit:** Developer A can work on `features/leaderboard` while Developer B works on `features/profile` with minimal risk of overlap or merge conflicts. Shared logic resides in `/lib` and shared UI in `/components`, which are modified less frequently.

Git Workflow & Code Review

We use a simple and rapid **feature-branch workflow**.

1. **Branching:** Create a new branch from `main` for every new task (e.g., `feature/real-time-leaderboard`).
2. **Pull Requests (PRs):** Once a feature is complete, open a PR against the `main` branch. The PR description should clearly explain the "what" and "why" of the changes.
3. **Code Review:**
   - Each PR requires at least **one approval** from another developer.
   - **Automated checks** (linting, testing via GitHub Actions) must pass.
   - Reviews focus on logic, readability, and adherence to our development guidelines.
4. **Merging:** Once approved and checks pass, the PR is squashed and merged into `main`. This keeps our git history clean.

Preventing Merge Conflicts

- **Architectural Prevention:** The feature-sliced structure is our primary defense.
- **Communication:** A quick daily stand-up or chat message ("I'm starting work on the shared `Navbar` component") prevents concurrent modifications.
- **Small, Frequent PRs:** Breaking down large features into smaller, manageable PRs reduces the surface area for conflicts.
- **Rebase Regularly:** Developers should frequently rebase their feature branches with the latest changes from `main` (`git pull --rebase origin main`) to resolve conflicts early.

AI/ML Integration Strategy for Personalization

Where to Integrate AI/ML

The core opportunity is to create a **truly adaptive learning path** for each student. The system should identify a student's strengths and weaknesses and dynamically adjust the curriculum to maximize learning efficiency.

Data Collection

We need to collect granular data on student performance:

- `ProblemID` and `LessonID`
- `UserID`
- `UserAnswer` (what they submitted)
- `IsCorrect` (boolean result)
- `TimeTaken` (ms to answer)
- `Attempts` (how many tries before correct)

Implementation Ideas

1. **LLM for Problem Tagging:**
    - **Concept:** Use a Large Language Model (LLM) in a one-off batch process to analyze the text and context of every problem in our database.

- **Action:** The LLM would automatically assign metadata tags to each problem, such as `{"concept": "linear-equations", "sub-concept": "slope-intercept-form", "difficulty": "medium"}`. This enriches our content without manual effort.

2. **Dynamic Strength/Weakness Profiling:**
   - **Concept:** As a student answers questions, we build a profile of their performance against the AI-generated tags.
   - **Action:** If a student has a <60% success rate on problems tagged `"sub-concept": "factoring-trinomials"`, we identify this as a weakness. Conversely, a >90% success rate on `"concept": "basic-arithmetic"` indicates a strength.
3. **Personalized "Workout" Sessions:**
   - **Concept:** Based on the weakness profile, the system can generate custom-tailored quizzes.
   - **Action:** A student struggling with fractions would get a "Fraction Workout" session, mixing problems they previously failed with new, similar problems to reinforce the concept.
4. **Difficulty-Weighted XP System:**
   - **Concept:** Not all problems are created equal.
   - **Action:** Use the AI-generated `difficulty` tag to adjust XP rewards. A `hard` problem could be worth 25 XP, while an `easy` one is worth 10 XP, further incentivizing students to tackle challenges.
5. **Curriculum & Syllabus Alignment:**
   - **Concept:** Use an LLM to analyze a student's answer patterns against a known educational standard (e.g., Common Core, Kurikulum Merdeka).
   - **Action:** The system could provide feedback like, "You've mastered 80% of the Grade 7 algebra syllabus. Your next focus should be on geometry concepts to complete the curriculum."

Technical Communication Example

Most Complex Feature: Idempotent Answer Submission & Streak Logic

This is the `POST /api/lessons/:id/submit` endpoint, which processes a user's answer, awards XP, and updates their daily streak.

Explanation for a Non-Technical Founder

- **What It Is:** "This is the brain behind our rewards system. When a user answers a question, this system securely validates their answer, adds points (XP) to their profile, and checks if they've continued their daily learning streak. We've built it to be **idempotent**, which is a fancy way of saying that if a user's internet glitches and sends the same answer twice, they only get points once. It guarantees fairness."
- **Business Impact:** "This system is **critical for user trust and retention**. If users lose a streak unfairly or don't get the points they earned, they become frustrated and may stop using the app. A rock-solid reward system keeps the game fun and fair, which is key to building a daily habit and keeping users engaged."
- **Trade-offs:** "We could have built a simpler version faster, but it would have been prone to errors. These errors would lead to user complaints, support tickets, and require us to manually fix user data. We invested extra time upfront to build this robustly, saving us significant time and protecting our user experience in the long run."
- **Timeline:** "The core logic for this took about 3-4 hours of focused development and testing. This was a high-priority task because it's the foundation of our entire gamification loop."

Product Strategy: 3 Technical Improvements

After reviewing the (hypothetical) live platform at `fokuslah.com`, here are three technical improvements I'd prioritize to drive engagement.

1. **Implement a "Friends" System & Social Leaderboards:**
    - **What:** Allow users to add friends, view their friends' progress/XP, and see a leaderboard filtered to just their social circle.

- **Why:** For teenagers, social connection is a primary motivator. Competition and collaboration among friends can dramatically increase time spent on the platform and create powerful network effects.

2. **Offline Mode via Progressive Web App (PWA):**
    - **What:** Convert the application into a PWA, allowing users to download a few lessons and complete them without an internet connection. Progress would sync automatically once they are back online.
    - **Why:** This increases accessibility for users with limited or intermittent internet access (e.g., during a commute). It removes a key friction point and ensures learning can happen anytime, anywhere, boosting daily active use.
3. **Real-time Activity Feed:**
    - **What:** Add a small, real-time feed on the dashboard showing anonymized achievements from other users (e.g., "A student just mastered Division Basics!", "Someone just hit a 10-day streak!").
    - **Why:** This creates a sense of a living, active community. Seeing others succeed provides social proof and motivation for a user to start their own lesson. It makes the app feel more dynamic and less like a solo activity.

Designing Engaging Post-Lesson Progress Reveals

The moment after a lesson is completed is a critical opportunity to deliver a dopamine hit and reinforce the learning habit. My approach would be:

1. **Animated & Dynamic:** Instead of a static summary, use animations. Have XP points physically fly into the user's level progress bar. Make the streak counter burst into flames as it ticks up.
2. **Contextual & Specific Feedback:** Don't just say "5/5 Correct." Say, "Flawless! You've mastered **Simplifying Fractions**." This frames their achievement in terms of a new skill gained.
3. **Unlockables & Badges:** Finishing a lesson, especially with a high score, should unlock tangible rewards. This could be a "Perfect Score" badge, a new avatar frame, or progress towards a larger achievement.

4. **Forward-Looking Teaser:** Immediately guide them to the next step. The reveal screen should end with a clear call-to-action button: "Up Next: Challenge yourself with Mixed Numbers!" This channels their momentum directly into the next lesson.

Handling 1000+ Simultaneous Students

Scaling to handle a thousand concurrent users requires a proactive approach focused on the database, API, and caching.

1. **Database Optimization:**
    - **Indexing:** Ensure database columns that are frequently queried (e.g., `userId`, `problemId`, `lessonId`) have indexes. This is the single most important step for query performance.
    - **Connection Pooling:** Prisma handles this automatically, but we must ensure the connection limit on our PostgreSQL instance is sufficient.
    - **Read Replicas:** For read-heavy operations like leaderboards, we can introduce a read replica database. All write operations go to the primary DB, while read operations are distributed to replicas, reducing load.
2. **Stateless & Scalable API:**
    - Our API, built on Next.js serverless functions, is inherently scalable. Vercel (or a similar provider) will automatically spin up new instances to handle traffic spikes.
    - The key is to ensure our API endpoints are **stateless**—they should not store any session information in memory. Our current setup with JWTs in NextAuth adheres to this principle.
3. **Aggressive Caching Strategy:**
    - **Lesson Content:** Lesson and problem data rarely changes. We can cache this data aggressively at the edge (e.g., using Vercel's Edge Cache or a CDN) for a very low time-to-live (TTL), meaning most users will get this data almost instantly without hitting our database.
    - **User Profiles & Leaderboards:** This data changes more frequently but can still be cached for short periods (e.g., 30-60 seconds). This prevents

every single page load from hammering the database for the same information. We can use a service like Redis for this shared cache.