# Cosolvent System Design

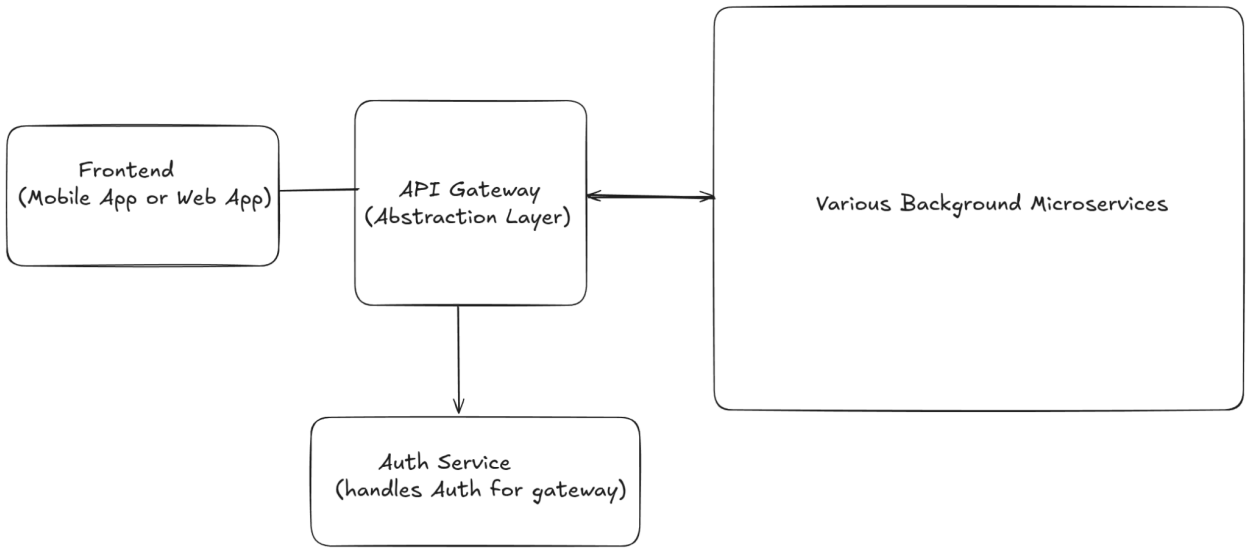
**Core Philosophy Adopted for this Design:**

- **Decentralization & Single Responsibility:** Each microservice will own a distinct business capability.
- **API and Event-Driven:** Services communicate primarily via well-defined APIs (REST or gRPC) and asynchronously via a message bus.
- **Data Isolation:** Each service manages its own data, preventing tight coupling at the database level.
- **Technology Standardization:**
    - Python for backend services. We'll use FastAPI.
    - Typescript (NextJS) for Frontend.
- **LLM Abstraction:** Critical for easy swapping. We'll need a way to abstract LLM calls.

# Proposed Microservice Architecture

Here's a breakdown of potential microservices. Remember, the exact boundaries can be iterated upon.

**Diagrammatic Overview (Conceptual):**




Backend Microservices communicate via:
1. Direct API calls (synchronous, for immediate needs)
2. Message Bus (asynchronous, for decoupling and background processing)

**1. User & Authentication Service (`user-auth-service`)**

- **Responsibilities:**
  - User registration (collecting minimal initial metadata).
  - User login and session management (JWTs).
  - Manages user identity, basic profile info (name, contact, roles).
  - Handles authorization and permissions (e.g., user can only access their own data).
  - Manages user policy metadata (where profiles/assets may be used).
- **Data Store:** Relational DB (e.g., PostgreSQL) or NoSQL (e.g., MongoDB/DynamoDB).
  - Recommendation: NoSQL for prototyping, SQL for production
- **Key APIs:**
  - `POST /register`
  - `POST /login`
  - `GET /users/me`
  - `PUT /users/me/policy`
  - `GET /users/{user_id}/profile-summary` (basic, not the LLM one)
- **Interactions:**
  - Called by the API Gateway for every authenticated request.
  - Publishes `UserRegistered` event.

**2. Asset Ingestion & Management Service (`asset-service`)**

- **Responsibilities:**
  - Handles file uploads (documents, images, videos) from users.
  - Stores raw assets in a dedicated, user-specific location (e.g., S3 bucket with `user_id/` prefix).
  - Manages metadata about assets: filename, type, size, upload date, user_id, storage path, processing status (e.g., "pending_translation", "translated", "metadata_extracted").
  - Provides basic CRUD operations on assets for users (add, replace, delete their *own* assets).
  - Handles policy-driven asset storage (ensuring user data isolation).
- **Data Store:**
  - Object Storage (AWS S3) for the files themselves.
  - NoSQL DB for asset metadata.
- **Key APIs:**
  - `POST /users/{user_id}/assets` (upload)
  - `GET /users/{user_id}/assets`
  - `GET /users/{user_id}/assets/{asset_id}`
  - `DELETE /users/{user_id}/assets/{asset_id}`
  - `PUT /assets/{asset_id}/metadata` (internal, for processing services to update status)

- **Interactions:**
    - Receives upload requests from frontends (via API Gateway).
    - Publishes `AssetUploaded` event to message bus.
    - Listens for `AssetProcessingFailed` events to mark assets.
    - Provides asset access to processing services.

**3. Metadata Extraction Service (`metadata-extraction-service`)**

- **Responsibilities:**
    - Check metadata from images. (Edited / AI Generated / etc..)
    - Uses VLMs to extract descriptive metadata from images (e.g., "this photo is a John Deere 400hp Tractor").
    - Uses LLMs to mine curated industry context documents for generally useful variables/descriptors.
    - Potentially extracts descriptive metadata from user-uploaded documents.
- **Data Store:** Stores extracted metadata associated with asset IDs or industry context document IDs. This might be updating the asset's metadata record in `asset-service` or its own specialized store if metadata is complex.
- **Key APIs (Internal):**
    - `POST /extract/image` (accepts asset ID)
    - `POST /extract/document` (accepts asset ID)
    - `POST /extract/industry-context` (accepts context document ID)

- **Interactions:**
    - Subscribes to `AssetUploaded`.
    - Fetches asset/document content from `asset-service` or `industry-context-service`.
    - Calls the `LLM Orchestration Service` for metadata extraction tasks.
    - Updates asset metadata in `asset-service` or its own store.
    - Publishes `MetadataExtracted` event (specifying asset or context doc).

**4. Translation Service (`translation-service`)**

- **Responsibilities:**
    - Detects the language of uploaded text-based assets.
    - Translates assets to English if they are not already in English.
    - Integrates with the chosen LLM or NLP service for translation.
    - Manages different translation models/providers (part of LLM swappability).
- **Data Store:** Potentially a cache for common translations or to avoid re-translating unchanged documents. Stores translated versions alongside originals or as new asset versions.
- **Key APIs (Internal):**
    - `POST /translate` (accepts asset ID or text content)

- **Interactions:**
    - Subscribes to `MetadataExtracted` event.
    - Fetches asset content from `asset-service`.
    - Calls the `LLM Orchestration Service` for translation.
    - creates a new "translated" asset.
    - Publishes `FinalAsset` event.

**5. Profile Generation Service (`profile-generation-service`)**

- **Responsibilities:**
    - Uses a RAG system with user assets to respond to a standard system profile-builder prompt.
    - Handles user-supplied instructions to append to the system prompt for profile regeneration.
    - Generates a "pending" profile for user review.
    - Ensures no registered user's data is ever integrated or confused with any other *during profile generation*.
- **Data Store:** None directly; it orchestrates data from other services and produces a temporary output.
- **Key APIs (Internal or triggered by events -> Metadata Extraction):**
    - `POST /users/{user_id}/generate-profile`
- **Interactions:**
    - Subscribes to `FinalAsset` (a new event, possibly published by `asset-service` once translation and initial metadata extraction are done for a new asset) or `UserProfileRegenerationRequested` event.
    - Fetches user assets (text, key metadata) from `asset-service`.
    - Fetches the standard profile-builder prompt (from `admin-service`).
    - Calls the `LLM Orchestration Service` with the RAG setup (user assets as context) and prompt.
    - Sends the generated draft profile to `profile-management-service`.

**6. Profile Management Service (`profile-management-service`)**

- **Responsibilities:**
    - Stores and manages user profiles ("pending", "approved", "rejected", historical versions).
    - Handles the user profile approval workflow (accept, edit, reject).
        - "Accept": Marks pending profile as approved, making it the formal reference.
        - "Reject": Discards the pending profile.
        - "Edit": Triggers regeneration by allowing users to:
            - Change assets and their metadata (handled by `asset-service`, which then triggers `profile-generation-service`).

        - Add English instructions (passed to `profile-generation-service`).
    - Makes approved profiles available for public galleries and searches.
- **Data Store:** NoSQL Document DB (e.g., MongoDB, Elasticsearch) is ideal for storing profiles (which are largely text).
- **Key APIs:**
    - `POST /users/{user_id}/profiles/draft` (internal, called by `profile-generation-service`)
    - `GET /users/{user_id}/profiles/draft` (for user review)
    - `POST /users/{user_id}/profiles/draft/approve`
    - `POST /users/{user_id}/profiles/draft/reject`
    - `POST /users/{user_id}/profiles/draft/regenerate` (with optional instructions)
    - `GET /users/{user_id}/profiles/approved` (for internal use by search, gallery)
    - `GET /profiles/gallery` (paginated list of approved profiles for public view)
- **Interactions:**
    - Receives draft profiles from `profile-generation-service`.
    - Interacts with `notification-service` to inform users of drafts ready for review.
    - If "edit" involves changing assets, the flow goes back to `asset-service`. If "edit" involves adding instructions, it passes them to `profile-generation-service`.
    - Provides approved profiles to `search-service`.

**7. Industry Context Service (`industry-context-service`)**

- **Responsibilities:**
    - Manages the library of supporting reference information (government regulations, industry test protocols, crop statistics, etc.).
    - Allows administrators to upload, update, and curate these documents.
    - Prepares this context for RAG (indexing, embedding) for use by the `search-service` and potentially `metadata-extraction-service`.
- **Data Store:**
    - Object Storage for the documents.
    - NoSQL or Relational DB for metadata about documents.
    - Vector Database (e.g., Pinecone, Weaviate, FAISS index) for embeddings if RAG is implemented here directly.

- **Key APIs:**
  - `POST /admin/industry-context/documents` (upload)
  - `GET /industry-context/documents`
  - `GET /industry-context/documents/{doc_id}`
  - `GET /industry-context/search` (internal, for RAG by other services)
- **Interactions:**
  - Used by `admin-service` UI for management.
  - Provides context data to `search-service`.
  - Potentially interacts with `metadata-extraction-service` to find useful descriptors within this context.

**8. Search & Discovery Service (`search-service`)**

- **Responsibilities:**
  - Provides the "lossy" search tool using LLM + RAG.
  - Allows searchers (e.g., buyers) to ask general and interesting questions.
  - Combines information from approved user profiles (`profile-management-service`) and the general industry library (`industry-context-service`).
  - NEVER commingles information from two registered users in a way that attributes one user's data to another. It can *compare* or *list* multiple users.
  - Uses carefully engineered prompt templates for various search scenarios.
  - Implements metadata-based filtering in conjunction with LLM search.
- **Data Store:** May maintain its own indexed/embedded versions of approved profiles and industry context for efficient RAG, or rely on other services to provide this. A Vector Database is highly likely here.
- **Key APIs:**
  - `POST /search` (takes natural language query, filters)
- **Interactions:**
  - Fetches approved profiles from `profile-management-service`.
  - Fetches industry context from `industry-context-service`.
  - Calls `LLM Orchestration Service` with RAG setup (profiles + context) and search prompts.
  - Subscribes to `ProfileApproved` event to update its search index.
  - Subscribes to `IndustryContextUpdated` event.

**9. LLM Orchestration Service (`llm-orchestration-service`)**

- **Responsibilities:**
  - Acts as a central gateway for all LLM interactions (translation, metadata extraction, profile generation, search queries).
  - Abstracts the actual LLM provider (OpenAI, Google, HuggingFace model, etc.).

    - Manages API keys and specific model endpoint configurations.
    - Handles prompt templating and injection of context (for RAG).
    - Allows easy swapping/testing of new LLMs for each function by changing configuration or routing logic.
    - Could implement basic caching for identical LLM requests (if applicable).
    - Manages rate limiting or retries to LLM APIs if needed.
- **Data Store:** Configuration data (LLM endpoints, API keys, prompt template versions).
- **Key APIs (Internal):**
    - `POST /llm/translate`
    - `POST /llm/extract-metadata`
    - `POST /llm/generate-profile-rag`
    - `POST /llm/search-rag`
- **Interactions:**
    - Called by `translation-service`, `metadata-extraction-service`, `profile-generation-service`, `search-service`.
    - Retrieves prompts from `admin-service` (Prompt Library).

**10. Admin Service (`admin-service`)**

- **Responsibilities:**
    - Provides a UI for administrators.
    - Manages the "Prompt Library" – storing, versioning, and testing prompts for all LLM functions.
    - Manages LLM configurations (which model for which task, via `llm-orchestration-service` config).
    - User management (view users, suspend accounts – though most user data is in `user-auth-service`).
    - Management of `industry-context-service` content.
    - Viewing system logs and basic monitoring.
- **Data Store:** Its own DB for admin-specific data (e.g., prompt library).
- **Key APIs:** (Primarily for its own frontend, but could expose some for system-level tasks)
    - `GET /admin/prompts`, `POST /admin/prompts`
    - `GET /admin/llm-configs`, `PUT /admin/llm-configs`
    - APIs to interact with other services (e.g., trigger re-indexing in `industry-context-service`).
- **Interactions:**
    - Reads/writes prompts, potentially stored in its DB or a dedicated config store accessible by `llm-orchestration-service`.
    - Interacts with various services for management tasks.

**11. Notification Service (`notification-service`)**

- **Responsibilities:**

    - Sends notifications to users (e.g., "Your profile draft is ready for review," "New asset uploaded").
    - Supports multiple notification channels (email, in-app, potentially SMS later).
- **Data Store:** Minimal; possibly logs of sent notifications.
- **Key APIs (Internal):**
    - `POST /notify`
- **Interactions:**
    - Subscribes to relevant events from other services (e.g., `ProfileDraftReady` from `profile-management-service`, `AssetUploaded` from `asset-service`).

**12. API Gateway (`api-gateway`)**

- **Responsibilities:**
    - Single entry point for all client requests (web app, mobile app).
    - Request routing to appropriate backend services.
    - Authentication (often by offloading to `user-auth-service` or validating tokens issued by it).
    - Rate limiting, CORS handling, SSL termination.
    - Basic request/response transformation if needed.
- **Technology:** Nginx, Spring Cloud Gateway, AWS API Gateway, Kong, etc.
- **Interactions:** Routes requests to all other user-facing or internally accessible services.

**Supporting Infrastructure & Cross-Cutting Concerns:**

- **Message Bus:** (RabbitMQ) for asynchronous communication between services. This is vital for decoupling and resilience.
    - *Events examples:* `UserRegistered`, `AssetUploaded`, `AssetTranslated`, `MetadataExtracted`, `AssetProcessingComplete`, `ProfileDraftReady`, `ProfileApproved`, `IndustryContextUpdated`, `UserProfileRegenerationRequested`.
- **Cloud Platform:** AWSe
- **Containerization & Orchestration:** Docker for packaging services, Kubernetes (EKS, GKE, AKS) for deployment, scaling, and management. This aids in cheap development hosting (Minikube/Kind locally) and scaling up.
- **Configuration Management:** Centralized configuration for services (e.g., Spring Cloud Config, HashiCorp Consul, Kubernetes ConfigMaps/Secrets).
- **Logging & Monitoring:** Centralized logging (ELK Stack, Grafana Loki, CloudWatch Logs) and metrics/tracing (Prometheus/Grafana, OpenTelemetry, CloudWatch Metrics/X-Ray).
- **CI/CD:** Jenkins, GitLab CI, GitHub Actions for automated building, testing, and deployment of microservices.

- **Localization/Translation for UI:** Frontend frameworks have built-in or library support (e.g., i18next). The `translation-service` handles backend data translation.

# Key Use Case Flows with Microservices:

**A. User Uploads Asset & Profile Generation:**

1. **Frontend (Mobile/Web)** -> **API Gateway** -> **User-Auth-Service** (Authentication).
2. **Frontend** -> **API Gateway** -> **Asset-Service**: Uploads file.
    - `asset-service` stores file in S3, creates metadata record in its DB.
    - `asset-service` publishes `AssetUploaded(asset_id, user_id, asset_type)` event to Message Bus.
3. **Translation-Service** (subscribes to `AssetUploaded`):
    - If asset is text-based & not English:
        - Fetches asset from `asset-service`.
        - Calls `LLM-Orchestration-Service` -> (External LLM API) for translation.
        - Updates asset in `asset-service` (stores translated text, updates metadata).
        - Publishes `AssetTranslated(asset_id)` event.
4. **Metadata-Extraction-Service** (subscribes to `AssetUploaded` and/or `AssetTranslated`):
    - Fetches asset from `asset-service`.
    - Calls `LLM-Orchestration-Service` -> (External LLM API) for EXIF/descriptive metadata.
    - Updates asset metadata in `asset-service`.
    - Publishes `MetadataExtracted(asset_id)` event.
5. *(Optional Orchestrator or `asset-service` itself)*: Once all initial processing on an asset is done (e.g., translation, basic metadata), publishes `AssetReadyForProfiling(asset_id, user_id)` event.
6. **Profile-Generation-Service** (subscribes to `AssetReadyForProfiling` or a periodic trigger if new assets are found for a user):
    - Fetches all relevant assets for `user_id` from `asset-service`.
    - Retrieves system prompt (from `admin-service` via `llm-orchestration-service` or its own config).
    - Calls `LLM-Orchestration-Service` (with RAG data and prompt) -> (External LLM API) to generate profile.
    - Sends draft profile to `Profile-Management-Service`.
7. **Profile-Management-Service**:
    - Stores draft profile with "pending" status.

    - Publishes `ProfileDraftReady(user_id, draft_profile_id)` event.
8. **Notification-Service** (subscribes to `ProfileDraftReady`):
    - Sends notification to user (e.g., email: "Your profile draft is ready!").

**B. User Approves Profile:**

1. **Frontend** -> **API Gateway** -> **Profile-Management-Service**: User views draft.
2. **Frontend** -> **API Gateway** -> **Profile-Management-Service**: User clicks "Approve".
    - `profile-management-service` updates profile status to "approved".
    - Publishes `ProfileApproved(user_id, profile_id)` event.
3. **Search-Service** (subscribes to `ProfileApproved`): Updates its search indexes/embeddings for the new approved profile.

**C. Buyer Searches for Sellers:**

1. **Frontend** -> **API Gateway** -> **Search-Service**: Submits search query (natural language + any filters).
2. **Search-Service**:
    - Retrieves relevant approved profiles (from `profile-management-service` or its own cache/index).
    - Retrieves relevant industry context (from `industry-context-service` or its own cache/index).
    - Retrieves appropriate search prompt template (from `admin-service` via `llm-orchestration-service` or its own config).
    - Calls `LLM-Orchestration-Service` (with RAG data from profiles/context and search prompt) -> (External LLM API).
    - Processes LLM response, formats results.
    - Returns results to **API Gateway** -> **Frontend**.

# Addressing High-Risk Items:

- **Risk 1: Simple way for buyers/sellers to establish presence/profile (LLM extracts from uploads).**
    - This is covered by the flow: `asset-service` -> `translation-service` (optional) -> `metadata-extraction-service` -> `profile-generation-service`. The `llm-orchestration-service` is key to making the LLM part work and be swappable.
- **Risk 4: "Lossy" search tool (LLM+RAG, user info + general library, no cross-user data confusion).**
    - This is the responsibility of the `search-service`, leveraging `profile-management-service` for approved/isolated user profiles, `industry-context-service` for general info, and

`llm-orchestration-service` for the complex RAG query. The isolation of user data is paramount in how `profile-management-service` provides data and how `search-service` constructs its RAG context for the LLM.

## Next Steps & Considerations:

1. **POC for High-Risk Services:** Focus initial development on:
    - `asset-service` (basic upload and storage).
    - `llm-orchestration-service` (basic abstraction).
    - `profile-generation-service` (core RAG logic for profiles).
    - `search-service` (core RAG logic for search). This aligns with your "Build Critical Admin Functions Early" and tackling high-risk items. The "Prompt Library tool" within the `admin-service` (interfacing with `llm-orchestration-service`) would be part of this.
2. **Define APIs & Event Schemas:** Detail the request/response payloads for APIs and the structure of messages on the event bus. OpenAPI/AsyncAPI specifications are useful.
3. **Database Choices:** Select specific databases for each service based on data structure and query patterns. (e.g., Vector DB for RAG indices in Search and Profile Generation).