# **Konnaxion Platform Technical Specification v14**

This document specifies the **Konnaxion Platform** architecture and components, updated from v12 to align with the *Mind Map Konnaxion v2* functional structure. The platform comprises five primary modules — **Kollective intelligence**, **ethiKos**, **keenKonnect**, **KonnectED**, **Kreative** — plus a common core. Each module is detailed across four technical layers (Frontend, Backend, Database, DevOps). All original branded nomenclature (e.g. *CertifiKation*, *Kreative*, *Krowd*, *Konsensus*) is retained. Corrections from the v12 specification are integrated where applicable (excluding aspects already resolved by adopting the Next.js + Django Cookiecutter boilerplate, Ant Design, Tailwind CSS stack). Ambiguous or previously TBD areas are now finalized (given current defaults and requirements). The following sections detail each layer for the core and each module.

## **Common/Core Platform**

### **Frontend (Common)**

* **Next.js Enterprise UI** – The platform uses a Next.js 13+ frontend (TypeScript) as a unified interface for all modules. A common layout and navigation system provides seamless access to Kollective intelligence, ethiKos, keenKonnect, KonnectED, and Kreative sections. Shared components (headers, menus, notifications) are implemented once and reused across modules.
* **Ant Design + Tailwind UI** – The design system combines Ant Design (for rich UI components) with Tailwind CSS utility classes for custom styling. This ensures a consistent look and feel aligned with Konnaxion’s branding. Ant Design themes are customized (via Tailwind config) to reflect Konnaxion’s visual identity.
* **Internationalization & Accessibility** – The core frontend supports multiple languages and scripts, enabling global use. All common text is localizable, and the UI is optimized for right-to-left where needed. The interface follows WCAG guidelines to remain accessible (supporting screen readers, high contrast, etc.), reflecting the platform’s emphasis on inclusivity.
* **Universal Navigation & Search** – A global search bar and navigation system (“Krowd Navigator”) allow users to discover content across modules. For example, a search query can retrieve KonnectED knowledge articles, keenKonnect projects, or Ethikos debates in one go. This is backed by a unified indexing service (PostgreSQL full-text) for quick retrieval. Navigation menus clearly delineate modules but maintain a unified user session and profile across all.
* **Responsive & Low-Bandwidth Support** – The frontend is fully responsive (mobile-first design) and optimized for low-bandwidth scenarios. Media and scripts are lazily loaded; an alternate lightweight mode (text-oriented UI) is available for users on slow connections or older devices. A Progressive Web App (PWA) capability enables caching of key resources for offline access (especially for KonnectED content in remote areas).

### **Backend (Common)**

* **Cookiecutter Django Framework** – The backend is built on Python Django 4.x (using the Cookiecutter Django template for a production-ready project structure). This provides a modular monolith with each Konnaxion module implemented as a Django app within a single project. Shared functionality (the *Core* app) includes user management, authentication, and cross-cutting services.
* **REST API with DRF** – A Django REST Framework (DRF) API serves the Next.js frontend. Each module’s data and logic are exposed via RESTful endpoints (JSON), protected by token-based authentication (JWT). The common layer handles user auth (login, registration, password, OAuth if needed) and provides core APIs (e.g. user profile, search index, notifications) consumed across modules.
* **User Profile & Roles (Krowd)** – The platform has a unified **Krowd** user system. Each user has one profile used across all modules. Profiles store personal info and preferences, as well as *expertise fields, reputation scores, and verification status* as determined by the Kollective intelligence mechanisms. A role-based access control (superadmins, moderators, standard users) is implemented using Django’s groups/permissions for common and module-specific actions.
* **Ekoh Services & Konsensus Engine** – At the core, the **Ekoh** service computes dynamic reputation weights for users based on their contributions and ethical behavior. It categorizes expertise by domain (science, engineering, etc.) and assigns merit-based scores to ensure decisions reflect collective knowledge. The **Konsensus** engine, built atop Ekoh and *Smart Vote*, aggregates weighted votes and opinions across the platform to produce balanced outcomes. These services are implemented as Django services or management commands that update user weights and compute results for votes (used by Ethikos, etc.) after each relevant event. The logic emphasizes transparency and fairness, replacing charisma with competence by weighting input from the most qualified users.
* **Integration & External Services** – Common backend handles integration with third-party services: e.g. an email service for notifications, an object storage (S3 or MinIO) for user-uploaded content, and optional APIs for things like translation (used by keenKonnect’s real-time translation feature) or mapping (if needed for projects). It also includes a task scheduler (Celery + Redis) for background jobs such as sending emails, recalculating Ekoh scores periodically, bulk indexing for search, and generating offline content packages.

### **Database (Common)**

* **Primary Relational DB (PostgreSQL)** – A PostgreSQL database (with PostGIS extension if needed for geospatial data in projects) stores all persistent data. Common entities include:  
  + **User** – Master user table (extending Django’s user model) with fields for profile info (name, bio, etc.), **expertise areas**, **Ekoh reputation scores** per category, and flags for ethical standing or account verification. For example, a user may have Ekoh scores in *Environment* or *Medicine* domains which influence their weighted voting power. Ethical multipliers (increasing or decreasing influence based on behavior) are stored per user.
  + **Certificate (CertifiKation)** – Common data model for issued certificates (the *CertifiKation* system) used mainly by KonnectED. Records link a user to a skill or course certification, with issue date and metadata (e.g. level, mentor who certified, etc.).
  + **Expertise & Categories** – A reference table for fields of knowledge (e.g. categories like Humanities, Natural Sciences, Technology, etc.) to classify both users and content. Users can be linked to multiple expertise categories with their Ekoh score in each. These categories are also used to tag content (KonnectED articles, keenKonnect projects, Ethikos topics) for relevance matching.
  + **Global Content Index** – An indexed view or table to support the unified search. This aggregates key fields from knowledge articles, project descriptions, debate topics, etc., enabling efficient global searches.
  + **Audit & Logs** – Common tables record activity logs, notifications, and moderation flags. For instance, a content moderation queue (for flagging inappropriate content) is shared across modules, and all user actions (posts, edits, votes) can be logged for transparency and analysis.
* **Full-Text Search Index** – In addition to relational tables, a full-text search index is maintained (either via PostgreSQL’s tsvector columns or an external search engine). This index covers titles, descriptions, and tags of content across modules, supporting the global search feature. It updates whenever content is created or edited.
* **Scalability & Sharding** – The database design keeps module-specific tables separate (namespaced by app) but on the same primary DB for simplicity. However, if usage scales, certain high-traffic tables (e.g. large content or logs) can be partitioned or moved to separate databases. The common user and Ekoh tables remain central to maintain consistency of the Krowd data.

### **DevOps (Common)**

* **Containerized Deployment** – The platform is delivered via Docker containers orchestrated with Docker Compose (for development) and Kubernetes or Docker Swarm in production. By default, the Cookiecutter Django stack provides Docker images for the Django API, a Postgres DB, and services like Celery and Redis. The Next.js frontend runs in a Node.js container (with Next’s build output optimized for production). A reverse proxy (Nginx or Caddy) routes requests: API calls to Django, and frontend requests to the Next.js server or static assets.
* **CI/CD Pipeline** – Continuous integration and deployment is set up via GitHub Actions. On each merge, tests are executed (both backend Django tests and frontend unit/integration tests). Upon success, a CI pipeline builds Docker images and can deploy to staging/production. Infrastructure-as-Code (like Terraform or Docker configs) manages staging and prod parity. Deployments are versioned (supporting rollback if needed).
* **Monitoring & Logging** – Common monitoring includes health checks for each service, centralized logging (with ELK stack or a cloud monitor). Error tracking (e.g. Sentry) is integrated into both Django and Next.js to catch exceptions. Performance metrics (APM) help identify slow queries or pages.
* **Security & Compliance** – All services enforce HTTPS and secure credentials (secrets managed via environment variables or vault). Regular backups of the Postgres database are automated (daily snapshots), and migrations are run via CI to update schema. The platform complies with data privacy norms (GDPR etc.), with features in core to export or delete user data on request.
* **Offline Content Distribution** – For remote areas, a process exists to package KonnectED content for offline use. On a schedule, a script exports selected knowledge content, media, and a lightweight viewer app onto a portable medium (such as an external drive). This static bundle can be updated periodically and contains an offline web server or static HTML so that remote communities can access KonnectED material without internet. DevOps maintains the build pipeline for these offline packages alongside the main deployment.

## **Kollective Intelligence**

### **Frontend (Kollective Intelligence)**

* **Konsensus Dashboard** – A central **Konsensus** dashboard provides a bird’s-eye view of collective intelligence metrics. It includes visualizations of ongoing **Smart Vote** outcomes and consensus results across the platform. For example, it may show the current consensus on featured Ethikos debates or highlight top-voted solutions in keenKonnect, distilled via weighted voting. Graphs and charts update in real-time or near-real-time to reflect the *Konsensus* (aggregate vote distributions, expert vs public opinions, etc.).
* **Contributor Leaderboards (Krowd Highlights)** – Kollective intelligence UI showcases leaderboards of top contributors (the **Krowd** leaders) by domain. Users with the highest Ekoh scores in various fields (science, arts, etc.) are highlighted, recognizing competence and ethical contribution. This fosters positive competition and transparency about who the leading voices are. The leaderboard pages allow filtering by category or region and are updated as Ekoh weights change.
* **Unified Notifications & Feed** – A combined activity feed shows important events drawn from all modules, filtered by the Konsensus relevance. For instance, if a debate reaches a significant consensus or a project solution is validated by experts, it appears in this feed. The frontend implements this as a scrollable, personalized timeline (akin to a social feed) so that users can see the platform’s collective progress in one place. Users can customize what module events they follow.
* **Admin & Moderation Interface** – Although primarily a backend concern, Kollective intelligence includes frontend tools for moderators or admins to oversee the crowd dynamics. There are interfaces for adjusting system parameters (like weights or categories), reviewing flagged content (from any module), and initiating platform-wide votes or surveys. These admin pages are built with Ant Design components (tables, forms) and protected by admin permissions.

### **Backend (Kollective Intelligence)**

* **Ekoh Reputation Engine** – The core logic that evaluates user contributions and computes expertise-weighted reputation (Ekoh) is implemented as a Django service. It continuously updates users’ scores based on their activities: contributions (articles, projects, arguments) that gain community approval increase domain-specific scores, while unethical behavior (spam, reports) decreases their ethical multiplier. Achievements or credentials can also boost scores. This service likely uses scheduled Celery tasks to recalc scores periodically or triggers updates on certain events (e.g., a project marked successful or an argument upvoted by many experts).
* **Smart Vote Aggregator** – The **Smart Vote** mechanism ties into various voting processes (Ethikos stance voting, internal polls, etc.) to produce weighted results. The backend provides a generic voting API that modules can call: it takes a voting topic and tallies votes, weighting each vote by the voter’s relevant Ekoh score. For example, in an Ethikos topic tagged as “Economics,” the votes of users with high economics Ekoh score count more. The aggregator ensures only relevant experts significantly influence a given vote, while still counting all inputs for inclusivity. Results (percentages, consensus level) are stored and updated in real-time or on demand. This could be implemented using Django Channels (WebSocket) to push live updates to the frontend.
* **Krowd Coordination & APIs** – Kollective intelligence provides common APIs to coordinate across modules. This includes the *global search API* (which queries the unified index for the frontend search bar), a *cross-module notifications API*, and a *feed generator* (compiling events from all modules based on user preferences). Another API is for *content recommendation*: using collective data to suggest relevant content to users (e.g. “users who contributed in KonnectED might like this keenKonnect project”). While each module has its own backend logic, Kollective intelligence acts as an integration layer to ensure they work in concert (e.g., an Ethikos vote triggers an Ekoh update, a high-impact keenKonnect solution triggers a notification to interested users, etc.).
* **Moderation & Quality Control** – The backend includes processes to maintain quality of collective content. For example, **Konsensus validation** might require a certain threshold of expert participation to mark a consensus as “validated.” The system can automatically detect anomalies (like brigading or spam votes) and adjust or flag them. There are admin tools to override or recalc certain outcomes if needed. Content moderation (shared with Common) ensures any user-generated content (comments, posts) that violates guidelines can be flagged and reviewed; Kollective intelligence ensures that low-quality content doesn’t distort the consensus or reputation system (e.g., by weighting feedback from trusted users more heavily, or by temporarily excluding flagged content from calculations).
* **APIs for Data Analysis** – For transparency and research, the backend exposes certain data (with privacy safeguards) for analysis of collective behavior. For instance, an API to fetch anonymized vote distributions or participation metrics can be provided. This encourages third-party analysis of how decisions are reached, supporting the platform’s ethos of transparency and continuous improvement.

### **Database (Kollective Intelligence)**

* **User Reputation Data** – A set of tables store the Ekoh reputation details. Key tables include:  
  + **UserExpertiseScore**: linking user\_id to an expertise\_category and storing the current score (a numeric weight). This is updated whenever contributions are evaluated. It may also store a history or timestamp of last update.
  + **UserEthicsScore**: storing an ethical conduct score or multiplier for each user (could be a single value or per category). Positive contributions (e.g., mentorship, constructive feedback) raise it, whereas reports or harmful behavior lower it. This multiplier is applied to their expertise scores when computing final weights.
  + **ExpertiseCategory**: as mentioned, a reference table of all knowledge domains (with descriptions).
* **Voting Records** – The Konsensus engine keeps records of all weighted votes:  
  + **Vote**: stores individual votes cast on various issues (fields: vote\_id, user\_id, topic\_type (which module/entity the vote is for), topic\_id, raw vote value). Votes in Ethikos, for example, would have a topic\_type “ethikos\_issue” and topic\_id referencing a debate topic.
  + **VoteResult**: stores the aggregated result for a voting topic after applying weights. Contains topic\_type, topic\_id, timestamp, result details (e.g. percentage for/against, consensus strength metric, etc.). This table is updated by the Smart Vote aggregator and read by frontends (like Ethikos or the dashboard).
* **Cross-Module Indexes** – In addition to each module’s content tables, Kollective intelligence may maintain linking tables for cross-module features:  
  + **Notification**: unified notifications with fields (id, user\_id, module, entity\_id, message, timestamp, read\_flag).
  + **ActivityFeed**: could be a materialized log of significant events (like a denormalized table combining key info from each module’s events) for efficient feed generation.
  + **ContentTags**: if a unified tagging or taxonomy is used across modules, a table linking content (by generic FK) to tags or categories (like linking a KonnectED article and a keenKonnect project that both relate to "Climate Change").
* **Moderation Queue** – A common table tracking flagged content or users (with fields for module, content id, reason, status) used by moderators. This ensures problematic contributions are reviewed and, if necessary, not counted in consensus or reputation calculations until resolved.
* **Caching and Performance** – To speed up heavy calculations, certain data might be cached in the DB. For instance, precomputed leaderboards or trending consensus issues can be stored in tables that are refreshed periodically. The DB might also use materialized views for things like top Ekoh users per category to serve the leaderboard quickly.

### **DevOps (Kollective Intelligence)**

* **Scheduled Jobs for Analytics** – The collective intelligence components rely on periodic background jobs. Cron-like schedules (via Celery beat) trigger recalculation tasks (e.g., recompute all Ekoh scores nightly to account for the day’s contributions, or refresh search indexes). The DevOps configuration ensures these jobs run reliably and are monitored. For example, a nightly job might recompute every user’s scores against any new contributions or decay old contributions’ weight to keep scores current.
* **Real-time Infrastructure** – For delivering live updates (like vote counts or feed events), the deployment includes a WebSocket or real-time layer. Using Django Channels (with Redis as a channel layer), the system can push updates to clients. The infrastructure provisions a **Redis** instance not just for Celery tasks but also for WebSocket message brokering. We ensure horizontal scalability by allowing multiple channel workers. If needed, a service like Socket.io or a managed Pusher service could be integrated, but an in-house Channels solution keeps data on our servers.
* **Scalability Considerations** – The Konsensus computations and reputation updates could become CPU-intensive as the user base grows. To handle this, the Ekoh/SmartVote processing can be offloaded to specialized worker pods. DevOps can allocate separate Celery queues or even separate microservice processes for heavy analytics. These can be scaled (more replicas) independently of the main web app. For example, during a major platform-wide vote, additional workers can be spawned to handle the vote tallying quickly.
* **Data Consistency & Backup** – The integrity of reputation data is crucial. Regular backups of the reputation and voting tables are taken (more frequently than other data if needed) to prevent loss of collective intelligence metrics. The DevOps strategy includes failsafes: if the Ekoh calculation fails or returns anomalous results, the system can revert to the last good state (maybe via transaction or backup) to avoid propagating errors. All jobs have error logging and alerting so the team is notified of any issues in the consensus engine calculations.
* **Testing & Simulation** – The collective algorithms are tested with simulation data. The CI pipeline includes tests for the Ekoh weighting logic and Smart Vote outcomes to ensure they behave as expected (e.g., test that higher expert scores truly sway outcomes, test the 7-level nuance handling in Ethikos). DevOps may maintain a staging environment with dummy users and votes to continuously validate that the collective intelligence features scale and respond correctly under load.

## **ethiKos**

### **Frontend (ethiKos)**

* **Debate Forum UI** – **ethiKos** provides a rich web interface for ethical debates and opinion gathering. The main page lists current issues or questions under discussion, with filters for categories (e.g. politics, science ethics) and statuses (open, concluded). Each debate topic has its own page, which includes a **structured debate forum** for in-depth discussion threads and a **position polling section** for quick stance taking.
* **Position-Taking with Nuance** – A key feature is the ability for users to take a stance on a topic as *“For” or “Against”* with gradations of intensity (up to seven levels of nuance from strongly agree to strongly disagree). The UI presents this as a slider or segmented scale (–3 to +3) so users can express degree of support or opposition. The selected stance is visualized (e.g. with color intensity or an icon) to reinforce the nuance. Users can change their stance at any time, and the interface updates the overall tally dynamically.
* **Real-Time Results & Filters** – The stance summary on each topic updates in real-time, showing the evolving **Konsensus**. Results can be filtered by various cohorts using a dropdown or toggle UI: for example, the user can view what **“Scientific experts”** think versus the general public, or filter to **“Validated accounts”** only, or even see the breakdown for **“women over 50”** if enough data is available. These filters trigger calls to the backend to recompute or fetch the filtered results, which are then displayed as charts or percentage bars. A timeline chart might also show how the overall opinion has shifted over time (reflecting dynamic insights).
* **Discussion Threads** – Below the polling interface, a discussion section allows threaded conversations. Users can post arguments (with rich text and links for evidence) in support of or against the topic. These threads can be sorted (e.g. by Ekoh-weighted importance or recency). The UI highlights posts from high-reputation contributors (e.g., an expert’s post might be bordered or labeled) to draw attention to high-quality insights. There is also a visual indicator if a user editing their post changed their stance, to underline how new info can shift views.
* **Ethical Reference Library** – A sub-section of the UI acts as a repository of key ethical documents or prior consensus statements (like a library of “global ethical reference” material). This may be presented as a list of resolved questions or an FAQ style interface where users can see what the consensus on past issues was, serving as a knowledge base for ethical decisions. It’s read-only content that the community or admins have curated from previous debates.

### **Backend (ethiKos)**

* **Debate Topic Management** – The Django backend for ethiKos manages the lifecycle of debate topics. Each **EthikosTopic** entry contains fields like title, description, category, status (open/closed), created\_by, timestamps, etc. The backend provides endpoints to create new topics (by admins or authorized users), list topics, and fetch details (including associated arguments and votes). Business rules ensure quality: e.g., new topics might require moderator approval or minimum reputation to propose.
* **Structured Debate & Arguments** – A model **EthikosArgument** represents an argument or post in a debate thread. It references the user, the topic, possibly tags it as “Pro” or “Con” (if the thread is bifurcated), and stores the content (text, attachments). The backend API supports threaded discussion: an argument can have a parent (for replies), enabling nested threads. There are endpoints for posting arguments, editing, deleting (with permission checks). The system can apply rate-limiting or length limits to ensure discussions remain substantive.
* **Stance Voting Logic** – When a user submits a stance (For/Against + nuance), the backend records it in an **EthikosStance** model (user, topic, value). It then triggers an update to the **Smart Vote** aggregator: essentially, it informs Kollective intelligence’s vote service to recalc the weighted result for that topic. The raw counts and weighted results are stored (in Vote and VoteResult tables as described earlier). The backend can then immediately return the updated results (or broadcast via WebSocket for live update). If a user changes their stance, the previous record is updated and the results recalculated. The logic ensures one active stance per user per topic (no duplicate votes).
* **Filter and Cohort Analysis** – For the advanced filtering, the backend implements queries to slice the vote data. For example, to filter by “scientific experts”, the query finds all stances on that topic where the users have a high Ekoh score in the “Natural Sciences” or related category above a threshold, then computes the weighted outcome among that subset. Similarly, “women over 50” filter would query user profiles for gender and age, then the stances by those users. To facilitate this, user profiles include demographic and expertise info that can be queried, and the stance table is indexed by user for quick lookups. These filtered results are computed on the fly or pre-computed if performance demands (caching common filters for popular topics).
* **Dynamic Insights & Notifications** – The backend monitors changes in overall stance distributions. If a significant shift occurs (e.g., the majority flips from For to Against), it can log this or send a notification to interested users (or simply mark it in the data for frontend to show a “trend” indicator). Also, if a debate reaches a high consensus (e.g., 90% agreement among experts), that might trigger an event broadcast (making it appear in the Kollective intelligence feed as a notable consensus achieved). The back-end ensures such triggers happen responsibly (debates need a minimum participation to count as significant).
* **Moderation & Quality Enforcement** – Ethikos back-end ties into the common moderation system. In practice, this means arguments can be reported; if an argument is flagged by multiple users, it might be hidden until a moderator reviews it. There are also automated checks: for instance, content filtering to detect hate speech or spam in arguments, and disallow or quarantine those posts. Users who repeatedly post such content could have their ability to participate in Ethikos temporarily suspended (and their ethical score in Ekoh reduced accordingly). This ensures the debate stays constructive and evidence-based.

### **Database (ethiKos)**

* **EthikosTopic** – Table storing each debate topic (columns: id, title, description (text), category (FK to a Category table or a simple choice field), created\_by (FK to User), status, etc.). Could include a field for field\_expertise (linking to an expertise category) to indicate which domain the topic falls under (used for filtering expert votes). Also includes counters or summary fields like total votes, last\_activity timestamp, etc., for quick listing and filtering (though these can also be calculated).
* **EthikosStance** – Table recording user stances. Columns: user\_id, topic\_id, stance\_value (integer from -3 to +3, where positive = For, negative = Against), timestamp. Unique constraint on (user, topic) to prevent multiple stances. This table can be large, but queries will typically filter by topic (to get all stances for a debate) or by user (to find a user’s positions). Indexes on topic\_id and user\_id optimize those queries.
* **EthikosArgument** – Table for forum posts. Columns: id, topic\_id (FK), user\_id, content (text), parent\_id (self-FK for replies), side (perhaps an enum {Pro, Con} if we classify arguments supporting or opposing the topic), created\_at, updated\_at. Perhaps a boolean is\_hidden for moderation. There may also be an upvotes count or similar if users can like arguments (though actual weighting of arguments could be by Ekoh externally, an upvote system can still be present for UI feedback). If present, a separate **ArgumentVote** table could store who liked which argument.
* **Validated Accounts / Demographics** – Although not an Ethikos-specific table, the filtering uses user data: e.g., User.is\_validated (a boolean if an account is verified), User.birth\_date (to derive age), User.gender. These fields would be utilized in queries for filters like women over 50, etc. We ensure these fields exist and are appropriately populated (optional for users to provide, perhaps).
* **Historical Konsensus** – Optionally, a table might store snapshots of consensus over time for each topic (to produce the trendline graph). For example, **EthikosTrend** with topic\_id, timestamp (daily or hourly), and aggregated stance metrics (like %for among general, %for among experts, etc.). This can be populated by a periodic job or as part of vote result calculation. It enables showing how opinions shift (as mentioned in dynamic insights).
* **Ethical Reference Library** – If we maintain a curated library of concluded debates or reference positions, these could be stored either in the same Topic table marked as archived/reference or in a separate **EthikosReference** table. The latter could store a summary of the consensus and perhaps a formal statement that was the outcome. This provides content for the ethical reference section on the frontend.

### **DevOps (ethiKos)**

* **Real-Time Updates** – The push infrastructure (WebSockets via Channels) is heavily used in Ethikos to broadcast updated poll results. DevOps must ensure the channel layer is robust and scaled: multiple Daphne or ASGI workers can handle subscription to debate topics, pushing new results to clients. For example, every time an EthikosStance is saved, a signal triggers a Channels group send to all clients viewing that topic’s results. The system is tested for high frequency if thousands of users vote concurrently.
* **Caching Layer for Filters** – Some filter computations can be intensive (especially if a topic has tens of thousands of votes). To optimize, the deployment can include an in-memory cache (Redis or Django cache framework) for filtered results. DevOps might configure a Redis cache that the Ethikos API uses: e.g., store the result of “experts only” filter for a given topic and refresh it every X seconds instead of recomputing on every request. This trade-off yields near-real-time data with reduced DB load. Proper invalidation strategies (like clear cache on new vote or periodically refresh) are configured.
* **Load Testing & Scaling** – The CI/CD process includes load tests for the voting endpoint to ensure it can handle bursts of input. The infrastructure is scaled to avoid bottlenecks: the stance submission endpoint is lightweight (just recording vote and deferring heavy calc to workers if needed), and multiple application server replicas can accept votes concurrently. The database is tuned with indexes and possibly uses logical replication if read-heavy (for example, one primary for writes and replicas for serving the filter queries). The DevOps plan includes the possibility of scaling out read replicas if the filter queries (which can be complex aggregations) start to tax the primary DB.
* **Content Moderation Pipeline** – Any content (arguments, new topics) flagged in Ethikos should alert moderators promptly. DevOps can set up an alert system (e.g., an email or Slack notification) triggered by new entries in the moderation queue related to Ethikos. This ensures timely review to keep discussions healthy. Additionally, backups of Ethikos data are included in routine DB backups; given the possible sensitivity (political or ethical discussions), data retention policies might be in place (for example, keep debate data for transparency but allow users to delete their own posts if needed, which requires careful compliance).
* **Configuration** – The Ethikos module may have specific configuration toggles in environment: e.g., enable/disable certain filters (if gathering demographic info is optional, the filter might not show if data insufficient), or threshold settings (like minimum experts required to calculate an expert-only result). These settings are managed via environment variables or a Django settings module, allowing quick tuning without code changes.

## **keenKonnect**

### **Frontend (keenKonnect)**

* **Project & Solution Explorer** – The keenKonnect UI centers on discovering and collaborating on practical projects. A landing page shows featured projects by category (energy, health, etc.) with thumbnail images or 3D model previews. Users can filter projects by focus area (via categories aligned with core focus areas like Tech Innovation, Sustainable Development, etc.) or search by keyword. Each project has a dedicated page with tabs for *Overview*, *Blueprints & Guides*, *Team & Collaboration*, and *Discussion/Updates*.
* **Blueprint Viewer and AR Previews** – A standout feature is the interactive blueprint/3D design viewer. On a project page, users can view technical designs (uploaded CAD files or images) in-browser. If a 3D model is provided (e.g. in glTF format), the viewer (powered by Three.js or a similar library) allows orbit/zoom and even a VR mode for immersive inspection. Additionally, an **AR mode** is available for supported devices: users can scan a QR code or use a mobile app to overlay the design in their physical environment (for instance, seeing a prototype in their room through AR). This is facilitated by WebXR or an external AR toolkit.
* **Collaboration Tools** – Each project’s Team & Collaboration tab provides virtual tools for team members. This includes a real-time chat or messaging panel for brainstorming and Q&A, a task list or Kanban board for project management (integrated via a library or custom component), and document sharing (lists of attached documents or Google Docs integration). For brainstorming, a simple collaborative whiteboard or mind-map component might be embedded, allowing participants to sketch ideas together. All these tools aim to mimic an innovation lab experience online, accommodating contributors across different time zones.
* **AI Assistance UI** – To leverage AI, keenKonnect’s frontend offers a “Team Builder” and translation toggles. The Team Builder appears when a project founder or member clicks “Find Collaborators”: it suggests users to invite, showing a modal with AI-recommended people (with profile info and matching skills). The UI might highlight why each person is suggested (e.g. “Alice – experienced in solar energy, 5 projects completed”). The user can select and send invites from this interface. Also, for communication, a translate button on messages or posts can instantly show the content in the user’s preferred language (using machine translation behind the scenes).
* **Merit Display and Validation** – In line with community validation of solutions, project pages display a “community validation” score or badge. For instance, once a solution is built and tested, other users can rate its practicality/impact. The UI shows an aggregate of these ratings (like 95% positive, or a medal icon if a project is validated by consensus). Projects that have reached certain validation milestones might be marked as “Konsensus Validated Solution” in the listing. Additionally, contributors’ roles and Ekoh reputations are visible on the project page (e.g. lead, mentor, contributor with expert badge) to recognize expertise and encourage trust.

### **Backend (keenKonnect)**

* **Project Management** – A Django app for keenKonnect handles **Project** objects. Each project record includes fields: title, description, category (e.g. matches focus areas), created\_by (owner), status (idea, in progress, completed, validated), and timestamps. Backend APIs allow creating new projects (any user can propose an idea), editing details, and listing projects (with filters by category or status). Business logic: possibly require at least one blueprint or detailed description before a project can be marked “in progress”, etc.
* **Blueprints and Resources** – Projects can have multiple associated **Blueprint** or **Resource** entries. These could be stored in a model with fields: project (FK), file path or URL, type (CAD model, document, image, etc.), and metadata (e.g. description, version). Upload endpoints (secured via authentication) handle storing these files (to S3 or server disk) and creating the DB entry. If a file is a 3D model, the backend might generate a web-friendly version or preview (perhaps leveraging a Celery task to convert a CAD file to glTF if necessary).
* **Collaboration & Communication** – The backend supports real-time collaboration features: for chat, it might use Django Channels to route messages. However, a simpler approach is to have a **ProjectMessage** model (project FK, user FK, content, timestamp) and an API endpoint to fetch/send messages, combined with WebSockets for push. For task boards, a **ProjectTask** model can track tasks (with fields: title, description, status, assignee, etc.), and an endpoint for updating tasks. These allow basic project management. If integrated with external tools (like Google Docs), credentials and webhooks would be managed here (though likely out-of-scope for now; initial implementation keeps it in-house).
* **AI Integration** – The backend implements the **AI-Powered Team Formation** logic. This likely uses a module or service that can query user profiles and project requirements to find matches. When a request is made (e.g. user opens Team Builder), the backend could run a recommendation function: input is the project’s keywords and needed skills (maybe derived from its category or tags), and the output is a list of users. This could be done via a simple algorithm initially (e.g. find users with high Ekoh scores in related domains who are active), or via a more complex ML model if available. For translation, the backend either calls a third-party API (like Google Translate) or uses an on-prem model to translate text on request. It likely caches translations for frequently repeated content.
* **Merit-based Validation** – To implement “solutions rising through community validation”, the backend provides endpoints for users to rate or endorse a project’s outcome. A **ProjectRating** model might store user, project, score (e.g. +1 upvote or a 1-5 star). The backend can compute an aggregate validation score. Moreover, by tying into Ekoh/Smart Vote, these ratings could be weighted by the rater’s reputation; however, the initial approach might simply count votes or upvotes. Once a project reaches certain criteria (e.g. X upvotes from experts, or passes a threshold in a weighted vote possibly initiated on project completion), the backend marks it as *validated*. It could then trigger a notification (and possibly an Ethikos-like vote result if formal voting was used for critical projects).
* **Versioning & Archiving** – The backend may support versioning of project blueprints or documentation. Each blueprint upload can have a version number or flag to indicate if it supersedes a previous one. Old files might be archived but still accessible. For completed projects, an archival process might move them to a historical repository (still accessible read-only for learning purposes). The API could then allow querying active vs archived projects separately.
* **Access Control** – Projects may be public (default) or could be created as private or invite-only (if sensitive), though keenKonnect philosophy likely leans open. The backend enforces that only team members can post in the collaboration sections (others might only comment or view). Team membership is managed via a **ProjectTeam** model (project FK, user FK, role), with roles like owner, collaborator, mentor, etc. The API ensures actions like adding blueprints or tasks are restricted to team members. Joining a project could be via request: an endpoint to request to join, which notifies the project owner or auto-adds if open collaboration is allowed.

### **Database (keenKonnect)**

* **Project** – Core table for projects (id, title, description (text), category, creator\_id, status, created\_at, updated\_at, etc.). Category might reference a core category table if standardized (like linking to the ExpertiseCategory table, or a separate enumerated list of domains identical to focus areas).
* **ProjectResource** – Table storing references to project files/resources. Columns: id, project\_id, file\_path (or URL), file\_type (maybe MIME or an enum like “image”, “3d\_model”, “document”), title/description, uploaded\_by, uploaded\_at. Possibly additional fields like converted\_path for a processed version of the file (e.g. a preview image or converted model).
* **ProjectMessage** – If using persistent chat, a table for messages: id, project\_id, user\_id, content (text), timestamp. This can grow large; for performance, old messages could be archived or limited. In a simple approach, this table retains all messages.
* **ProjectTask** – Table for task tracking: id, project\_id, title, description, status, assignee (user FK, nullable), created\_at, due\_date (nullable), order (for sorting). Status could be an enum (todo/in-progress/done). This supports a lightweight Kanban board.
* **ProjectTeam** – Table linking users to projects with roles: id, project\_id, user\_id, role (string or enum: owner, collaborator, mentor, etc.), joined\_at. Could enforce one owner and allow multiple collaborators. Used for permission checks and for the AI to identify gaps in team composition.
* **ProjectRating** – Table for community validation: id, project\_id, user\_id, value (e.g. +1 or a rating scale). If needed, include a field for whether the user is considered an expert in this domain (for reporting purposes). Alternatively, separate counts can be derived (like count of expert upvotes vs others).
* **Tags/Focus** – Possibly a many-to-many for tags if projects can have multiple tags beyond the main category. This could reuse a general Tag table if one exists.
* **AI Data** – Not a formal table, but the AI recommendations might rely on aggregated data like user skill profiles. For example, if using the ExpertiseCategory and UserExpertiseScore from Ekoh, the recommendation logic queries those directly (no need for separate store). If more complex, a precomputed similarity matrix or embedding store could be used, but likely not at this stage.

### **DevOps (keenKonnect)**

* **File Storage and CDN** – Given potentially large blueprint files and images, the deployment uses an object storage service (like AWS S3 or Azure Blob). The Django backend uses appropriate storage backend to upload files there. For serving, a CDN is configured to deliver these assets quickly worldwide. DevOps sets up bucket policies or signed URLs to protect any private files, though most are public. Large files might need increased upload limits on the web server and chunked uploading support (the frontend could use a chunk upload for very large files, so the backend might need to handle that or use a direct-to-S3 upload approach).
* **3D/AR Support** – For AR/VR features, DevOps ensures that the necessary HTTPS and WebXR configurations are correct (e.g., some AR features might require SSL and specific headers). If a separate service or worker is needed to convert models, that is deployed (e.g., a Celery worker with Blender or other converters installed to process CAD to web formats). Such tasks could be resource-intensive, so perhaps they run in an isolated environment.
* **Real-Time Collab** – The chat and possibly simultaneous editing (if whiteboard or similar) require real-time communication. The same Channels/Redis setup from core can be reused. DevOps might allocate a separate channel group or WS endpoint specifically for project collaboration to segregate traffic. Load considerations: if many projects have active chats, ensure enough channels workers are running. In testing, simulate multiple active chats to ensure performance.
* **AI Services** – If the AI collaborator suggestion is complex, DevOps might containerize a small recommendation engine or use a serverless function. For example, a Python script using scikit-learn or a lightweight ML model might run periodically to update a list of recommended collaborators for each project (caching results). Alternatively, on-demand calls to an external AI service (like OpenAI or a custom model) might be used; these require API keys and latency considerations. DevOps must secure those keys and possibly provide a fallback if the external API fails. Rate limiting is also important to avoid abuse (especially for translation services if used heavily).
* **Monitoring Project Health** – Additional monitoring might be placed on keenKonnect processes because they handle file uploads (monitor storage space, file antivirus scanning if needed) and because collab features could be points of failure (like a stuck Celery job on file conversion). Alerts for low storage or failing file conversions help maintain service quality. Backups include project data and possibly the metadata of files (actual files are on S3 which has its own redundancy).
* **Onboarding & Documentation** – From a DevOps perspective, supporting the keenKonnect community might involve providing documentation on how to structure projects, what file formats are supported, etc. While not code, ensuring the deployment includes a static help section or documentation accessible in the app could be part of the deliverable. This may simply be a set of markdown files in the repo rendered by the frontend, but making sure it's updated and available (and perhaps translatable) is a detail to manage.

## **KonnectED**

### **Frontend (KonnectED)**

* **Knowledge Library Interface** – **KonnectED** features a comprehensive knowledge library UI. The homepage offers a categorized catalog of educational content, organized by subjects (e.g. Science, History, Ethics) and skill areas. Users (especially young learners or educators) can browse by category or search by keywords. Content cards show titles, brief descriptions, and indicators like content type (article, video, lesson) and language availability.
* **Content Viewer & Course Module** – Clicking content opens a viewer page. If the item is an article or lesson, it’s displayed in a reader-friendly format (with images, videos embedded as needed). If it’s part of a course or learning path, the UI shows a sidebar or progress tracker for the sequence of lessons. Users can mark lessons as completed. The viewer also supports interactive elements (quizzes or knowledge checks) if present – these might be simple Q&A components embedded in content.
* **Contribution & Curation Tools** – There is a *community contribution* UI allowing educators or experts to submit content. This might be accessible via a "Contribute Knowledge" button. The submission form supports rich text, media uploads, and tagging of the content with subject, level, and language. The frontend guides contributors to adhere to format. Once submitted, content may go into a pending state for review.
* **CertifiKation & Mentorship** – A **CertifiKation** section in the UI lists available skill certifications and vocational training modules. Each certification has a page describing requirements (e.g. complete certain lessons or pass an evaluation). Users can track their progress towards each certificate. Upon completion, the UI allows downloading a certificate (PDF) or displaying a badge on the user profile. Additionally, a Mentorship page connects learners with volunteer mentors: it shows mentor profiles and expertise, and an interface to request mentorship or join cross-age learning groups. This might function akin to a forum or matching system where a user can post “I want to learn X” and mentors can respond.
* **Offline Access Mode** – Recognizing limited connectivity, KonnectED’s frontend includes an *offline mode*. Users can select content to “Save for Offline,” bundling text and media for use without internet. If a device is offline (or using an external drive with preloaded content), the UI detects it and switches to a simplified offline interface – a static content viewer and basic navigation without dynamic search. This mode uses previously cached data or the data on the external drive. The design is minimal to run on older browsers and low-power machines (e.g., avoids heavy scripts, uses plain HTML/CSS for most of offline UI).

### **Backend (KonnectED)**

* **Content Management System** – KonnectED’s backend functions partly like a CMS. The **KnowledgeContent** model (or a set of models) handles educational items. Depending on complexity, we might have a **Content** model for generic items and specialized models if needed (Lesson, Article, Video, etc., or just fields to indicate type). Django admin or custom endpoints allow creating and editing content. Each content item includes fields: title, body (could be HTML/Markdown), subject, recommended age or level, language, type, and status (draft/published). Content submitted by users might go into draft/pending status until approved by a curator or sufficiently upvoted by experts.
* **Weighted Content Curation** – The backend integrates **Ekoh’s Smart Vote** for content ranking. This means when multiple pieces of content cover similar topics, or when new contributions come in, the system can employ a voting mechanism. For example, experts in the topic can vote on the quality/usefulness of a new article; these votes are weighted by their Ekoh scores, and the content can be sorted or labeled as “expert-verified” once a threshold is met. Implementation-wise, this could reuse the voting tables: content could be a votable entity where the result influences a quality\_score field on the content. Periodic processes might push top-voted content “forward” (e.g., mark as recommended).
* **Learning Paths & Certifications** – If courses are defined, a **Course** model ties together multiple content items in sequence. It can store the ordering, prerequisites, and a link to a **Certification** if completion grants a certificate. The backend tracks user progress via a **UserProgress** model (user, course, current lesson, completed flag). For certifications, a **Certification** model defines criteria (could be a relation to a Course or a set of content IDs and an optional exam or project). When criteria are met, a **UserCertification** record is created for that user. Possibly an automated or admin-reviewed process issues it; for automated, the backend listens for events (like “user completed course X”) and then creates the certification record and triggers a notification.
* **Search and Localization** – The backend provides robust search and filter endpoints for content (by subject, keyword, language). If using a search index, queries are routed to it. There’s also an API to fetch available languages for a content piece or to get a translation. Content can be stored in multiple languages either by separate objects or a translation table. The backend ensures that if a user requests a piece in another language and a human translation exists, it serves that; if not, it could fallback to machine translation on the fly (though more likely they rely on contributed translations). The system encourages multilingual contributions to provide content in as many languages as possible.
* **Offline Data Export** – The KonnectED backend includes functionality to generate offline content packages. For example, an endpoint or management command can gather all content (or a subset for a region/curriculum) and output it as a static bundle (collection of HTML files, media, and a JSON or SQLite database). This ties into the DevOps pipeline for offline distribution. The backend might also allow downloading specific content packages from the UI (e.g., “Download all resources for Grade 8 Science”). This requires collating content and compressing it, possibly handled by Celery tasks due to size.
* **Mentorship Matching** – The backend supports the mentorship feature. Mentors could be a special user flag or a separate **MentorProfile** model (linked to user, listing areas of expertise and availability). A user can send a mentorship request (create a **MentorshipRequest** record linking a mentor and a mentee, possibly with a message). The mentor can accept or decline. The backend then might create a connection (like a chat channel or assign them to a “learning group”). It could simply be managed via messaging: once accepted, the pair can communicate (perhaps the system automatically sets up a private discussion thread or email exchange). This is relatively lightweight: essentially storing the relationship and maybe facilitating scheduling (not necessarily with built-in calendar, but mentors and mentees can decide externally or via chat).

### **Database (KonnectED)**

* **KnowledgeContent** – Main table for content. Fields: id, title, body (could be large text for articles; for videos could be a link or embedded code), subject (FK to Subject table if exists, or a char field), level (e.g. age group or difficulty), language (FK to Language or ISO code field), type (enum: lesson, article, video, etc.), created\_by, created\_at, status. Possibly parent\_id if some content is a child of another (like part of a course or multi-part series). If using separate models for different types, they might all inherit from a base content model (Django model inheritance).
* **Course** – Table linking content into sequences. Fields: id, title, description, maybe subject (or it can infer from included content), creator, etc. There could be a many-to-many linking Course to Content with an through table that has an order field (to sequence the lessons).
* **Certification** – Table for certifications. Fields: id, name, description, maybe related course (if one course equals one cert), or criteria definition (this could be a JSON field or a set of foreign keys pointing to required content or tasks). Could also store if an exam or project is needed externally to fully certify.
* **UserProgress** – Tracks course progress: id, user\_id, course\_id, last\_content\_id (the last lesson completed), completed (boolean or completion date). Alternatively, one could track individual content completions (UserContentDone table user\_id, content\_id, timestamp) and derive course progress from that.
* **UserCertification** – Records issued certificates: id, user\_id, certification\_id, issued\_on, issuer (who verified or if auto). Could also store a unique certificate code or URL for verification.
* **Contribution & Review** – If contributions require review, a **ContentSubmission** table could hold user-submitted content separate from published content. Fields similar to KnowledgeContent but flagged as pending and maybe with a review\_status. Once approved (perhaps by an expert reviewer), it gets merged into KnowledgeContent. If using the same table with a status flag, then content table has status (pending/published) and maybe a reviewer\_id and reviewed\_at.
* **ContentRating/Votes** – For the Smart Vote curation, a **ContentVote** table can log votes by experts on content quality. Fields: user\_id, content\_id, vote (e.g. +1 or rating). The aggregated outcome might go to a **ContentQuality** table or as fields on content (like content.upvotes, content.downvotes, content.rating). Weighted aspect can be computed on the fly or stored if needed (like an Ekoh-weighted score).
* **Subject/Category** – A taxonomy table listing subjects or knowledge categories for organizational purposes (e.g. Math, Science, etc.). Content links to one or many. This might reuse the ExpertiseCategory from core if they align (though those were more academic fields, which could overlap with subjects).
* **MentorProfile & Mentorship** – If implemented, MentorProfile has user\_id, bio, areas\_of\_expertise (maybe link to expertise categories or free text), availability (e.g. hours or number of mentees they can take). **MentorshipRequest** table: id, mentor\_id (user), mentee\_id (user), status, message, created\_at, responded\_at.

### **DevOps (KonnectED)**

* **Content Delivery & CDN** – Many educational contents include media (images, videos). Videos especially can be heavy; for those, the platform might rely on an external solution (like embedding YouTube or using a streaming server). If hosting videos, the DevOps should use a streaming service or at least store them in an optimized format and serve via CDN. For images and PDFs, ensure they are stored in object storage and cached via CDN for global access.
* **Offline Package Generation** – A specialized job (could be a management command run periodically) generates the static offline packages. DevOps coordinates the scheduling and distribution of these. Possibly, a GitHub Actions workflow could trigger building an offline package artifact whenever content is updated significantly. The output might be versioned (e.g. “KonnaxionKonnectED\_offline\_v2025Q1.zip”) and made available for download. Ensuring this zip is not too large or can be segmented by topic might be a consideration (maybe one per subject or curriculum).
* **Performance & Caching** – Searching and listing content can be database-intensive if content grows. DevOps might deploy a caching layer (Redis or similar) for frequently accessed queries or pages (e.g. cache the homepage content lists, or popular content pages). Also, if using a search engine like ElasticSearch, that cluster needs to be maintained (monitor memory, reindex as needed).
* **Scalability** – The KonnectED module could attract a large number of concurrent users (e.g., a classroom of students accessing simultaneously). The infrastructure should scale horizontally: multiple instances of the Django API behind load balancers to handle many content requests. Next.js can leverage ISR/SSG for largely static content like published articles (serving them as static pages, updating periodically when content changes). Using Next’s static generation for published content pages can drastically reduce load on the Django API (as the content can be pulled at build time). DevOps might set up incremental static regeneration for content pages, if feasible.
* **Data Integrity & Backup** – Educational content is core to the platform’s value. Regular backups of the content database are critical (with versioning to recover from accidental deletions or corruption). If there’s a lot of user-generated content, also backup the storage for images/videos. Perhaps maintain an archive of all published content (even if deleted later, for auditing or restoration). Also, consider compliance: if users can create content, we need a way to remove content upon request or handle personal data in it, though likely educational content is not personal.
* **Monitoring Usage** – It’s useful to monitor which content is accessed frequently to inform improvements. DevOps might integrate analytics (privacy-respecting) to track page views and downloads. Also, monitoring memory and CPU usage for large content (like huge PDFs or many concurrent video streams) ensures the server and CDN are coping. High traffic spikes (like a teacher telling 100 students to download a package at once) should be tested and planned for (maybe suggest pre-downloading or torrent distribution for extreme cases).
* **Integration Testing** – The CI should run tests for content rendering and course logic. For instance, a test to ensure completing a course issues a certificate, or that the offline generation script runs without errors. If using translations, tests to ensure switching languages returns the correct content are valuable. These automated tests help maintain the quality of the KonnectED functionalities through updates.

## **Kreative**

### **Frontend (Kreative)**

* **Virtual Gallery Showcase** – The Kreative module provides immersive gallery experiences for art. On the main page, users can explore *virtual galleries* featuring curated artworks. This could be implemented as a 3D gallery environment (using WebGL/Three.js) or a 2D mosaic view depending on device capability. Users can navigate through gallery “rooms” to view art pieces, with controls to move around or switch to a simple list view if 3D mode is not supported. Artworks are displayed with high-resolution images and descriptions when clicked. A VR mode toggle allows users with VR headsets to enter a fully immersive gallery space.
* **Creative Collaboration Workspace** – Kreative offers an online workspace where artists can co-create. This might manifest as a shared canvas or project page similar to keenKonnect but focused on creative works. For example, a collaborative painting tool or music composition tool embedded in the browser. Initially, it could be simpler: e.g., an image gallery where multiple artists upload drafts and comment on each other’s work, or a synchronized drawing board for two or more users. The UI would display the participants and provide drawing or editing tools if applicable. The workspace also includes chat for real-time communication during co-creation.
* **Interactive Exhibitions** – Aside from static galleries, **interactive exhibitions** are featured. These could be special pages or events where art is combined with interactive elements (for instance, a 3D sculpture that users can manipulate, or an AR experience where users scan a QR code to see art in their environment). The frontend handles these by loading the appropriate interactive scripts or AR components. One example: an exhibition of cultural heritage artifacts might let users rotate and examine 3D models of the artifacts, with hotspots to learn about each part.
* **Mentorship Program Interface** – There is a dedicated section listing mentorship opportunities and cultural preservation projects. Mentor profiles (experienced artists or craftsmen) are displayed, indicating what traditions or arts they teach. A user can click a mentor to see details and maybe schedule a session or send a request to be mentored. Similarly, for *digital archiving of endangered traditions*, a submission interface allows users to contribute media (photos, videos, descriptions) of a cultural practice. The UI for this is a form where they describe the tradition and upload supporting media. These contributions are then showcased in an “Archive” gallery, possibly as a timeline or world map interface where users can explore by region/culture.
* **Global Art Community Features** – The Kreative frontend also provides community features like following favorite artists, commenting on artworks, and upvoting or “appreciating” pieces. Artworks have a detail page with the artwork image/video, description, artist info, and a comment thread. Users can give “Kudos” (a form of like) to artworks. The interface might also highlight diversity: for example, filters to view art from different regions or styles, emphasizing the global reach and inclusivity of the platform.

### **Backend (Kreative)**

* **Artwork & Gallery Management** – A **KreativeArtwork** model stores individual artworks with fields: title, description, artist (FK to User or a separate Artist profile), media file path(s), and metadata like year, medium, style. The backend provides endpoints to create and update artworks (for artists uploading their work) and to retrieve them for gallery displays. A **Gallery** model could represent a virtual gallery or exhibition, containing a set of artwork references (many-to-many). Curators or admins can create galleries (curated sets) via an endpoint or admin panel, selecting artworks to include and designing the order/layout (the layout might just be implicit or a simple ordering since actual spatial layout is handled in frontend).
* **Collaboration Sessions** – If a real-time co-creation canvas is provided, the backend handles session management. Possibly a **CollabSession** model (id, title, created\_by, type (painting, music, etc.), started\_at, etc.). When users join a session, the backend (via WebSocket or other protocol) relays their actions (e.g., brush strokes or edits). We might use a simpler approach at first: periodic save of the collaborative artifact (like saving the image every X seconds). The backend provides an endpoint to fetch the current state of a collaboration (like the latest image) and to post updates (though for truly live collaboration a peer-to-peer or specialized service might be better; still, an MVP can use broadcasting through channels).
* **Mentorship & Archive Backend** – For mentorship, possibly reusing or extending the mentorship request system from KonnectED (or a separate **ArtMentorship** model if needed to differentiate). Mentors can be identified by a flag or separate profile with field of art they mentor. The backend processes mentorship requests similarly to KonnectED: create request, notify mentor, allow accept/decline. For archiving endangered traditions, a **TraditionEntry** model stores submissions: fields for title, description, culture/region, media files, submitted\_by, approved (boolean). Admins or designated experts review these submissions (maybe requiring approval before public listing to ensure accuracy and respect). Once approved, they become part of the public archive content.
* **Community Interaction** – A **Comment** system (which might be global but filtered by module) stores comments on artworks. The backend uses the common commenting or a custom one for Kreative to allow threaded comments on each artwork or exhibition. Similarly, a **Kudos** or like system can simply be a counter on artworks plus a record if needed (or just count unique user likes in a separate table). These APIs are straightforward: endpoints to post a comment, like an artwork, etc., with proper auth.
* **User Profiles & Portfolios** – Artists might have extended profiles or portfolio pages. The backend might treat an artist’s uploaded artworks as their portfolio. By querying all artworks by a user, it produces their portfolio listing. If more structure is needed, a **Portfolio** model could allow artists to curate their own sets of work, but likely unnecessary initially. The existing user model can cover basic info, with maybe an added field “is\_artist” or a profile type that the UI can display accordingly.
* **Cultural Data** – The system might integrate with external cultural heritage databases or at least allow linking out. For example, if an artifact is documented, you might store references or links to museum records. But likely out of scope; we focus on user submissions.
* **Moderation & Rights** – The backend enforces that users can only upload content they own or have rights to (there might be a terms acceptance). A content flagging mechanism is in place: if someone reports an artwork (for offensive content, copyright issues, etc.), it goes to the common moderation queue. The backend can auto-hide content that is flagged multiple times until reviewed. Also, for underage users in art, ensure any NSFW or sensitive content is flagged and hidden behind warnings (the submission form might include a “mature content” flag which, if set, requires confirm to view).

### **Database (Kreative)**

* **KreativeArtwork** – Table for artworks: id, title, description (text), artist\_id (FK to User), file\_path (or multiple if e.g. different resolutions or a video thumbnail, etc.), media\_type (image, video, audio if any), created\_at, perhaps fields like style or tags (could be many-to-many to a Tag table for art genres). If multiple files (like an artwork could have several images or a video plus images), a separate ArtworkMedia table might handle that, but a simple approach is one main file and maybe additional media links in JSON.
* **Gallery** – Table for curated galleries/exhibitions: id, title, description, created\_by (FK User or maybe a special curator user), created\_at, maybe theme. A join table GalleryArtwork links Gallery to Artworks with an order field. Possibly also a layout descriptor (if we want to store how an art is placed in a 3D space – but likely we won’t store coordinates; the frontend might just arrange them arbitrarily or in a preset template).
* **CollabSession** – If using sessions: id, name, host\_id, session\_type, started\_at, ended\_at (nullable if ongoing). If we store collaborative art output, maybe link to an Artwork or store the final output file path on session end. If using persistent state, a separate table or data store might capture intermediate steps (but that could be too granular; often such live collab might rely purely on WebSocket without storing every action).
* **TraditionEntry** – Table for cultural archive entries: id, title, description, region (could be a string or FK to a Region table), media (similar approach to artworks, maybe one primary media file or multiple), submitted\_by, submitted\_at, approved (bool), approved\_by, approved\_at. If not approved, it's only visible to moderators. Once approved, it can be treated similarly to an Artwork in terms of display.
* **Mentorship** – If separate from KonnectED’s, an **ArtMentorProfile** (user\_id, art\_styles, bio) and **MentorshipRequest** (id, mentor\_id, mentee\_id, message, status) similar to earlier. We could reuse one system for all mentorship with a field for area, but keeping separate might allow different handling.
* **Comment & Kudos** – A **Comment** table keyed by content type and id could be reused or separate for art: e.g. a generic Comment model with fields (id, user\_id, content\_type, object\_id, text, created\_at, parent\_id for threads). Or simpler, **ArtComment** with art\_id, user\_id, text, etc., if we assume only comment on artworks (but they might also comment on traditions or galleries). For likes, a **ArtworkLike** table or just a count on artwork with a separate **UserArtworkLike** if needed for preventing multiple likes.
* **Tagging** – Possibly a Tag model and a mapping table if we want to label artworks by genre, style, or culture. Could use a generic tagging library or have separate fields, but for future flexibility, a tagging table is good.

### **DevOps (Kreative)**

* **Media Storage & Processing** – Artworks can be high-resolution images or videos. The DevOps must ensure efficient storage (again leveraging S3/Cloud storage). For images, on upload, a Celery task can generate multiple resolutions (thumbnails, screen size, original preserved) for performance. For video or audio content, transcoding might be necessary (e.g., ensure videos are in web-friendly format H264 MP4 or WebM). This could be done by integrating a tool like ffmpeg in a worker container. Large media might also need virus scanning if security is a concern.
* **3D/VR Hosting** – If the virtual gallery uses 3D models (for environment or 3D art), those models (glTF, etc.) should also be stored and possibly optimized. DevOps might involve creating light versions of heavy 3D assets for web performance. Also, hosting a VR environment might require specific web server settings, but usually static assets suffice. Monitoring client performance (via logs or analytics) can inform if assets are too heavy (e.g., if many drop-offs or long load times in gallery).
* **Real-Time Collab Infra** – The collaborative painting or creation feature, if real-time, will likely use WebSockets. The same Django Channels setup can be employed, but the nature of drawing events (rapid, frequent messages) might demand careful scaling. If the traffic is heavy, consider using a specialized service or protocol for real-time drawing (like WebRTC data channels for peer-to-peer), but initially channels+Redis can manage small groups. DevOps ensures adequate message throughput: possibly increasing Redis capacity or using a pub/sub service.
* **CDN for Art** – Using a CDN is important so global users can view art without latency. Also enabling HTTP/2 or HTTP/3 for efficient loading of many small assets (like if a gallery loads dozens of image thumbnails). The static content for the 3D galleries (JS libraries, models) should also be on CDN or at least cached.
* **Backups and IP Rights** – Backup the art database and media, but also consider intellectual property. For safety, the platform might maintain internal backups but not expose them (users may delete art if they choose, etc.). There should be an internal policy (not necessarily in spec, but in operation) to handle takedown requests. DevOps may involve deletion pipelines for removing content thoroughly when needed (including from caches).
* **Analytics & Community Health** – Monitoring user engagement (how many likes, comments, active mentorships) can guide resource allocation. If a particular feature (like VR exhibitions) is rarely used, maybe scale it down; if mentorship requests surge, maybe allocate more support.
* **Event-based Scaling** – Art events (say a big virtual exhibition opening) might spike traffic. DevOps should have a procedure to handle scheduled high-traffic events: e.g., temporarily upsize the number of web server instances, enable a cache for gallery data, etc. Possibly run a load test in advance.
* **Integration with Core** – Ensure that any cross-module interactions (like using Ekoh in art – though not explicitly needed, but maybe high Ekoh users could get featured? Or art contributions could reflect in their profile) are not broken by deployments. This implies running integration tests where an action in Kreative triggers something in core or vice versa. For example, uploading a new artwork might increase a user's reputation in "Arts" category; that linkage should be tested and working.