Technical requirements:

1. Table of content:
    1. Table of content (SELF)
    2. Notational Conventions
    3. Main competitors
    4. Name, moto, description
    5. Page counter
    6. Additional documents {in progress}
    7. General requirements
    8. Pages description (related to backend AND frontend)
        1. General Description
        2. Overview, General and features description, user onboarding, use-cases, and UX writing (including requirements for designer AND for developer)
        3. Modules and components (for developer mostly)
        4. Design/UI/UX checklist
            1. On EVERY page / dynamically shown
    9. Technical development requirements and general code requirements (relevant to backend mostly, but some part for frontend also; if it's relevant to frontend, it have remark)
        1. General Technical Requirements
        2. Software Architecture
            1. Frontend
            2. Backend
            3. Tests
            4. Admin (backend and frontend)
        3. Data Handling, Validation, XSS prevention and Edge Cases
        4. Security. For developer only
    10. Glossary (Ubiquitous Language)
    11. Code implementation
        1. General Code requirements
        2. Proposed structure
        3. Architecture, detailed code requirements, code best practices
2. Notational Conventions
    1. Text between == == (it can become yellow-highligted {in Google Docs it's just yellow without " == == "}): these are the items of the technical

requirements that you need to pay special attention to and probably discuss them with me, because I am not 100% sure how exactly to design / implement these. So, all items between == == / yellow are subjects of discussion and must be determined in the process

2. Text between < description>< /description> tags: totally ignore it. This is improvements and new features for second phase of the project, after it's launching and will have another budget. But some of them is just marks/notes for myself, so anyway - discuss those.
3. There is some elements between HTML and other tags, like < button>< /button>, etc. If you don't understand those tags - it's OK, just implement what in the textual description.
4. Bold text or underlining text between ** ** or < u> < /u>: pay special attention to it, because they probably emphasize something (or sometime bold - is just a header)
5. -> represent options, selection and conditional IF -> then logic (Conditional Rendering / Decision Tree / Dynamic or Smart Forms).
6. 🔲 (empty checkbox) represent separate features / elements to take attention to them and make sure their will be implemented, each (it's just annotations).
7. "EXAMPLE OVERVIEW": this is usually followed by URL on screenshot of the whole page with target UI element to show how this specific element looks on page.
8. "EXAMPLE ZOOMED": this is usually followed by URL on screenshot of the specific UI element, zoomed in.
9. "NOTE FOR MYSELF": ignore these, it's just a notes for myself
10. Something between (backslash tics): usually represents code or some entities or fields.
11. Strikethrough text between , ~~like this~~: in most cases you should ignore it.
12. "Note for future": totally ignore it. This is improvements and new features for second phase of the project, after it's launching and will have another budget. But some of them is just marks/notes for myself, so anyway - discuss those.
13. Notations like {TEXT_AREA_FIELD} or similar usually represents type of fields, similar to HTML types of fields.
14. EXAMPLE 1, EXAMPLE 2, etc.: meaning you must elegantly and accurately combine features from different examples into one.
15. Numbers at the end of description of some items, like "70/100" and so on: these are either to specify priority or affiliation with some category or assignments in term of strength.
16. "TODO" mark at the end of some items: most of that items should be discussed separately in the process. Some of them even maybe will not be implemented in final version.

17. U+200C (Zero Width Non-Joiner (ZWNJ) Unicode Character) - hidden character to avoid markdown formatting issues

3. Page counter:
    1. Registration multi-step process: about 4-5 steps, so there will be about 5 pages
    2. Login
    3. Password reset
    4. **“Practice” This is the main category/pages of the app. including Learn Outline and Milestones, etc. Quizzes also sometimes appear here.**
    5. “Please Rate content” window (modal window)
    6. Quizzes. About 3-5 pages, but with many small variations (about 10 variations). Quizzes can be used in the “Practice” pages
    7. “My progress” 1 page
    8. “PREMIUM” - Prices, Plans and checkout (about 3 pages)
    9. Profile and settings (about 3 pages)
    10. Subscription management (2 pages)
    11. Unsubscription multi-step process (about 5 pages)
    12. Micro-interactions, micro UX, dialogue, user-flow, surveys/feedback (about 5-10 pages)
    13. Some minors modal/dialogue windows (about 5)
    14. Admin (including tooltip editor): about 10 pages
    15. **TOTAL: 35-50 pages** (mobile, mostly compact screens, therefore not that large and feature rich in term of each screen)
4. Main competitors:
    1. Mimo. Best UI and UX combined.
    2. Sololearn. Also good.
    3. AnkiDroid (bad UI, but good UX and killer feature)
5. Name, moto, description [currently in development stage, you can ignore it]
    1. Domain and preliminary name: metaokee.com
    2. Moto: New revolutionized approach to study. Learn new things, improve your skills like never before
    3. ==App name for Google Play and App Store: Learn code/programming with AI | Learn anything ~~with~~ by AI new way /// Learn any sub with AI new way /// Learn with AI new way /// Education with AI new way /// Educate with AI new way /// Educate yourself by AI new way

    4. App description:
        - Learn code, Languages, Math, Science and Business with AI in a new way / applying totally new approach, like never before.

- Scientifically proven techniques to improve learning effectiveness and memorization.
- Forget about boring and ineffective courses, YouTube tutorials, bootcamps and labs

  Metaokee is a first app in the world that ~~truly~~ ~~in real time~~ adapt your learning process based on your ~~personal~~ level of skills, knowledge and understanding in real time *and ~~also~~ make it fun by introducing gamification in the learning process

  Download now and dive deep into ~~the~~ new world of education with power of AI

6. Additional documents:
    1. General requirements (current, main document)
    2. Backend
        1. Backend architecture. Look at ARCHITECTURE_INDEX.md
        2. API endpoints description
        3. In-depth, detailed technical requirements
        4. Deep justification, explanation of chosen solutions and FAQ
    3. Frontend
        1. In-depth, detailed technical requirements - FRONTEND - REACT.md
    4. Edge cases
        1. Edge cases - generic
        2. Edge cases for educational apps (quizzes, etc.)
    5. Detailed requirements
        1. Code requirements (backend)
    6. Security (for developer only).md
    7. Guidelines and checklists
        1. Design and frontend organization guidelines
        2. Tamagui guidelines
7. General requirements
    1. Align to good and avoid bad practice.
8. Pages description (related to backend AND frontend):
    1. Registration
        1. Language selection
        2. What you want to learn? This page must be done similar to Twitter registration process, like here: https://i.insider.com/5d371613100a243cac3e48bc?width=800&format=jpeg&auto=webp and here:

https://static1.agorapulse.com/social-media-lab/wp-content/uploads/sites/6/2021/02/twitter-promote-interests.png (look for more details: https://www.researchgate.net/figure/Explicit-interest-inputted-by-users-during-signing-up-on-Twitter_fig1_343144513)

1. Checkboxes 🔲:
    1. Coding
    2. Math
    3. Languages
    4. …
    5. it must have expandable, clickable divider wrapper with numeric value of how many categories collapsed (for sub-categories numbers also showed separately, but inside of labels); if you click it, it will show more categories. Right after that expandable divider wrapper there must be fixed “Other” option.
    6. Other (?) {Nothing from the list. Specify manually}
        1. When you click “Other” option, {TEXTAREA} field must be shown (demonstrate this state also separately)
        2. {When user only written custom category (via “Other” field) it must show error: “Choose at least 1 standard category also” (?) TOOLTIP TEXT: “Non-standard categories in development stage, so content can appears with delay. To make sure you can start learning immediately, choose at least 1 standard category”}
    7. Look how Twitter registration is done. Pinterest and Tumblr also
3. Choose sub-topics. Similar to Twitter or Pinterest. EXAMPLE: https://i.insider.com/5d371613100a243cac3e48bc?width=800&format=jpeg&auto=webp
4. Rate how strong each topic you want to learn? That page will display ONLY interests that user selected on the previous page and bellow of the each interest - there will be slider. Near each topic there will be “+” sign to expand it to rate CHOOSEN sub-topics also.
5. Rate your level for each area. That page will display ONLY interests that user selected on the beginning, first page and bellow each of the interest - there will be:
    1. SLIDER: Beginner intermediate advanced
        1. difficulty Level is from 1 to 100 and assigned to words by ranges: beginners, intermediate advanced

6. Set your pace page. You want to spend how many time per day on learning?:
    1. SLIDER: 30 m 2-3h 6h+
    2. CUSTOM_META_FIELDS: JSONB
7. Success page with building personalized program animation

2. Menu (is displayed almost on every page):
    1. Learn
    2. Train (infinity icon) - it opens progress for each “related_concepts” (see backend description) - all related sub-categories to what user choose in onboarding/registration process (pages with checkboxes, sliders, etc. - where user select his interests); there must be wide horizontal UI card for each related_concept similar to “Mimo” app
    3. Memorize (brain or book icon). It must should open textual description that this section is in development with ability to join waitlist
    4. PRO
    5. Profile
3. Home page / dashboard
    1. GENERAL LAYOUT
        1. Home page is a vertically scrollable dashboard with stacked sections; order (from top to bottom) MUST be: `Header stats row` → `Daily challenge card` → `Active training block` → `Improve areas strip` → `Today progress` → `Streak / 2X boost` → `Informational article carousel` → `Premium upsell card`.
        2. All sections MUST be rendered from config / backend schema (no hardcoded titles), so new blocks can be injected or reordered in the future without layout rewrite.
        3. On mobile, header and bottom navigation MUST stay visible as long as there is space; when user scrolls down aggressively, header may compact, but bottom navigation MUST stay pinned.
        4. Each section MUST support “empty / first-time user” state (no active course, no XP yet, no streak yet) and “active / returning user” state.
    2. HEADER STATS ROW (TOP BAR)
        1. Left side: small mascot / app icon (tap opens micro “tips & news” modal later, but for now just placeholder action).
        2. Right side: language selector with flag and current language code; tap opens language picker from onboarding (reuse the same component).
        3. Under language selector: compact level progress pill:

        1. Shows current level number, small progress bar to next level, and remaining XP to level up (for example: “Level 12 · 15 XP to next level”).
        2. Tapping pill opens dedicated `Progress` page (or tab) focused on level, XP and achievements.
    4. Under progress pill: three “wallet” chips in one row (spacing like screenshot):
        1. Chip 1 – hard currency balance (e.g. `Tokens`).
        2. Chip 2 – streaks.
        3. Chip 3 – soft currency or bonus points (e.g. `Gems` or `Meta XP`).
        4. Each chip MUST be tappable and open the same `Rewards / Store` surface (even if store is MVP text for now).
3. DAILY CHALLENGE CARD (`Today's challenge` block)
    1. Sticky blue card near the top with title “Today’s challenge”.
    2. Inside the card there MUST be:
        1. Countdown timer until the end of the challenge (HH:MM:SS).
        2. Main CTA button (play icon) that starts challenge training session immediately using the same engine as regular quizzes/lessons, but with challenge preset. When user complete challenge, it must show “Completed for today” label and option to review or redo (configurable).
    3. States:
        1. Idle (challenge not started today) – card shows full timer and “Start” CTA.
        2. In-progress – card shows remaining time and “Continue”.
        3. Completed – show “Completed for today” label and option to review or redo (configurable).
        4. Timer resets at midnight (user’s timezone) and start when user open dashboard (and always show “X2 user’s pace” value). {specific algorithm will be determined in the process of development}
4. ACTIVE TRAINING BLOCK (`Today's training` area)
    1. Left card – `Active related_concept / category / topic` card (e.g. “Introduction to Python”):
        1. Displays related_concept name, short subtitle, circular progress (percentage of related_concept completion), and primary CTA “Continue learning”.

        2. Tapping the card or CTA MUST open training session starting from the last unfinished lesson or quiz of that related_concept.
        3. If no active related_concept – show generic recommendation card (“Start your first training) powered by the same recommendation engine that is used when user selects interests.
    2. Right side stats panel:
        1. Shows XP progress to next level (current XP / needed XP).
        2. Shows current level number in circle.
        3. Shows list of earned badges (with a “+ more” if there are more than N).
        4. Shows remaining lives and tokens with “Get more” secondary CTA that leads to store / rewards page.
5. IMPROVE AREAS STRIP (RECOMMENDATIONS)
    1. Horizontal strip titled “Improve” OR “Improve Areas” (configurable) with ability to collapse and uncollapse (like in the Figma designs). It must be shown only in the dashboard, not in the other pages.
    2. Content: chips / tabs with key domains a user can improve (e.g. `Programming`, `Science`, `Marketing`, etc.), populated from backend user’s interests.
    3. When user change interest, recommendations MUST update (quizzes, lessons) filtered by that areas.
6. IMPROVE AREAS – MOBILE BEHAVIOUR
    1. On mobile devices, these recommendations must hide when we scroll down and appear again when scrolling up.
    2. On mobile devices, there must be very small button to collapse and uncollapse them. When collapsed, - the “uncollapsing button” must stick to the top, and, again, - it must totally hide when scrolling down.
    3. There must be “pre-made” suggestions, based on user interests, TOPs.
7. TODAY PROGRESS SECTION
    1. Section title: “Today progress” with percentage indicator (e.g. “33%”) and small info icon next to title.
    2. Info icon MUST open tooltip or modal explaining how daily progress is calculated (e.g. “Based on completed lessons, quizzes, and time spent today relative to your daily pace goal”).

3. Visual: horizontal progress bar with filled portion matching daily completion percentage.
4. Progress bar styling:
    1. Filled portion – accent color (e.g. blue gradient).
    2. Empty portion – muted / dark gray background.
    3. Bar MUST be smooth, rounded edges.
5. Data source: backend calculates daily progress based on `completed lessons count + quiz answers + active learning minutes` vs. `user's daily pace target` (set during onboarding).
6. Progress resets at midnight (user timezone).
7. If user exceeds 100% (overachiever), bar MUST show 100% but text can say “>100%” or actual value (configurable).

8. STREAK / 2X BOOST SECTION (DAILY STREAK CALENDAR)
    1. Section title: “Practice 5 days in a row and get a 2X boost” (text configurable from backend).
    2. Visual: horizontal row of 5 day cards (Day 1, Day 2, Day 3, Day 4, Day 5) each represented as small card/icon with day label and phone/device icon.
    3. States per day card:
        1. Completed (past days with activity) – icon shows filled / checked phone with color accent.
        2. Current day (today, if user already did activity) – icon glows or has highlight border.
        3. Future days (not yet reached) – icon muted / grayed out.
        4. Missed day (user broke streak) – icon shows broken chain empty battery.
    4. Below the 5-day row, MUST show current streak status:
        1. If user is on streak: “You are doing in a row, X days” (where X is current consecutive days).
        2. If user broke streak: “Start a new streak today!” or similar motivational copy.
    5. Tapping the section MUST open dedicated `Streaks & Rewards` page explaining:
        1. Streak rules (must complete at least 1 lesson or quiz per day).
        2. Rewards for milestones (5 days, 10 days, 30 days, etc.).
        3. History of past streaks and longest streak.
    6. Backend must track:

        1. `streak_start_date`, `current_streak_count`, `longest_streak_count` {more specifically will be determined in the process of development}.
        2. Streak breaks at midnight if user didn't complete minimum daily activity (configurable threshold, e.g. 1 quiz or 10 minutes).
    7. 2X boost reward:
        1. When user completes 5-day streak, unlock "2X XP boost" for next 24 hours (or configurable duration).
        2. Badge or label MUST appear in header showing active boost status.
9. INFORMATIONAL CAROUSEL
    1. Section with carousel of informational cards (swipeable horizontal scroll).
    2. Example card from Figma designs: "The science behind training" with subtitle "*Gamification improve learning outcomes with large effect size*".
    3. Card design:
        1. Background image or gradient (unique per card, loaded from backend).
        2. Title text (large, bold, white or contrasting color).
        3. Subtitle or excerpt (smaller font, secondary color).
        4. "Learn More" CTA button inside card (secondary button style).
    4. Carousel navigation:
        1. Swipe gesture for mobile.
        2. Small dot indicators below cards showing current position (e.g. 7 dots if 7 articles).
        3. Arrow buttons (< >) on desktop or tablet.
    5. Tapping "Learn More" or card itself MUST open:
        1. Full view (modal or dedicated page) with content rendered from backend.
        2. Content can include images, videos, references, "Related topics" links.
    6. Content source:
        1. Content fetched from backend or API endpoint.
        2. Content can be tagged by topic (e.g. "Learning Science", "Gamification", "Memory Techniques") and filtered by user interests (optional personalization).
    7. Empty state: if no content available, hide entire section (don't show empty carousel).
10. PREMIUM UPSELL CARD (GO PREMIUM)

1. Large card near bottom of dashboard with visually distinct styling (e.g. gradient background, premium badge icon).
2. Card content:
    1. Small label “UPGRADE TO PREMIUM” with premium badge icon.
    2. Main heading: “Go Premium” (large, bold).
    3. Subtitle: “Unlock exclusive features and remove ads” (configurable marketing copy).
    4. Two CTA buttons:
        1. Primary CTA: “Choose your plane” [sic - typo in original, but keep as-is for now] – opens subscription plans page.
        2. Secondary CTA: “Learn more” – opens detailed premium features explanation modal or page.
3. Visibility rules:
    1. If user is already premium subscriber, hide this card entirely OR replace with “Upgrade to Higher Plan” card (depending on the plan).
4. Premium features page (opened by CTAs):
    1. List of premium benefits (e.g. “Ad-free”, “Exclusive content”, “AI usage”, etc.).
    2. Subscription plans (monthly, yearly) with pricing from backend.
    3. Payment integration (Stripe / in-app purchases).

11. BOTTOM NAVIGATION BAR (MAIN MENU)
    1. Fixed bottom bar visible on all main app pages (Home, Train, Memorize, Progress, Profile).
    2. Five navigation items (from left to right):
        1. “Learn” tab – icon: book or graduation cap – navigates to dashboard / home (this page).
        2. “Train” tab – icon: infinity symbol or dumbbell – navigates to training / progress page (see section on Menu).
        3. “Memorize” tab – icon: brain or book – navigates to memorization feature (MVP: show “Coming soon” modal).
        4. “Progress” tab – icon: chart or analytics – navigates to user progress overview (XP, levels, achievements, streaks).
        5. “Profile” tab – icon: person silhouette – navigates to user profile & settings.

3. Active tab MUST be visually highlighted (e.g. accent color icon, underline, or filled icon vs. outline for inactive).
4. Each tab MUST show icon + label (text below icon).
5. Tapping active tab (e.g. tapping “Learn” while on dashboard) MUST scroll page to top (standard UX pattern).
6. On desktop / tablet, bottom bar can be replaced with side navigation panel (configurable layout).

4. Todays training. It must start quizzes and training process straight away
    1.
        1. **On this page, user can enroll in courses and mini-games. Also, we provide him courses and mini-games recommendations based on user preferences, chosen topics, tags, categories, etc. “Courses” itself are not separate entities, but more like a vectors and refined user interests, because most of the content will be created by AI (for example: “Python Basics” - it is refined interests based on user interest AND his level/difficulty. But it is still mostly content created by AI)**
            1. When the user enrolls in a course, he starts to be involved in the gamified part of the app, similar to Mimo or Sololearn: Here is examples: 1) https://drive.google.com/uc?id=10C-oc60Fjuc1d-6g7704kvf7itT-cs-0 2) https://drive.google.com/uc?id=13vKgSMFIX6nYX0boK3V_WgnmhW4TH-Ol 3) https://drive.google.com/uc?id=1xRRH96dwA_ecVMLRKObSD7HCkWIBaJnB 4) https://drive.google.com/uc?id=14Q3Se1yGgzrXqDYEQIaU7UbSktUjPXmg 5) https://drive.google.com/uc?id=10_BqHJw8xpMgxt23P1UoikencSF9Bdxf
                1. Each course is divided into modules, and each module is divided into lessons;
                    1. The user, when he takes lessons, get tasks and exercises done and therefore has progress, which, again, is similar to Mimo (see the previous screenshots). When he have more than X hours in related_concept, it will make 100% in ‘progress’ (dedicated ‘progress’ page).
            2. Example of content from one of the Courses showed on the previous screenshots;

3. All content must be “expandable”. So, we first start with a very broad/general/brief/abstract/summary explanation and then start to dive deep into more detail for those who don’t understand the summary.

5. Quizzes
    1. We must have 5 types of quizzes/fields:
        1. Put right options to missed places/field https://drive.google.com/file/d/168axoaehXwfbQ0ad-sTh50-KX08-pZ1m/view?usp=sharing
        2. Choose correct answer/statement from few options (tap the correct answer): https://drive.google.com/file/d/1xGaCPAiV3wkFxFMXbUGuivWa9DQod_jN/view?usp=sharing
            1. When answer is wrong, it must show correct answer
        3. {TEXTAREA} type of answer. When user must enter answer manually.
            1. on lowest level in textarea type of questions we can show partly grey right answer {demonstrate this state in design also separately}
            2. When user type it will show mistakes with red color and right typed symbols shown in green.
6. Lessons. Lessons are short piece of information that generated by AI with a goal to teach user some subjects, but do it adaptively, based on user answers and level of knowledge. Here the example of how lesson may look:
    1. Under the lesson, here we have options / buttons text:
        1. I understand but want to learn more about it (?) Click if you want stick to that topic little bit more
        2. I don’t understand anything
        3. GO TO the NEXT QUESTION/LESSON ANYWAY
    2. Each Lesson AND Quiz must have likes-dislikes. Put on top right.
    3. Report for exercise, similar to Mimo. Here the example: {First step OVERVIEW before report: https://drive.google.com/uc?id=1Vps7JAVfvWK4WS2HsvsBfcExXvEuGmeR; First step, OVERVIEW with annotations: https://drive.google.com/uc?id=1n8QJy5qrmyEMvHU9nBVQ--yUDKyIgsXH; Second step, choose report: https://drive.google.com/uc?id=1FB1J2NzlKvNsjtoaVY6uooggKsKBE2GK; Third step, clarify: https://drive.google.com/uc?id=1P0LvTLGLVMbHactckNyjATRa2q2-dGcW}
7. Mini-polls between lessons and quizzes. There must be:

1. Ability to rate a difficultness
2. Ability to rate quality and satisfaction.
   Here are examples:
   Rate How you feel about the passed section?
   This helps adapt future content.

   “Fully Enlightened”
   100

   “Proficient”
   90

   “Some Small Insight”
   50

   “Unresolved Questions”
   5

   “Totally Unmet Expectations”
   0

   {also, below, there is “enter manually” field}

   -> if positive:
   1. What is the biggest challenge you face when dealing with learning {MAIN_USER_INTEREST}?
   {TEXT_AREA_FIELD}
   2. How are you currently (or before Metaokee) handling learning? What tools or methods do you use?
   {TEXT_AREA_FIELD}
   * This survey strictly deanonymized, so even we don’t see your personal data and can’t connect your answers to your account; we only see your answers and no other information related to you or your account.

   -> if negative:
   what was wrong?:
   Not related
   Too easy
   Too hard
   {also, below, there is “enter manually” field}
   Is any good thing?
   Yes
   No

If yes ->
What was good? (choose up to 2 most important for you)
1. Ease of use
2. It's already effective for me
3. Tracking progress/insights
4. Unique features not found elsewhere
{also, below, there is "enter manually" field}
If no ->
What was bad? (choose up to 2 most problematic things)
1. Too complicated to use
2. It is not effective for me
3. I know better alternative
4. Some features are confusing
5. I don't trust the data/insights
{also, below, there is "enter manually" field}
Is there anything OK, but can be better?
if Yes.
What can be better?
{TEXT_AREA_FIELD}
if No:
We're really sorry you're feeling disappointed. Could you please share a few thoughts of feedback and how we can improve?
{TEXT_AREA_FIELD}
Send No, I'm not leaving any feedback / Skip feedback

-> if positive:
What was good?
1. {Same options as above}
What can be better?:
2. {Same options as above, but excluding that user already choose}

🔲 Also, show how much steps passed and how much left (progress bar) in survey.

Examples:
41e216c5231d418c2354d3e69b997750.png (1600×1200) (dribbble.com)
(we also can add tooltip as in the example above and explain in this tooltip that "your rating will help improve our service")

🔲 We need to ask questions gradually and in a multi-step manner. Here is one example:
1*MQg2PbLWvxd_LGxL1CqSmA.png (1158×920) (medium.com)

Here, in this example, you can see that questions appear only after choosing the amount of stars, and questions are different depending on context and situation.
Also, here is some really useful stuff on this topic:
https://uxdesign.cc/the-ux-of-rating-systems-bc4f9d424b90
Transitions between questions and pages must be in same way as in typeform.com forms: when you choosing option, it is “collapsed” / moved with animation to the next section and on next section I can see compacted version of the previous question and answer. If I click on it, I can return to that question and modify answers.

🔲 When the user gives a rate - it is saved, and then **we offer a slider to the user** so he can rate either content or advice tips from 1 to 100 [slider must looks like this: https://cdn.dribbble.com/userupload/42399094/file/original-819cec768c0d230bef2668289749be44.gif, but smile and central dot must be like this: https://cdn.dribbble.com/userupload/21608493/file/original-500b6eacff5b34ea2b29cdf9217afb55.gif and also we need to add numeric value close to central dot, from 1 to 100 (which will float with dot moving)] . Or we can use a technique similar to the Medium-like technique, where users can rate many times. We need 2 variants for AB-testing: slider and Medium version.
For lessons/separated content parts (not big sections and not the whole course… means for a small portion of content), we have a simpler rating, like this:
–
How good is that part:

“Mind blowing”
“Excellent”
“Good”
“Acceptable”
“Underwhelming”
“Disastrous”

1. **For developer: 🔲 slider shows only every 3 rates 🔲 and no more than 2 times per session; in the future, we can use machine learning to show it optimal times**
   for designer AND developer:
   1. In the admin panel, we must have the configuration for frequency, so we need to show it on designs

2. And the interface for AB-testing configuration and switching between Medium-like rating or slider-like rating.

8. Learning feed / stream and **milestones**. This section does not need to be standalone, we can show the feed/stream elements in previous sections, for example in the “Course Outline”. The bottom line is that we need to show the user some kind of dynamics, both in his progress and in updates in the learning process, in lessons, etc. This feed should be dynamic and the user should be interested in what has changed since his last learning session. In this feed, there should be likes and dislikes of content, comments, etc.
9. Gamification part.
    1. There is timer for each question with wick. Timer calculated as: 100 - streak - Current level. So, higher the level and higher the streak - faster timer. With a shorter timer, more bonuses
    2. We must have points/XP system
    3. Streaks
    4. badges for streaks and … (TODO: for accuracy? XP? Checkpoints?), etc.
10. Tokens
    1. Because AI cost money, we must have some limits on how much user can interact with AI, measured in tokens.
    2. When user perfectly answer questions, he can earn some amount of free tokens
    3. When user have wrong answers, then we have some fines and decrease amount of tokens - so it will probably improve engagement
11. Progress. QUESTION: TODO: do we need separate progress_timeline or it can be put / implemented via progress_tracking?
    1. progress in areas separately:
        1. Today,
        2. weekly,
        3. monthly
        4. Part of the progress must be shown in “today training” and part on separate, dedicated “Progress” page
        5. To easier count it, track progress every day and then, just sum probably progress for week and month
12. “Report bug button”. It should be on each page / on all pages. UPD: INSTEAD, IF USER DISLIKE, IT WILL SHOW REPORT MODAL.
    Report bug / feedback modals (should be similar to Mimo) {EXAMPLE: https://drive.google.com/uc?id=1fPowVpaMnJKlyIbcgVln0tFkjBlSXKBy} but instead, it should be called “Report Lesson” and have new option: “I

want to report something else or request feature →" (UPD: MAYBE PUT UNDER DISLIKE POLL, WILL BE SEPARATE BUTTON "Report about issue")

1. Reporting dialogues (textarea):
   this must be designed from scratch. It should be something between this:
   https://assets-v3.circle.so/z8z2kk8s6bz0t5f44o883byuih3o and this:
   https://support.titan.email/hc/article_attachments/10453622248089/report-bug.png , meaning it must contain at least these elements:
    1. Modal window on Semi-transparent background that can be closed if you tap or click outside of that window, making it non-intrusive (on mobile version this window must be showed on whole screen, similar to Mimo, here EXAMPLE: https://drive.google.com/uc?id=1FQ1dR0msm-wYdvMrUyvXGDkY-zo6OR1C)
    2. "Close" button (in addition to ability to close it by tapping or clicking outside of window, making it super-unintrusive)
    3. Header
    4. Textarea field
        1. Semi-transparent description inside of textarea field.
        2. Limit on amount of symbols, when refreshed while typing
            1. 2000 by default
    5. "I need feedback (?)" checkbox with tooltip.
        1. Text of tooltip:
            1. Check this box and our team will get back to you on your report
        2. If checkbox marked
            1. There will be additional fields to leave contacts
    6. Attach file(s) button
    7. "Submit" button
    8. Semi-transparent, non-intrusive Privacy notice:
        1. Text:
            1. We respect you Privacy. You can check our Privacy Policy.
                1. It must contain a href to http://metaokee.com/privacy
    9. "Success" message after Submitting form:
        1. "Thank you for your feedback! We'll review and, if you selected "I need feedback" and left your contact details, we'll be in touch soon."

2. Window bellow should be opened after clicking “Report bug” button outside of the Exercises or Lessons or when user choose “Report Lesson. This must be designed from scratch. It should be something between this: https://assets-v3.circle.so/z8z2kk8s6bz0t5f44o883byuih3o and this: https://support.titan.email/hc/article_attachments/10453622248089/report-bug.png , meaning it must contain at least these elements:
    1. Modal window on Semi-transparent background that can be closed if you tap or click outside of that window, making it non-intrusive
    2. “Close” button (in addition to ability to close it by tapping or clicking outside of window, making it super-unintrusive)
    3. Header
    4. Textarea field
        1. Semi-transparent description inside of textarea field.
        2. Limit on amount of symbols, when refreshed while typing
            1. 2000 by default
    5. “I need feedback (?)” checkbox with tooltip.
        1. Text of the tooltip:
            1. Check this box and our team will get back to you on your report
        2. If checkbox marked
            1. There will be additional fields to leave contacts
    6. Attach file(s) button
    7. “Submit” button
    8. Semi-transparent, non-intrusive Privacy notice:
        1. Text:
            1. We respect you Privacy. You can check our Privacy Policy.
                1. It must contain a href target=“_blank” to http://metaokee.com/privacy
    9. Visit the help Center {redirect to our Chatwoot instance}
    10. “Request feature” button
        1. Text under button:
            1. Missing a feature? Request now {redirect to our Fider instance}
    11. “Success” message after Submitting form:
        1. “Thank you for your feedback! We’ll review and, if you selected “I need feedback” and left your contact details, we’ll be in touch soon.”

13. Tooltips
14. Page with changing preferences:
    1. Interest: , Strength of interest, Desired difficulty Level:, Current real level, by default 50% strength for each
15. Payment wall / popup after demo trial attempts finished (they should shown as credits): Designs will be provided later in development
    1. (look also at Gradually decreasing credits in each month down bellow, in the abuse prevention section, it's connected/related)
16. Payment wall / popup, periodically: ==get inspiration for logic and overall design from Mimo and Sololearn ==
17. Billing / Payments in general: use one of this https://www.star-history.com/#killbill/killbill&getlago/lago&polarsource/polar&meteroid-oss/meteroid&ProjWildBerry/wildberry&type=date&legend=top-left or RevenueCat (https://www.revenuecat.com/blog/company/introducing-revenuecat-mcp/) or Adapty.io (should be discussed)
    1. Detailed Billing and abuse prevention requirements (backend only):
       1. For OpenAI / ChatGPT API part, There must be limitations similar to Bing…
       1. For example:
       1. 5 requests per day per user on Free plans.
       2. In total, we must limit 200 tokens per prompt and 1000 tokens per day on prompts +10000 tokens on prompts with history of the chat included. Regarding completion: 300 tokens in one completion and 1000 tokens per day on completions
       3. After one month:
       100 tokens in one prompt and 500 tokens per day on prompts. completion: 200 tokens in one completion and 500 tokens per day on completions +7000 tokens on prompts with history of the chat included.
       4. After 2 months
       ==\{NOTE FOR MYSELF: NEED TO TEST MYSELF VIA API in Visual Studio\} ==(just dialogue through all my requirements, like, for example, pass exercise with real problems… and also choose Context carrier and look how much it will costs) PRELIMINARY SYSTEM PROMPT: Act as you are AI coach. You will receive interests from user and you must try to help him by providing recommendations, exercises, etc. with focus on his interests
       5. After 3 months:
       End of demo. The user can switch to the Gemini 2.5 Flash model and then repeat the 3-month cycle and then there will be the **final end of the demo**.
       6. All these values (limits of tokens after 1 month, after 2 months, after 3 months, etc.) must be in configuration, so we can test and

change them later.
7. We must have ability AB-tests where we can choose how many percent's of users use one or another OpenAI model and look at retention (for example, via Google Analytics). So we can compare, how much ChatGPT 5 will have more of retention, than, for example, Gemini 2.5 Flash - which is much cheaper. If we don't have too much retention difference, we can switch to Gemini 2.5 Flash
2. We must log maximum details in the event if limits are reached: total events, Country[by GEO], Language[by choice], Username, email, date of registration, date of the event, chosen tariff Plan, UTM-label[so we can look at even more detailed stats in Google Analytics], etc., etc.. Each user must shows-up in Google Analytics. When we open the user in the log - we must see previous events.
3. We need detailed usage statistics in admin or, at least, pass UTM data to Google Analytics for the following data:
1. How many users
2. How many authors
3. How many tags and preferences
4. How many times each tag and preference was used
5. How many times each tag and preference was used by each author
6. Tokens spending by user and group of users within date range
7. Filtering by date, sources, etc. (Google Analytics already has those functionalities, so a better option would be just to pass parameters / UTM / data to GA).
4. Abuse prevention:
1. API of LLMs are expensive. To prevent bots and abuse usage (there a lot of cases where "free AI plans" used on some tasks on scale via bots and automation, totally unrelated to task and purpose of the app), we must use categorization ML model or third-party API to classify each of the message;
1. If messages considered not related to mental health, health, etc. - meaning they on totally other, unrelated topics - it can be sign of abuse.
1. For developer only: we can use one of that approach https://www.reddit.com/r/LocalLLaMA/comments/1btu68b/whats_the_best_model_for_topic_classification/ or third-party API: https://www.uclassify.com/browse for classification (can be discussed later).
2. If we have more than X message in Y timeframe that unrelated to our topic (mental health, health overall, etc.). it must trigger moderation and temporary restriction on chatbot in account (only on chatbot/LLM usage; user can use other parts of the app normally

with that restriction)
3. When user trying to send messages from restricted account to LLM, he will get this message:
1. “Your account has been restricted due to potential Terms of Service violations, and has been flagged for review. A Community Manager will investigate, and you will receive a determination notification within 48 hours. You do not need to contact us.”
1. Example of the design:
1. https://drive.google.com/uc?id=1t8Im4aEHmAkW-9ktUlh5pOKFbyppuaOT

18. Periodic global survey. Customer Satisfaction Survey:
    1. **How do you like Metaokee’s products and services so far?**
        1. It’s great! I’m really satisfied.
        2. I’m not satisfied.
        3. This is just not what I was looking for, wanted, or expected.
19. Prompt caching: all responses are saved; if user don’t have subscription (free-plan user) and there is already generated lessons and quizzes within his focus_area/interests, then he will get that cached/saved versions with NOTE/NOTIFICATION that “You are using free limited version without precise personalization. If you want full power leverage of AI and adaptive learning personalization, please consider PRO or PREMIUM subscriptions”. If there is no related to user’s interests already generated lessons or quizzes (there should be limits for free per day), then user will get message: unfortunately, there is no free lessons for you today. Come back tomorrow or try to extend your interests here: .. (or consider PRO/PREMIUM subscriptions, which will generate lessons by AI personally for you, in real time in opposite to FREE plan, where lessons and quizzes are just come from archives after would generated for paid users)
20. Admin panel
    1. Tooltip management

9. Technical development requirements and general code requirements (relevant to backend mostly, but some part for frontend also; if it’s relevant to frontend, it have remark)
    1. General Technical Requirements
        1. Programming Languages: Python, JavaScript(React with Tamagui compiler to React Native).
        2. Web Framework: FastAPI for API creation for backend and React for frontend.
        3. Database: PostgreSQL.
        4. APIs/integrations: OpenAI API for generating suggestions; Kong Gateway or https://github.com/membrane/api-gateway as an API

and LLM gateway + auth and BASIC permissions management; PostHog for surveys, feature flags and AB-tests; Chatwoot for support; Mautic API for email notifications and/or Dittofeed/https://openalternative.co/alternatives/customer-io for messages and/or email notifications. OPA for advanced permissions management

5. Other Libraries:
    - psycopg2-binary for PostgreSQL adapter.
    - python-decouple for separating configuration variables.
6. Deployment: Ubuntu VPSLocal for development, and production (optionally you can use Render and Vercel for development) separate solution would be required for production.
7. GitLab for versioning, code storage, and collaboration and CI/CD pipelines
8. Checks and linters (looks bellow) for code readability and early bug-catching.
9. Because app mostly consist of CRUD operations, use fastcrud by benavlabs (1) for CRUD as a standardized entry point. (1)
   It's Async and also have many other features, such as Offset Pagination, Cursor-based Pagination, Auto-generated Endpoints for CRUD out of box.

2. Software Architecture
    1. Frontend (React):
        1. For all User Interaction and UI use React
        2. The mobile application will be built using React Native via cross-platform Tamagui React (this library provide compilation of plain/vanilla React to React Native without changing any code, meaning **there will be one codebase and this code base is pure React**… Tamagui just compile it to React Native)
        3. UI Kit/library: Tamagui
        4. The web application will use a just Shared codebase between mobile and web application and responsive+adaptive design; I don't need high-level adaptation for the web, the main focus is mobile ("mobile first" concept).
        5. Web version:
           React (Tamagui library)
        6. SSR (need for SEO purposes mainly): Next.js (but if you know more elegant solution for this specific problem (SEO) - it would be better of course not use extra dependency and work without Next.js)

7. Fetching, uploading and progress. The application will fetch data from the backend API through various REST API endpoints (FastAPI) using typed clients and data-fetch libraries.
    1. For fetching: TanStack Query with `ky` (fetch wrapper). Tanstack Query takes a query function and runs it for you. You can just keep your existing functions using `ky` and give them to tanstack query
        1. Few reasons to use TanStack Query:
            1. Caching
                1. Auto Caching + Refetching (stale-while-revalidate, Window Refocus, Polling/Realtime)
                2. Multi-layer Cache + Automatic Garbage Collection
                3. With TanStack Query, components simply subscribe to queries. If the data is available in TanStack Query cache, it is delivered immediately. If it is not, it is queried and then cached and delivered. Anytime that data changes, all subscribers to that query key will be notified automatically, which lets you do clever tricks like smart invalidation. If you have a mutation, you can tell react query that that mutation will invalidate one or more query keys, and it will automatically do the refetch with no extra code from you.
            2. Simplifying asynchronous data handling, and enhancing application performance
            3. Retries
            4. Error Handling (automatic error state management with `isError`, `error` object, `onError` callbacks, and Error Boundaries support via `useQueryErrorResetBoundary`)
            5. infinite-loading APIs
            6. Background updates

2. `ky` (fetch wrapper) — **Feature-rich, tiny, friendly API**
    1. Why:
        1. Built on top of `fetch`. Adds retries, timeouts, hooks, JSON convenience, and readable exceptions.
        2. Works in RN
    2. Limitations:
        1. Like `fetch`, progress events are not part of the API — use for JSON; use native lib for file uploads if you need progress.
        2. Not as ubiquitous as axios but often enough for REST.
3. Don't use axios, because it is not compatible with RN

2. file uploads / progress:
    1. `react-native-blob-util` (or `rn-fetch-blob` fork)
        1. Why:
            1. Native module designed for RN file uploads/downloads and progress reporting.
            2. Provides upload/download progress callbacks on both iOS & Android — reliable where fetch/axios progress can be flaky.
            3. Use inside TanStack Query `useMutation` for uploads and to update UI on progress.
        2. Limitations:
            1. Native dependency (requires linking / config), so it's heavier than pure JS.
            2. Use it only for file operations; keep fetch/ky for JSON API calls.

8. Repetitive forms and probably some admin part: TanStack Form (why? The only popular form management that are RN compatible {react-hook-form, for example, less compatible and more tangled with DOM})

9. Data validation and shaping: Zod. **You still often *want* Zod (or some frontend runtime validation), even with FastAPI/Pydantic on the backend.** Zod - functionally “almost 100% safe” for React Native code (it’s pure JS runtime validation). **The main risk is bundler/runtime issues** (Metro/packager import paths, `navigator.userAgent` checks, etc.), which affect your ability to run/build the app but not the correctness of your validation code. Test in a real RN environment and pin a working zod release. GitHub Backend validation is mandatory for security and data correctness, but frontend validation gives you UX, safety, and developer ergonomics that the backend can’t (or shouldn’t) replace.
10. To simplify and accelerate development (and also, for naming convention): openapi-ts by hey-ap. With OpenAPI TypeScript you can generate code based on API schema. Use it with its Zod plugin (or a generator like `openapi-zod-client`). **You don’t *have to* hand-write Zod schemas — but you still need *some* runtime validator.** Generating Zod schemas from OpenAPI gets you the best of both worlds (compile-time types + runtime checks). All workflow described in more details in “In-depth, detailed technical requirements, terms, code examples and implementations, parameters, arguments, deep specifications, models, list of components, modules - FRONTEND - REACT.md” file
11. For state management, only if it’s really necessary: Zedux or Redux with Redux Toolkit (must be discussed)
12. CDN: https://www.litespeedtech.com/solutions/application-servers/nodejs https://twitter.com/mgechev/status/1661871561014665216
13. Captcha: https://dev.to/ryabinin/protect-your-react-native-application-using-cloudflare-turnstile-4l4l
14. Other Integrations:
    1. PostHog for AB-tests and feedback collection
15. General Code requirements for frontend:
    1. **Component / file size limits**
        1. No more than **100 lines** of code in one file or one React component (anything larger must be split into smaller components / hooks / utilities). An absolute maximum of **200 lines** is allowed

only in rare cases and must be justified in a PR description.

2. No more than **6 top-level functions/hooks** inside a single component file (e.g., event handlers, helper functions, custom hooks declared in the same file). If you need more, extract to `utils`, custom hooks, or child components.

3. Prefer many small, focused components (single responsibility). If a component has > 3 responsibilities (UI, data fetching, heavy logic, state orchestration), break it up.

2. **Directory / module structure**

    1. Add nesting / sub-directories when there are **more than 5–8 files** or **more than 8–10 "objects"** (components, hooks, utils, pages) at the same level. Example layout:

        ```
        src/    features/      featureA/
        components/        hooks/        styles/
        index.ts
        ```

    2. Organize by **feature** (feature folder pattern) rather than by type when the app grows — it improves discoverability and makes code-splitting easier.

    3. Use route-based code splitting (lazy + Suspense) or bundler-level code splitting for large feature folders.

3. **Component shape / props / state limits**

    1. Do not exceed **~20 props** on a single component. If a component needs many props, group them into a single object prop or split the component.

    2. Prefer local state for UI-only concerns (open/closed, local form fields) and use global state (Redux/Zustand) for cross-cutting shared data. Keep slices focused: try to keep **state slice shape** concise — if a slice grows beyond

~20 fields, consider splitting it into multiple slices/contexts.

3. For forms, use form state libraries (TanStack Form) and keep form field definitions modular (split large forms into sub-forms/components).

4. **When to split into separate packages/apps / monorepo considerations**

    1. Move large/core domains into their own packages or micro-frontends once a feature contains **~10 or more components/pages** or becomes independently deployable. In a monorepo, create a package like `packages/feature-foo` or `apps/feature-foo`.

    2. Keep a lightweight **`shared`** package/folder for UI primitives, types, hooks, and utilities used across features: e.g. `packages/shared` or `src/shared`.

    3. Prefer stable public interfaces for shared packages (clear exports, documented props/types) and avoid circular dependencies.

5. **Logs / telemetry**

    1. All logging and analytics should go through a **separate logger/telemetry module** (e.g., `src/shared/logger.ts` or `packages/shared/logger`). This module should expose typed functions like `logger.info()`, `logger.error()`, `logger.event()`.

    2. Avoid scattering `console.log` / `console.debug` across components. Instrument via:

        - Middleware (Redux middleware, Zustand middleware) for state-change logs.
        - Higher-order components (HOCs) or hooks for page-view and lifecycle logging.
        - Build-time removal of debug logs (e.g., using

`babel-plugin-transform-remove-console` in production).

3. Keep logging non-blocking and lightweight in UI thread; defer heavy telemetry to background tasks or send batched events.

6. **Checks / tooling**

    1. **ESLint** (base) — with `eslint-plugin-react`, `eslint-plugin-react-hooks`, and rules for accessibility (`eslint-plugin-jsx-a11y`).

    2. **TypeScript** — static types (`tsc --noEmit` or `fork-ts-checker-webpack-plugin`) for type safety. For JS projects, use strict ESLint rules and JSDoc types.

    3. **Prettier** — formatting. Keep editorconfig + Prettier consistent across the team.

    4. **import/order** or `eslint-plugin-simple-import-sort` / `isort`-equivalent — enforce import ordering and grouping.

    5. Optional (highly recommended): `lint-staged` + `husky` to run linters and tests pre-commit; `stylelint` for CSS/SCSS; bundle-size checks; accessibility testing (axe), and unit/RTL tests run in CI.

    6. Add `eslint --fix` and Prettier auto-format in CI/pre-commit to avoid style churn.

16. Please, separate all texts to i18n so we can use translation in the future and/or easier management of all static texts (including tooltips). But overall most of the texts must be stored in backend database, even some static texts (storing texts in frontend can be done only in some rare cases or even better to be avoided at all, because it's making them hard to manage).
    1. Proposed structure:
        1. app-name-or-feature-name/
        │ ├── i18n/ # Translation setup & resource files
        │ │ ├── index.ts # i18next initialization

```
│   │   ├── locales/
│   │   │   ├── en/
│   │   │   │   ├── common.json # e.g. { “welcome”: “Welcome”, … }
│   │   │   │   └── dashboard.json
│   │   │   └── tr/
│   │   │       ├── common.json # e.g. { “welcome”: “Hoş geldiniz”, … }
│   │   │       └── dashboard.json
│   │   └── customBackend.ts # (optional) if you load translations dynamically
│   │
│   ├── locales/ # TypeScript types for translation namespaces
│   │   ├── common.d.ts
│   │   └── dashboard.d.ts
```

2. Backend (FastAPI):
    1. FastAPI is the core of the backend.
    2. The backend fetches data from the PostgreSQL database using SQLAlchemy ORM.
    3. The backend interacts with the OpenAI API to generate content suggestions.
    4. General Code requirements for backend:
        1. Use DRY and SOLID (SRP, OCP, LSP, ISP and DIP) principles.
        2. PEP6 (for Python)
        3. No more than 100 lines of code in one file or class (anything larger must be split into modules to maintain a clear, modular structure). An absolute maximum of 200 lines of code is allowed only in rare cases. Also, no more than 6 methods in one class.
        4. Add nesting/sub-directories when there is more than 5-8 files or more than 8-10 objects in total (folders and files) on one level.
        5. Do not exceed more than 20 fields (about) per model.
        6. Moving large or core domains into their own apps once models exceed ~10 in count (split it in different apps, but same project in FastAPI) (sources: Stack Overflow, mshaeri.com {these sources for Django, but it basically applies to any project that have models}).
            1. For shared models and functionality across different apps, create app named “shared”
        7. Logs.

        1. All logs should be generated from separate logger module
        2. Logs statements should not be in functional parts of code, instead, they should be auto-generated / injected via patch, therefore avoid bloating codebase
    8. Checks:
        1. Ruff as a basic linter
        2. mypy - static types
        3. Black - formatting
        4. isort - import order
        5. FastAPI specific:
            1. flake8-fastapi — checks common FastAPI mistakes
            2. flake8-pydantic and flake8-pydantic-fields — these are *Pydantic-specific* AST checks for model/validator issues and opinionated rules (e.g. field default/`Field()` usage, presence of descriptions, certain validator/model anti-patterns). They encode heuristics that a general-purpose linter may not warn about. If you care about those Pydantic conventions, these plugins catch them. PyPI . Ruff already has Pydantic-aware rules (and things like `RUF012`) but there are known detection edge-cases and false-positive issues (e.g., subclasses or imported-base-model scenarios). Ruff also exposes settings (e.g., `runtime-evaluated-base-classes`) and a `preview` mode to tune behavior for runtime-evaluated classes like Pydantic/SQLAlchemy. If you run Ruff alone you may need to tweak config to avoid false positives, so easier just use flake8-pydantic
3. Tests: by using Pytest, create high level tests, like open real website and make sure some data available, etc. Also, there should be low level unit tests for areas that can have high impact on performance or security, like filtering, some fetching, authentication and other critical parts.

1. For most tests use property-based with hypothesis testing and TESTCONTAINERS approach (look at In-depth technical requirements for more details)
2. Utilize Playwright for End-to-End testing.

4. Admin (backend and frontend)
    1. Directus (or Hasura, but most likely Directus better for that case. Must be discussed in the process) auto-generated CRUD endpoints as main entry point for admin tasks
        1. For admin panel, use Directus, an open-source backend-as-a-service. Directus gives you CRUD operations. Auto-generate Create, Read, Update, and Delete endpoints for your data model
        2. For UI panel itself, use auto-generated interface by shadcn-admin-kit or “The API Platform Admin” (based on react-admin by marmelab) and/or react-admin itself by marmelab.
            1. What they do?
                1. They just auto generate simple admin panels from any REST API / OpenAPI specifications (FastAPI/Swagger in our case)
                2. and then, you can fine-grain and fine-tune them via ready-made components
                    1. And if this is still not enough, you probably need a Backend-for-Frontend. But Directus can help you with that, too: if offers a GraphQL API on top of your Postgres database.
    2. Form generator: Use FastAPI’s Pydantic models or `Form()` inputs to validate, then generate forms on the frontend from the OpenAPI / JSON Schema (e.g., react-native-jsonschema-form, react-hook-form + JSON Schema). This is the most flexible and modern approach if you separate backend API and frontend form builder. FastAPI exposes the schemas automatically. {this can be postpone for second phase, so it must be discussed}
    3. Database and database management (PostgreSQL and SQLAlchemy ORM and/or vector):
        1. Stores all data, including users, user categories, user tags, and user progress.
        2. Configuration management (API keys, etc.):

        1. Python conftest, Python config files, pydantic-settings and TOML
    3. OpenAI:
        1. Used to generate suggestions based on user categories and tags.
        2. Interaction accomplished via FastAPI and OpenAI APIs.
    4. Business logic, general logic, and functional part
        1. Daily suggestions and progress tracking (FastAPI, PostgreSQL)
            1. …
        2. Payment gateway API: Stripe / in-app purchases (Google Play marketplace, Apple Pay
        3. Email notifications: ~~getresponse.com~~ ==Mumara/Mautic/Mailwizz with PlexMail, emailwiz and Amazon Simple Email Service (SES) ==
        4. Push notifications API: Chatwoot?
        5. Messengers notifications: …
        6. Authorization: Google auth API, …
4. Other Integrations:
    1. Kong
    2. PostHog
    3. jd/tenacity: Retrying library for Python - retry / retrier / retryier for Python
        1. https://github.com/hueristiq/hq-go-retrier
        2. Jitter
        3. Backoff
    4. *Retrier* is a wasm-compatible utility for retrying standard library Futures with a Result output type
    5. FastStream with RabbitMQ for long-lasting tasks and events
5. Only for developer: third-party software (they mostly ready solutions, but must be installed, configured and probably integrated)
    1. Forum community and commenting
        1. Use Discourse open source with https://meta.discourse.org/t/dracula-a-dark-theme-for-discourse/146350 or LemmyNet

2. For “Invite your friend” logic you can use RefRef or similar solution
3. Affiliate program (affiliate partnership program)
    1. Use “Post Affiliate Pro” ==or post affiliate pro open source alternative that as feature rich (only need feature rich). We can consider Weferral, Refferq, Numok ==

3. Data Handling, Validation, XSS prevention and Edge Cases.
    1. Key texts and fields, if they shortened or truncated or partly hidden, should have a tooltip (or “title” attribute) that display the full text (for example, full name on profile/setting page) on hover or focus for accessibility.
    2. We must have client-side validation ensures only valid characters are accepted, depending on context. In some cases it can be letters, spaces, hyphens, apostrophes, in other - only numbers.
    3. Sanitize Unsupported Characters input to prevent XSS (for example, strip non-numeric).
    4. A regex pattern (e.g., /[1]+(?:+)*$/) may be applied to safeguard data integrity.
    5. Same relevant for frontend as well as backend.
    6. Network Latency: Show skeleton loader with — placeholder while fetching.
4. Security. For developer only:
    1. Always keep core framework and application’s dependencies up-to-date to keep up with security vulnerabilities.
    2.
        - Ensure that the application is never in `DEBUG` mode in a production environment. Never run `DEBUG = True` in production.
    3. Cloudflare captcha
    4. Configure environment variables
    5. Secrets management with Teleport by gravitational
    6. For backend only:
        1. Use `slowapi` ((Starlette-compatible rate limiter)) or `fastapi-limiter` to prevent brute-force attacks.
        2. Use `fastapi-users` for user authentication operations such as login, logout, password change, etc.
        3. Use the `@login_required` decorator to ensure that only authenticated users can access a view. The sample code below illustrates usage of `@login_required`.

[1] A-Za-zÀ-ÖØ-öø-ÿ’’-

4. Use password validators for enforcing password policies. Validate in your Pydantic `UserCreate` model with custom validators.
5. Use `passlib` to hash passwords (bcrypt or argon2).
6. Check a plaintext password against a hashed password. Use `passlib` verify method. **Example:** return pwd_context.verify(plain_password, hashed_password)
7. Key Management
8. More on security guidelines and checklists - check “Security (for developer only).md” document.

7. For frontend only:
    1. **The frontend interacts with content-generation services (e.g., OpenAI) only through secure backend endpoints.**
        1. Never call secret APIs (OpenAI keys etc.) directly from the browser. Always proxy via the backend to enforce auth, rate limits and input/output validation. ## Glossary (Ubiquitous Language) ^d5fe93

Context carryover {this is primarily intended for the developer, but it should also help the designer understand the overall picture}

Context carryover in our case means maintaining the history of user interactions, preferences, and progress across different sessions to provide more personalized and relevant recommendations.

For our app, context carryover would involve:

1. **Storing User Selections & History** – Tracking categories, tags, and previously watched content.
2. **Tracking Progress** – Logging daily updates on what the user has watched and their feedback.
3. **Using a Vector Database & PostgresSQL** –
    - PostgreSQL can store structured user data (e.g., watch history, ratings, timestamps).
    - A vector database (like pgvector {Open-source vector similarity search for Postgres), Pinecone, Weaviate, or FAISS}) can store and retrieve embeddings for similarity-based recommendations.
4. **Personalized Suggestions** – Instead of recommending random content, the system refines its suggestions based on past engagement, patterns, and evolving preferences.
5. **Session Persistence** – Even if a user logs out or comes back after days/weeks, their previous context is remembered, ensuring continuity.

For implementation, OpenAI's API (like GPT-5) can handle reasoning and recommendation generation, while embeddings (from OpenAI or other ML models) can improve similarity-based suggestions.

User ↔ (React) ↔ FastAPI API ↔ PostgreSQL
↔ Vector DB (Embeddings)
↔ OpenAI API (GPT & Embeddings)
↔ Celery/FastStream (Tasks)

Code implementation

General Code requirements:

Backend

Testing-stage deploy: autodeploys on Render from GitLab (or you can deploy straight to VPS, but Render is quick and easy)
Production deploy: on our VPS via GitLab runner, GitLab CI/CD Components (it's similar to GitHub Workflow actions) and Docker

fastapi pagination litims your queries. create your own generic classes for paginated responses. do all the queries yourself using repository pattern. return items and total count from repo. return filled pagination schema from controller.

Frontend

Testing-stage deploy: autodeploys on Vercel from GitLab (or you can deploy straight to VPS, but Vercel is quick and easy)
Production deploy: on our VPS via GitLab runner, GitLab CI/CD Components (it's similar to GitHub Workflow actions) and Docker

Common/Shared/Both/Universal code requirements:

1. No more than 100 lines of code in one file
2. Use Google style guide for JSON data (camel case, which is used more often than other variants). Source: https://www.youtube.com/watch?v=dfFgaTdS3rE 09:30
3. All new code or modifications must be done via commits
4. Atomic commits. An atomic commit is a self-contained unit of work that addresses a single, logical change.(1) This principle is more important than the number of files or lines of code involved.(2). Think of each commit as telling a concise story about a specific modification to the codebase.(1) Whether you are fixing a single bug, adding a new feature, or refactoring a piece of code, all the changes related to that one task should be included in a single commit.

5. Hard limits on commits: Commits must be frequent: no more than 100-200 lines of code in one commit and no more than 5 files in one commit (except for rare cases)
6. All commits must go to our GitLab repo (access will be provided upon request)
7. Merge requests can be less frequent
8. A good rule for MR is to reach a certain milestone. For example - basic project setup and structure is already a good MR milestone. <200–400 lines of changes is ideal for MR; more will make it difficult for everyone to review (but styles and tests can be excluded from count). In the frontend there's a lot of stuff that can't really be called code, like styles; also tests aren't particularly interesting for review. Here's a good study on this topic: https://smartbear.com/learn/code-review/best-practices-for-peer-code-review
9. For commit message use **Conventional Commits / Angular commit message** style — i.e. the `type(scope): subject` format.
    1. Quick breakdown
        1. `type` — what kind of change (e.g. `feat`/`feature`, `fix`, `chore`, `docs`, `refactor`, `perf`, `test`, `build`, `ci`, `revert`).
        2. `(scope)` — optional, what area of the code this touches (e.g. `login`).
        3. `subject` — short summary (imperative, ≤50 chars recommended).
        4. optional **body** — long description.
        5. optional **footer** — metadata (e.g. `BREAKING CHANGE:` or issue references).
    2. Examples:
        1. feat(login): Header Description
        2. feat(login): add OAuth redirect handling

           Support OAuth redirect flow for Google and GitHub providers.
           BREAKING CHANGE: login tokens are now stored in secure cookie.
           Closes #42
    3. Why: this format is great for automated changelogs and semantic versioning (tools like `semantic-release`, `commitlint`, and `cz-cli` support it).
10. Follow semver 2 versioning

Files and folder structure (frontend)

use **Atomic (shared)** + **Feature-first (app)** structure

frontend/src/
├─ app/
│ ├─ container.tsx # app-level wiring & providers (TamaguiProvider,

```
QueryClientProvider)
│  └─ routes.tsx
├─ shared/
│  ├─ design-system/
│  │  ├─ tamagui.config.ts # createTamagui config (tokens, themes, shorthands, fonts)
│  │  ├─ atoms/ # Tamagui-based UI atoms (Button, Text, Avatar, Icon)
│  │  │  ├─ Button/
│  │  │  │  ├─ Button.tsx
│  │  │  │  ├─ Button.stories.tsx
│  │  │  │  └─ Button.test.tsx
│  │  │  └─ Avatar/
│  │  ├─ molecules/ # generic Tamagui molecules used across app
│  │  ├─ tokens/ # design tokens (colors, sizes, space)
│  │  │  └─ tokens.ts
│  │  └─ utils/ # small helpers (formatting, icon wrappers)
│  └─ infra/
│  ├─ http.ts
│  └─ analytics.ts
├─ features/
│  ├─ progress/ # Just example for “progress feature. Can be anything”
│  │  ├─ public.ts
│  │  ├─ README.md
│  │  ├─ view/
│  │  │  ├─ pages/
│  │  │  │  └─ ProgressPage.tsx
│  │  │  ├─ hooks/
│  │  │  │  └─ useProgress.ts
│  │  │  └─ components/
│  │  │  ├─ ProgressList.tsx
│  │  │  └─ ProgressFilters.tsx
│  │  ├─ ui/
│  │  │  ├─ ProgressCard.tsx
│  │  │  └─ ProgressHeader.tsx
│  │  ├─ core/
│  │  │  ├─ transforms.ts
│  │  │  └─ validators.ts
│  │  ├─ state/
│  │  │  └─ progressMachine.ts
│  │  ├─ services/
│  │  │  ├─ progress.api.ts
│  │  │  └─ progress.storage.ts
│  │  ├─ types.ts
│  │  └─ tests/
│  │  ├─ core.test.ts
```

```
│   │   ├── services.test.ts
│   │   └── components.test.tsx
│   └── auth/
└── routes/
tamagui.config.ts # optional root-level export/alias
```

Files and folders structure (backend)

General files and folders structure:

```
backend/
├── src/
│   ├── main.py # FastAPI app entry point
│   ├── config.py # Application settings
│   │
│   ├── 🔗 shared/ # Cross-app shared code
│   │   ├── business_sanitization/ # Business-level input sanitization. 🔧 SPLIT from validators
│   │   │   └── input_sanitizers.py
│   │   ├── business_validation/ # Business-level input validation
│   │   │   └── input_validators.py
│   │   ├── api_contracts/ # Reusable API contracts (Pydantic fields/schemas)
│   │   │   ├── common_fields.py # Contains building blocks for DTOs, not complete DTOs themselves (complete is in presentation/dto_api_contracts/*.py)
│   │   │   └── common_responses_dto.py # DTOs for API responses
│   │   ├── ⚠️ exceptions/
│   │   │   └── base_exceptions.py # GridFlowException (business)
│   │   └── primitives/ # Primitive type operations
│   │       ├── datetime/ # Date/time operations
│   │       │   └── datetime_operations.py
│   │       └── text/ # String operations
│   │           └── string_operations.py
│   │
│   ├── ├── infrastructure/ # 🌍 Global infrastructure
│   │   ├── 💾 persistence/ # 🔧 SPLIT from database.py
│   │   │   ├── base.py # Base, metadata
│   │   │   ├── engine.py # Engine creation
│   │   │   ├── session.py # Session factory
│   │   │   ├── initializer.py # DB init, migrations
│   │   │   ├── infrastructure_sanitization/ # Infrastructure-level sanitization
│   │   │   │   └── postgres_sanitizers.py
│   │   │   └── infrastructure_validation/ # Infrastructure-level validation
│   │   │       └── postgres_validators.py
│   │   │
│   │   ├── platform_services_infrastructure/
```

```
│   │   │   ├── logging/
│   │   │   ├── monitoring/
│   │   │   └── ⚠️ exceptions/
│   │   │   └── base_exceptions.py # InfrastructureException (low-level)
│   │   │
│   │   └── 📡 messaging/ # EVENTS: Platform message bus
│   │   ├── interfaces/
│   │   │   └── message_bus.py # IMessageBus interface
│   │   └── faststream/
│   │   ├── broker.py # RabbitMQ broker setup
│   │   ├── consumer.py # Message consumers
│   │   └── publisher.py # Message publishers
│   │
│   ├── 🖥️ presentation/ # 🔧 MOVED from /api/v1/
│   │   ├── router.py # Main router registry (thin)
│   │   ├── 🔗 dependencies/ # Global FastAPI dependencies
│   │   │   ├── database.py
│   │   │   ├── auth.py # Authentication dependencies
│   │   │   ├── unit_of_work.py # UoW dependency
│   │   │   ├── mediator.py # Mediator dependency
│   │   │   └── external_services.py # Third-party service dependencies
│   │   └── middleware/
│   │   └── error_handler.py # Global error handling
│   │
│   └── 📦 apps/ # BOUNDED CONTEXTS
│   │
│   ├── token_generator/ # APP: Token Generator
│   │   ├── domain/
│   │   ├── application/
│   │   ├── presentation/
│   │   ├── infrastructure/
│   │   └── shared_across_sub_apps/ # Shared code across all sub-apps
│   │
│   ├── file_management/ # Same structure as token_generator
│   │
│   └── flow_engine/ # Same structure as token_generator, but with sub-apps
│   └── sub_apps/forms/ # SUB-APP: Forms. Same structure as main app, but without
shared folder
│
└── tests/
├── unit/
├── integration/
└── e2e/
```

Files and folders structure, EXAMPLE FOR REAL APP: flow_engine (Complex App WITH Sub-Apps)

```
apps/flow_engine/
├── shared/ # Shared between flow_engine sub-apps
│   ├── domain/
│   │   ├── models/ # Pydantic domain models
│   │   │   ├── __init__.py
│   │   │   ├── state.py # StateDomain
│   │   │   └── session.py # SessionDomain
│   │   ├── services/ # Domain services
│   │   │   ├── __init__.py
│   │   │   └── state_validator.py
│   │   └── repositories/interfaces/ # Abstract interface that different from implementation, therefore implementation can be easily swap or modified
│   │       ├── state_repository.py
│   │       └── session_repository.py
│   ├── exceptions/
│   │   └── state_errors.py
│   ├── schemas/ # API DTOs
│   │   ├── states.py
│   │   └── sessions.py
│   └── infrastructure/
│       ├── adapters/persistence/orm_models/
│       │   ├── states.py
│       │   └── sessions.py
│       └── repositories/ # Implementation itself
│           ├── state_repository.py
│           └── session_repository.py
│
├── forms/ # SUB-APP: Forms
│   ├── domain/
│   │   ├── models/ # Pydantic domain models
│   │   │   ├── __init__.py
│   │   │   └── image.py # ImageDomain
│   │   ├── services/ # Domain services
│   │   │   ├── __init__.py
│   │   │   ├── form_validator.py # Complex validation rules
│   │   │   └── completion_calculator.py # Completion stats logic
│   │   ├── exceptions/
│   │   │   └── form_errors.py
│   │   ├── repositories/interfaces/
│   │   │   └── image_repository.py
│   │   └── schemas/
```

```
│   │   ├── __init__.py # Re-exports from flow_engine/shared
│   │   └── images.py
│   │
│   ├── use_cases/ # Application services
│   │   ├── __init__.py
│   │   ├── save_form_state.py # SaveFormStateUseCase
│   │   ├── navigate_forms.py # NavigateFormsUseCase
│   │   ├── complete_forms.py # CompleteFormsUseCase
│   │   └── helpers/ # Use case helpers
│   │   ├── persistence_helper.py # Data transformation
│   │   └── navigation_helper.py # Navigation logic
│   │
│   ├── presentation/
│   │   ├── __init__.py
│   │   ├── dependencies/
│   │   │   ├── auth.py
│   │   │   └── unit_of_work.py
│   │   └── routers/
│   │   └── forms.py # backend.py call it/aggregate it
│   │
│   └── infrastructure/
│   ├── adapters/persistence/orm_models/
│   │   └── images.py
│   └── repositories/
│   └── image_repository.py
│
└── [other_sub_apps]/ # Future sub-apps follow same pattern
```

Why app-subapp structure? You can compare this to Django apps, where a single project can have multiple apps. For example, in Django, all apps are launched through a single file. These are NOT domains, since different apps can have both a domain layer and a service layer, their own repositories and schema. Why this structure? To have one shared infrastructure and less boilerplate.

Why CQRS
Easier to extend (most important)
Code easier to extend - because all operations are loosely coupled and it is easier to make changes without breaking anything.

Easier Testing
Test commands: “Does it change state correctly?”
Test queries: “Does it return correct data?”
Separate concerns = simpler tests

For more details and for specific app structure example with complex hierarchy and sub apps - look at “In-depth, detailed technical requirements” - “Detailed files and folders structure” section

Code implementation details and best practices

EXCEPTION AND ERROR HANLING. Look ‘## 7 Per-Route Error Handlers vs Existing Structure’ in “In-depth, detailed technical requirements”

Auth Dependency - Authentication vs Authorization. Look “In-depth, detailed technical requirements”

Permissions and roles. Permissions / access control must be implemented either with Casbin or OPA (subject of discussion). Permissions table will be provided later. Look “In-depth, detailed technical requirements”

Models/domain entities

Use Pydantic BaseModel for entities and put them to ‘domain_models_entities’ explicitly to avoid confusion (because we also have orm_models separately and also validations of types schemas for APIs, which is also use Pydantic)

- **Move *pure* calculations** (e.g., `duration_seconds`, `is_active`) out of the ORM.
- ** Pydantic `BaseModel` can host the same pure methods** (computed properties, business rules) just fine.
- Best practice: keep **DB-specific behaviour** (lazy loads, queries, session access, relationships that need loads) in the ORM/adapters; move **pure state + behaviour** to domain models (Pydantic ok for some lite coupling).

Why split

1. **Testability** — pure methods are easy to unit test without a DB.
2. **Reusability** — workers or CLI can use domain logic without importing SQLAlchemy.
3. **Clear dependencies** — infra (ORM) depends on domain types via mapping, not the other way.

Keep in ORM: anything that needs the DB session, relationships that may trigger lazy loads, query helpers.
Move to domain: any method that computes values from already-loaded fields (duration, active flag, derived totals, validation that uses only fields).

- Anything that **requires DB work** (counts, aggregates, joins, queries, lazy loads, expensive scans) belongs in the **repository** (or a repository helper method) — not on the domain model.

- `services` (aka application services / use-cases) orchestrate multiple repos, call domain methods, and implement higher-level business process logic. Use them for orchestration, not for basic field calculations. Services should NOT have direct DB operations, only through repos.

How to decide: simple checklist

1. **Does the method use only the object's own fields (no DB access, no lazy relationships)?** → put on **domain model**.
2. **Does the method need queries / aggregates / counts / access to related rows that might not be loaded?** → put on **repository**.
3. **Does the logic coordinate multiple repos, side effects, or transactional flow?** → put in **service / use-case**.

Examples:

- `is_active()` — domain (pure: `ended_at is None`).
- `duration_seconds()` — domain (pure: `ended_at-started_at`).
- `get_completed_count(token)` — repository (DB aggregate).
- `calculate_completion_stats(token)` — repository or service depending: if it's a single DB query/aggregate, repository; if it combines several repo calls and applies business rules, service. Services should NOT have direct DB operations, only through repos.

Why not put tiny pure calculations in repository?

Repositories are about **how** to fetch/store data. If you put pure business logic into repositories you:

- hide business rules behind persistence layer,
- make testing harder (need DB),
- break separation of concerns.

Tiny pure methods on the domain model are cheap, testable, and keep the domain expressive.

—
Repo pattern (applies to `IStateRepository`, for example)

- Keep interfaces (domain contracts) under `domain/repositories/interfaces/*.py` This allows us have high-level DESCRIPTION without functionality and logic (pure business domain)
- Concrete implementations live under `infrastructure/repositories/*.py`.
- Interfaces should expose domain types (Pydantic models in your codebase) or DTOs returned by mapping.

Because you use Pydantic for domain entities, you **still** do mapping between ORM rows and Pydantic models in the infrastructure layer before returning domain objects — keep that explicit.

Why this pattern? It allows us to separate high-level domain description in `infrastructure/repositories/*.py`without logic implementation, therefore we can migrate in future to another implementations, if needed (or just refactor implementation / improve - without touching domain/“interface”. In this way we have kinda API that separated from internal code).

and so on

1. Routes

Keep a **central gateway/router as the canonical API surface**, but **colocate router modules with their domain code** and register them into the gateway via a tiny registration/registry API (or a simple `routers()` export). That keeps the public interface central and human-discoverable while avoiding monolithic `api_router.py` files. For more details and few routers pattern look at

CRUD

Because this project is around 75% just simple CRUD (because all heavy service-logic like quiz generation, difficulty calculations, etc. - handled by another developers), use this approach for CRUD: {to not clutter main document, these are moved to “In-depth, detailed technical requirements”}

Code-Related Good Practices:

DO NOT use direct instantiation of infrastructure:**

```
python    # BAD - Service knows about concrete implementation
self.user_repo: IUserRepository = UserRepository(db_session)
```

Should use **Dependency Injection**:

```
python    # GOOD    def __init__(self, user_repo: IUserRepository):
self.user_repo = invite_repo
```

Bad Practices (AVOID):

General bad practice: Single responsibility or DRY violation, Tight coupling, top-down principle violation (router calling service and service only calling domain and domain only calling repo, and no one call DB except orm: UI → Router → Service → Domain → | INFRA | Repo → ORM → DB), issues with repositories and others

Specific bad practices:

INCONSISTENT DEPENDENCY INJECTION

validate.py (Line 18-19):

```
self.invite_repo: IInviteRepository = InviteRepository(db_session)
self.image_repo: IImageRepository = ImageRepository(db_session)
```

- Service depends on **2 different domain contexts** (`token_generator` + `flow_engine`)
- Creates tight coupling
- Violates Single Responsibility Principle
- Should inject repositories via constructor