ChatGPT

MDM Platform: Technical Design and Implementation Guide

This document details the end-to-end design of **MDM**, a modern SaaS AI-driven data management and Auto-BI platform. It merges the architecture and requirements from the provided project report with an updated UI/UX inspired by ChatGPT, Power BI, and Notion.

1. Page-to-System Mapping

Each user-facing page corresponds to a backend module or service, as defined in the project report's architecture. Key mappings include:

- **Data Upload Screen → Data Ingestion Module:** Users can upload CSV/Excel/JSON files or connect to external data sources. On upload, files are stored (e.g. in Google Cloud Storage) and a *Data Ingestion* service validates and formats the raw data [1] [2].
- **Data Integration Form → Connectivity Module:** A form to configure connections to databases or APIs. This triggers connector services that fetch and ingest external data in the ingestion pipeline [3].
- **Data Preview Page → Data Cleaning/Validation Module:** After ingestion, the platform displays a preview of the dataset (columns, sample rows, detected types). Users see quality reports and can correct schema or missing values. This interfaces with the *Data Cleaning* service (e.g. Great Expectations checks and preprocessing routines) [1].
- **Chat With Data Page → AI Chatbot Module:** This is an interactive chat UI (inspired by ChatGPT) where users ask questions or give instructions about the data. It connects to an *AI Chatbot* backend that uses the processed dataset to answer queries and execute commands (e.g. "Show me sales trends by region") [4] [5].
- **Dashboard Workspace → Reporting & Visualization Module:** A workspace listing projects and dashboards. Users can view, share, and manage dashboards. Behind this, a *Dashboard Service* retrieves computed insights and metrics (from the AI analysis) and renders them via charts and tables [4].
- **Dashboard Builder Page → Visualization/Builder Module:** An editor where users can drag-and-drop charts, tables, and filters onto a canvas to build custom dashboards. This maps to a *Visualization Engine* that dynamically renders charts (using libraries like Recharts/D3) and saves configurations to the backend.
- **Collaborative Workspace (Comments/Approvals) → Collaboration Module:** A team area for comments, approvals, and audit trails on datasets and reports. This engages the *Collaboration* backend, which stores comments and tracks status of review tasks [6].

These mappings ensure each UI component feeds the corresponding backend service: for example, the **Data Upload** page triggers the ingestion pipeline, and the **Chat** page interacts with the AI query service. The project report's flow (steps 1–8) aligns with these pages: upload (step 1) leads to ingestion; cleaning/preprocessing (steps 2–3) feed the preview page; feature extraction and AI analysis (steps 4–5) power dashboards and the chatbot; and dashboard/report generation (steps 6–7) appear in the workspace [1] [4].

2. Front-End Design

Technologies and Frameworks

The front end is built as a **Next.js** (React) application with **TypeScript** for type safety and **Tailwind CSS** for styling, as specified in the requirements [7]. This stack supports responsive, component-based UI development. We use React's component model to create modular pages (e.g. `<DataPreview>`, `<ChatWindow>`, `<DashboardBuilder>`). Tailwind's utility classes ensure a consistent design system and make it easy to adapt layouts for desktop and mobile. [7]

Key libraries include:

- **Charting:** Recharts or D3.js for rendering interactive charts and graphs [7].
- **Data Grids:** A React data grid (e.g. AG Grid or TanStack Table) to display tabular data previews.
- **Drag-and-Drop:** A layout library like **Gridstack.js** or React DnD to enable draggable, resizable dashboard widgets. For example, Gridstack.js is a TypeScript library designed for building draggable, responsive layouts, with full mobile support and easy grid-based layouts [8] [9].
- **Chat UI:** Custom components for a chat interface, possibly extending from libraries (or built from scratch) with features like streaming messages and markdown rendering [10].

Security-related libraries (SSL support) and authentication (OAuth2/OpenID Connect client) will also be integrated, but are primarily handled in API calls.

Key Page Layouts

- **Landing Page / Dashboard Home:** On login, users see a dashboard home or workspace (à la Notion/PowerBI). This page shows a list of their projects or data workspaces, and recent dashboards. The layout can mimic Notion's clean sidebar navigation (with workspace folders) and a main pane listing dashboards. The design emphasizes discoverability: e.g. cards or tables for dashboards, a prominent "New Upload" button, and notifications of pending review tasks. Tailwind utility classes ensure this layout is responsive and clean.

- **Data Upload Page:** Users can drag-and-drop files or click to open a file dialog. The page shows upload status, basic file metadata, and prompts for naming the dataset. Below, a collapsible **Data Preview** section appears once the file is ingested, showing a sample table of rows and detected columns. The preview is styled like a spreadsheet (light gray grid lines, sticky headers) and includes inline editing for column names or types. Buttons allow moving to "Clean Data" or "Approve" actions.

- **Data Preview/Cleaning Page:** This may be a dedicated tab or modal after upload. It contains:

- A **Data Grid** showing rows (with infinite scroll or pagination).
- Column-level controls to change data types or apply transforms.
- Validation highlights (e.g. missing value counts or format errors).
- A side panel with summary statistics (mean, unique count, etc.) for the selected column.

- Buttons to accept the schema or send back to ingestion.

- **Chat With Data Page:** A full-page chat interface inspired by ChatGPT's UI [10]. It has:

- A scrollable chat log area where user messages and AI responses appear as chat bubbles.

- An input box at the bottom for the user to type queries.
- Streaming behavior: as the AI generates an answer, text appears in real time (using Server-Sent Events or WebSockets) [10].
- Markdown support: responses may include formatted text, lists, or tables.
- A sidebar or header showing the current context (dataset name, last updated).

- Visual cues (e.g. spinner) while the AI is generating.

- **Dashboard Builder Page:** A rich canvas where users construct dashboards. Layout:

- **Sidebar (left):** Contains available "widgets" (chart types, tables, text blocks). Users drag a widget onto the canvas.
- **Canvas Area:** Initially empty grid. Users drop widgets to add them. Each widget is resizable (with handles) and movable (drag) thanks to a grid-layout library [8] [9].
- **Widget Toolbar:** Each widget has a toolbar (on hover) for settings (edit data source, switch chart type, delete). Clicking a widget loads its configuration in the right panel.
- **Config Panel (right):** When a widget is selected, the panel shows data-binding controls (choose fields for X/Y axes, filters, etc.) and style options (colors, labels).

Dynamic chart rendering is implemented using React: when the user changes data or type, the chart re-renders immediately using the selected library (e.g. a Recharts `<LineChart>` or `<BarChart>` component). Tooltips and animations are enabled by default for interactivity. Saving the dashboard records the widget positions and queries to the backend (Postgres).

Interactive Behaviors

- **Chatbot Interaction:** The chat interface behaves like modern AI assistants. Upon sending a message, the frontend calls the chat API endpoint (FastAPI) and receives a streamed response. Messages are appended to state as they arrive, and the view auto-scrolls to show new text [10]. Users can ask for charts or data transformations; the UI may render images or charts if the AI response contains them. The input supports pressing Enter to send, history of previous chats, and possibly quick-reply suggestions.

- **Dynamic Chart Rendering:** Charts on the dashboard update live as data or filters change. For example, if the user switches the X-axis variable in a chart's settings, the component re-fetches data via an API and animates to the new view. Hovering shows tooltips; clicking chart elements can drill down or bring up details. Tailwind CSS ensures charts and containers are responsive (auto-sizing to the layout grid).

- **Drag-and-Drop Dashboard:** Implemented via a library like Gridstack.js or React Grid Layout, the dashboard canvas supports intuitive dragging. Users click-and-drag a widget to reposition it; drag-corners to resize. The grid snaps widgets to a 12-column layout by default. The sidebar's widgets palette can be filtered (e.g. search "bar", "pivot table") and dragged onto the canvas. A "save" button locks the layout, and an "undo" history may allow reverting changes. These interactions are animated and touch-friendly (Gridstack has full mobile support) [8] [9].

3. Back-End Architecture

The back end consists of a suite of microservices (implemented in Python/FastAPI) orchestrated via message queues and databases. The design reflects the stack in the SRS: FastAPI for HTTP APIs, RabbitMQ for asynchronous tasks, and a mix of MongoDB and PostgreSQL for storage [11].

- **API Gateway / Auth Service:** All external requests go through a FastAPI gateway that handles authentication (OAuth2/OpenID Connect) and routes to internal services. The gateway validates JWTs (from an identity provider) and enforces RBAC (roles: Admin, Data Engineer, Analyst, Viewer) as defined [12]. Endpoints include login, data upload, query chat, and dashboard CRUD. HTTPS/TLS is enforced at the load balancer (TLS 1.3) [13].

- **Data Ingestion Service:** A FastAPI endpoint receives file uploads or connector configs. Uploaded files are stored in **Google Cloud Storage (GCS)** for persistence [14]. The service publishes a message to RabbitMQ to start processing. It may also spawn a container (via Kubernetes) to run heavy ingestion if needed. For connectors (e.g. JDBC, SaaS APIs), this service pulls data and writes to GCS or streams to the next step.

- **Data Cleaning & Preprocessing Module:** Worker services consume RabbitMQ messages for new data. Each worker runs a Python pipeline: using **Pandas** and **Great Expectations** [15] to validate and clean the data (impute missing values, correct types, encode categorical variables, normalize numeric ranges). Metadata (schema, stats) is stored in PostgreSQL via SQLAlchemy [16], and semi-structured output (e.g. JSON logs of data issues) is saved in MongoDB [14]. Any detected schema changes (new/removed columns) are noted for review.

- **Feature Extraction / Data Transformation:** Optionally, after cleaning, additional feature-engineering tasks run (e.g. deriving date parts, one-hot encoding). This is another RabbitMQ-driven step. The processed tabular data (or embeddings for text columns) are persisted (perhaps in MongoDB or a columnar store).

- **AI Analysis / Insight Generation:** Once data is prepped, an **Analytics Service** kicks in. This might include statistical analysis (correlations, trends) or machine learning (clustering, forecasting). For example, it could train a regression model to predict a target or use NLP to summarize text fields. Extracted insights (key metrics, recommended charts) are saved. The service can use libraries like scikit-learn or call external AI APIs (e.g. GPT) for natural-language insight generation [5]. The output (insights, charts data) is stored in MongoDB and Postgres (for structured results).

- **Chatbot Service:** A specialized FastAPI chatbot backend responds to natural-language queries. Internally, it uses a retrieval-augmented approach: when a user asks a question, the service retrieves relevant data or reports from the processed dataset (using indexes in MongoDB/Postgres) and optionally invokes an LLM (e.g. GPT) with this context to formulate an answer. The pipeline ensures near-constant response time by caching embeddings or pre-indexing (cf. AI chatbots in modern analytics platforms) [5].

- **Dashboard/Report Service:** This service handles rendering requests. When the front end loads a dashboard, it queries this service, which fetches the latest insights/metrics from the databases and returns JSON for chart components. It also generates downloadable reports (CSV, PDF) on demand. Data storage used: PostgreSQL holds relational metadata (users, permissions, dashboard definitions) and MongoDB holds time-series or semi-structured insight data [14].

- **Asynchronous Messaging (RabbitMQ):** Nearly all heavy or multi-step workflows use RabbitMQ queues. For example, upload triggers a "cleaning" queue; after cleaning, a "feature extraction" queue; then an "analysis" queue. This decouples services and enables horizontal scaling (workers can be scaled in Kubernetes pods). RabbitMQ ensures reliable, ordered processing of tasks [15].

- **Databases and Storage:**

- **PostgreSQL (via SQLAlchemy)** stores core relational data: user accounts, permissions, dataset metadata, dashboard configurations, and approved schemas [16] [12].
- **MongoDB** stores semi-structured data: cleaned datasets (if large, potentially as document collections or time-series), AI-generated insights, and chat logs. Its flexible schema is ideal for evolving analytic results.

- **Google Cloud Storage** holds raw and intermediate data files and backups [14]. Encrypted at rest (AES-256) as per requirements [17].

- **Infrastructure:** All services are containerized (Docker) and deployed on Kubernetes (e.g. GKE) for orchestration [18]. Horizontal Pod Autoscaling is configured to meet performance targets (see next section). CI/CD pipelines automate testing and deployment.

4. Data Flow Integration

The end-to-end flow of a dataset through MDM is as follows:

1. **Upload & Ingestion:** The user uploads a file via the front-end UI. The file is POSTed to the Ingestion API, which saves it to GCS. A RabbitMQ message ("NewData" event) is published.
2. **Validation & Cleaning:** A worker consumes the "NewData" message and runs the cleaning pipeline: validating formats, imputing missing values, encoding categories, and normalizing columns [1]. The pipeline logs any anomalies. Upon completion, the cleaned data is written to MongoDB or staged in PostgreSQL.
3. **Schema Review (HITL):** If new columns or type changes were detected, the system flags the dataset for review (see Section 6).
4. **Feature Extraction & Analysis:** Optionally, feature engineering tasks run next (e.g. parsing dates or generating text embeddings). Then an AI analysis service processes the data to compute insights: for example, statistical summaries, predictive models, or key-factor identification [5]. These insights (e.g. "Sales grew 15% YoY", "Top customer segment identified") are stored and used to populate initial dashboards.
5. **Dashboard Generation:** The Visualization Service queries the new insights and populates default charts (e.g. time series, distributions). The front-end then renders these as initial dashboards (viewable in the Dashboard Workspace). Users see charts and KPIs computed from the analysis.
6. **Chatbot Interaction:** Meanwhile, the Chatbot Service is aware of the new dataset. When the user types a question (e.g. "What was our highest sales region?"), the chat backend retrieves relevant data slices (using indexed database queries) and possibly supplements with a large language model. The answer is returned as a chat message. This tight integration means the chatbot uses the *same cleaned data and analysis results* produced in earlier steps.
7. **Export & Reporting:** Users can export the processed data or generate a PDF/CSV report of insights. These actions pull the final data from storage and format it.

Throughout this pipeline, RabbitMQ ensures asynchronous coordination and the FastAPI services expose necessary endpoints. Notably, each step can run independently, allowing the system to handle

large data asynchronously without blocking the user. This design aligns with the report's flow: steps 1–3 (upload, ingest, preprocess), step 4 (feature extraction), step 5 (AI analysis), step 6 (chat) and steps 7–8 (dashboards, export) [1] [4] .

5. Security, Performance, and Scalability

The system adheres to strict non-functional requirements from the report:

- **Security:** All communications use HTTPS/TLS 1.3 [13] . User credentials are never stored in plaintext; passwords are hashed and never kept in cookies or local storage [19] . Data at rest (in databases, storage) is encrypted with AES-256 [20] . Authentication uses OAuth2/OpenID Connect (e.g. via a third-party provider) with JWT tokens. Role-Based Access Control (RBAC) enforces permissions per resource (roles: Admin, Data Engineer, Analyst, Viewer) [12] . Multi-factor authentication is enabled for sensitive operations [12] . Sessions auto-expire after inactivity, and the API gates endpoints so that only authenticated/authorized calls succeed. Administrative interfaces are isolated and require elevated credentials. Audit trails log all changes (uploads, schema edits, approvals) for compliance.

- **Performance:** The frontend and API are optimized for responsiveness: UI pages (dashboards, data preview) must load within ~2 seconds, and API calls (data queries, chatbot replies) target under 500ms for normal workloads [21] . To meet this, the database queries use indexing, and a caching layer (Redis or in-memory) may store frequent results. Long-running jobs (data cleaning, analysis) run offline via RabbitMQ to avoid blocking. We aim to process at least 10,000 records/sec in batch pipelines [21] .

- **Scalability:** The architecture supports horizontal scaling. Kubernetes autoscaling (HPA) can add more worker pods when CPU/memory use is high. The system is designed to handle 100+ million records and 500+ simultaneous users without degradation [21] . It can auto-scale up to 10× traffic spikes (e.g. during working hours) by adding instances of FastAPI services and workers [22] . Databases are either clustered (MongoDB replica sets, PostgreSQL read replicas) or use managed services to scale storage. Static content (frontend assets) is served via a CDN.

- **Deployment:** The system is cloud-agnostic. It can run on GCP, AWS, or Azure using Docker/Kubernetes [23] . CI/CD pipelines with automated tests ensure rapid and reliable updates.

6. Human-in-the-Loop Workflow

To ensure accuracy and user trust, MDM incorporates humans in key decision points:

- **Schema and Data Validation:** After ingestion, the system may detect schema changes (new/renamed columns) or data quality issues. The **Data Preview UI** shows these to the user. For example, if a new column "Region" appears, the UI pops up a "Review Schema Changes" dialog (similar to Airbyte's workflow [24]). The user can examine the detected changes (shown in a comparison view) and click **Approve** or **Edit**. Approved changes update the stored schema; otherwise, the user can rename columns or adjust types before proceeding. This "Review changes" dialog mimics the Airbyte approach:
- Click "Review Changes" (triggered by a notification or icon).
- A modal lists detected differences (added/removed fields).
- The user reviews and clicks "OK" to accept each change [24] .

- Finally “Save” applies them for the next processing run.

- **Insight and Chart Suggestions:** Similarly, if the AI analysis suggests new insights or default charts, the **Dashboard Workspace** highlights them for user review. For example, the system might auto-generate a chart showing “Top 5 product categories by revenue.” The UI can show an “Edit Chart” panel where the user can rename metrics, change filters, or reject the suggestion. All such AI-proposed items carry an “edited by AI” flag until confirmed by a human.

- **Chat Corrections:** If the chatbot gives an unsatisfactory answer, the user can provide feedback (e.g. thumbs-down or comment), which feeds back to the model or triggers a retraining process. The chat history is saved for audit.

- **Collaborative Review:** Within a **Collaborative Workspace** (Project area), team members can comment on datasets and dashboards [6]. For example, an analyst may upload a dataset and request approval from a manager. The manager receives a notification, reviews the automated insights via the web UI, and leaves comments (e.g. “Verify the sales data aligns with ERP export”). They can then **Approve** or send back for revision. The system logs all comments and decisions (audit trail). This mirrors the specification: “supports comments, approvals, and audit trails within shared projects” [6].

- **Annotation and Training:** For use cases involving machine learning (e.g. image or text labels), a dedicated UI presents samples and predicted labels for human correction. Each approved annotation is stored to improve future models. (While not in the original SRS, this follows the “Human-in-the-loop improves quality” principle.)

In summary, at each critical AI decision point (schema changes, data cleansing rules, insight conclusions), the UI provides an **Approve/Edit** workflow. This ensures end users remain in control and the AI augments rather than replaces human judgment (reflecting the report’s emphasis on accuracy and collaboration [5] [25]).

Sources: Architecture and requirements from the provided report [26] [1] [4] [3] [27] [19] [12]; frontend design patterns from React/Tailwind and chat UI literature [10] [8] [9]; schema-review process (Airbyte docs) [24].

---

[1] [2] [3] [4] [5] [6] [7] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [25] [26] [27]
Project_stage_I_A4__2025_ (1) (1).pdf
file://file_00000000bc0871fa816d681aaa79b467

[8] [9] Gridstack.js | Build interactive dashboards in minutes.
https://gridstackjs.com/

[10] Creating a React Frontend for an AI Chatbot | Medium
https://medium.com/@codeawake/ai-chatbot-frontend-1823b9c78521

[24] Schema Change Management | Airbyte Docs
https://docs.airbyte.com/platform/1.7/using-airbyte/schema-change-management