System Architecture Overview

Thank you for inviting me for this very interesting task. This proposal is for the MVP, adnd potential expansion is introduced in “Technological Evolution”. The core part of the system is composed of 4 layers: data ingestion, data curation, AI agent and frontend.

Data ingestion

The data ingestion is realized in batch mode because a reasonable delay, e.g. 15 minutes to 1 hour is acceptable. Dagster schedules a controlled batch extraction from the Zendesk Incremental Export API using alternating mode: the new data collected in the last period of time is prioritized, and after the ingestion of the new data, the historical data ingestion is triggered with the order from the latest to the oldest. Retrieved records are written to S3/MinIO as immutable append-only files, and the cleaned data are written into the ClickHouse using a deterministic idempotence key composed of ticket_id + updated_at.

Data curation

This layer handles three main tasks: data model building and pre-aggregation for standard business metrics in ClickHouse, traceable cold data archiving to S3 / MinIO compatible object storage (Parquet), and vectorization of comments for similarity search.

This transformation process is implemented in ClickHouse with dbt. The modeling is organized into three layers. The staging layer data undergoes light cleaning operations, extracting event table, and establishment of the comment table. The core layer is responsible for constructing fact tables and dimension tables from the ticket table. The marts layer includes wide tables and aggregated datasets that are tailored to specific business metrics, enabling efficient and targeted data exploration.

The comments are classified with a lightweight sentiment model to find out the complains and rule-based keywords searching by concerned products and processes.

Cube exposes these models to external consumers through high-performance APIs. The data cleaning, modeling and verification processes are integrated into Git workflows to realize CI/CD.

AI Agent

Classification and vectorization of comments: a dedicated AI agent complements the comment table by identifying missing products or processes. To reduce the overhead cost of the repeated system prompt, classification is done in batch mode. Another AI agent is used to extract information apart from the concerned products and processes

so that the vectorization focusses on the remaining unclassifiable information. The vector database is ClickHouse because of the possibility to combine analytics with vector search.

The AI agents are managed by **LangChain**. Rather than a free-form conversational agent, the system uses a structured, tool-driven orchestration framework based on modular sub-agents and explicit function execution policies. The orchestrator agent interprets user intent (metrics, insights, retrieval, saved reports), decides which sub-agent to invoke and provides clarifying questions when the user's query lacks specificity.

The sub-agents are:

1. Metrics-SQL Agent: translates authorized queries into template-bound SQL.
2. Vector Retrieval Agent: performs vector search for semantic similarity with embedding search bounded by filters.
3. Chart-Builder Agent: generates Vega-Lite chart specs with template-locked chart schema.
4. Saved-Report Agent: loads, runs, versions, and schedules reports.
5. Summarization Agent: produces grounded summaries from SQL and vector results.

Frontend

The frontend is a Next.js Web App that exposes backend capabilities such as the Query Router, Chart Service, Saved Analyses, and Vector Retrieval. It provides users with a conversational interface powered by a chat assistant that supports natural language queries. The chat interface is designed as a streaming UI, meaning responses are rendered progressively as they are received from the backend, creating a smooth and responsive experience.

Session history is supported until the token size limit. Each chat session is stored with its full context, allowing users to search, trace, rerun, and share sessions. When a user requests to save an analysis, the backend stores the query logic, chart specification, and the result snapshot. This saved analysis is then accessible in the report library, where users can view, rerun, and manage versions. The system allows users to toggle parameters such as time span—for example, switching between Q2 and Q3—to dynamically update the analysis results. These analyses are stored in ClickHouse for fast analytical queries. dbt is used to build semantic models that structure the data for querying.

Data Flow Description

Dagster download the raw data via Zendesk's Incremental Exports API, a python code is used to execute basic data cleaning process for example remove useless fields (metadata, links etc), formats time, and adds helper columns (such as sync_time, last_updated), which ensures idempotence leveraging the ticket_id + updated_at.

To guarantee data completeness and ensure no tickets are missed due to webhook delivery failures, Dagster compares the exported tickets against S3 / MinIO and ClickHouse on the idempotence key to identify any missing or late-arriving records. Only records absent from ClickHouse are appended.

The cleaned and added data in ClickHouse are transformed to ticket tables, event tables, comment tables. The pre-aggregated table is built using materialized views and stores commonly used KPIs such as daily ticket counts by status. This allows the frontend to query summary data directly, reducing latency and computational load.

In the frontend, when a chat request reaches the API, an orchestrator AI agent analyzes the text input and convey the task to the corresponding AI agent. If the request is a pure metric query, the Metrics-SQL Agent launches the required data model which is executed against ClickHouse. If the request is about reasons, feedback, or complaints, SQL is used to narrow down the data by time, priority, product, or process. Additionally, a vector search can be done to retrieve the top-k most similar comments, and the agent generates a summary with citations to the retrieved evidence. If the user requests a chart, Chart-Builder Agent calls the SQL result and pass it to a Vega-Lite rendering service which returns an image, and the result is cached for performance. Finally, the system supports saving and reuse. Users can save the current view, which includes semantic metrics, filters, and chart specifications, as a named report. These reports can be scheduled to run automatically by the saved-report agent.

Technology Choices

For data storage, S3 is a widely adopted cloud object service, highly durable and scalable. MinIO is lightweight, S3 compatible, can be deployed on-premises or on any cloud.

Dagster is chosen for this data orchestrator because it is fully pythonic, eliminating the need for complex yaml/operator code, and integration with dbt and S3 is seamless. The UI displays the dependencies between each table. If the scale expands in the future—for example, migration to Dagster Cloud (which already supports enterprise-level features) is possible.

dbt (Data Build Tool) is chosen for data model on SQL because it is lightweight, modular, and testable framework that integrates seamlessly with ClickHouse. dbt supports

version-controlled and reusable models and automatically manages dependencies and execution order. It includes built-in testing and documentation features, fits naturally into CI/CD workflows. Compared to tools like Apache Spark, Airflow, or Pandas, dbt is more efficient and maintainable for structured data workflows in cloud environments.

Reasons for choosing ClickHouse (compared to Snowflake/BigQuery) for online OLAP are: extremely fast aggregation and filtering of time series data due to is column-based storage and batch data processing using CPU friendly operations; possibility of self-host ClickHouse indicating lower total cost of ownership (TCO); comprehensive functionality: materialized views, pre-aggregation, automatic TTL expiration (data movement to cold storage or deletion) partitioning to speed up queries etc.

LangChain is chosen for orchestrating AI agents because it offers a robust, modular framework specifically designed for tool-based agent orchestration. Unlike alternatives like CrewAI or Autogen, which focus on autonomous agent collaboration or multi-turn conversations, LangChain excels at structured, deterministic workflows where each sub-agent performs a clearly defined task.

Next.js stands out over other frameworks because it combines the power of React with advanced features like server-side rendering (SSR), static site generation (SSG), and API routes, all in one unified framework. This makes it ideal for AI-enhanced chat experiences with fast and dynamic UIs. Compared to plain React (which lacks routing and SSR), or frameworks like Angular and Vue (which are more opinionated and less flexible for full-stack development), Next.js offers better performance and developer productivity. It also integrates smoothly with edge functions, vector databases, and LLM APIs, making it a top choice for modern AI applications.

Data Schema Structures

Fact Tables

- ticket_id: unique ticket identifier
- created_at: when the ticket was created
- updated_at: last update timestamp
- status: ticket status (e.g., new, open, pending, solved)
- priority: ticket priority (e.g., low, normal, high, urgent)
- assignee_id: ID of the assigned support agent
- requester_id: ID of the user who submitted the ticket
- closed_at: when the ticket was closed (nullable)

- sla_breach: 1 if SLA was breached, 0 otherwise

Comment tables :

- ticket_id: associated ticket ID
- comment_id: unique comment identifier
- body: raw comment text (may be anonymized)
- sentiment: -1 (negative), 0 (neutral), 1 (positive)
- products: list of products concerned
- processes : list of processes concerned
- lang: language code (e.g., en, fr)

mv_ticket_daily_counts — Aggregated ticket counts

- day: date of aggregation
- priority: ticket priority
- status: ticket status
- cnt: number of tickets for that day/priority/status

Vector Table

comment_embeddings — Semantic search data

- ticket_id: ticket ID
- comment_id: comment ID
- chunk_id: index of the text chunk
- content: text content of the chunk
- embedding: high dimensional vector
- created_at: timestamp of embedding creation

Primary key is a combination of ticket_id, comment_id, and chunk_id. An index is created on the embedding column using cosine similarity for fast vector search.

Metrics (not physical columns but derived metrics):

- tickets.count: total number of tickets (sliceable by day, priority, status, tag)
- tickets.oldest_unresolved: top N unresolved tickets sorted by creation time
- tickets.by_feature_mentions: tickets grouped by product features (from comment topics)

- comments.negative_share: percentage of comments with negative sentiment
- tickets.sla_breach_rate: ratio of tickets with SLA breach

API Contracts

POST /chat/query

Used for natural language queries over the data.

Request fields:

- text: user's natural language question
- user_id: ID of the user making the request
- org_id: organization ID
- session_id: session tracking ID
- idempotence_key :
- timeout_ms :
- budget : {max_tokens :, max_cost_usd:}

Response fields:

- answer_md: markdown-formatted answer
- sources: optional list of references, either SQL queries or comment IDs
- chart_url: optional link to a chart visualization
- latency_ms: time taken to process the query
- cache: {hit: true, freshness: 300}
- audit_id:

This endpoint connects to the ClickHouse facts, depending on whether the query is metric-based or semantic.

POST /reports/save

Used to save a report definition.

Request fields:

- name: name of the report
- metric_query: includes metric name, filters (e.g., priority = high), and time range
- viz_spec: visualization type and axes (e.g., line chart with x = day, y = sla_breach_rate)

- schedule: optional cron-style schedule for recurring reports

Response:

- report_id: unique ID for the saved report

POST /reports/run

Used to execute a saved report.

Request fields:

- report_id: ID of the report to run
- time_range: optional override of the default time range

Response fields:

- chart_url: link to the generated chart
- data_url: link to the raw data (e.g., CSV)
- started_at: timestamp when execution started
- finished_at: timestamp when execution completed

Detailed Technical Explanations

1. User Experience Flow

Brenda types her question into the chat interface, the orchestrator AI agent analyzes the question to decide which sub-agent to launch. The corresponding sub-agent retrieves relevant data model by executing SQL query, or to do similarity search in the vector database, or to do save a report, or to use Vega-Lite to generate a chart (like a bar or line chart) as SVG. The chart is rendered server-side and cached for reuse. At last, the summarized answer, along with table, the chart and source references, is returned to Brenda in the chat interface.

Key technical challenges:

Generating safe and correct SQL: the system only executes white-listed SQL query. Template injection must be avoided, all external variables are escaped, ensuring that user input cannot be directly embedded into SQL.

Queries require too much resource and induces too much delay: the system should alert the user if the query requires complex joins, limit the number of returned vectors, limit the tokens per session.

Ensuring explainability: showing the SQL query or linking to the source data helps build confidence and allows users to verify results.

Handling ambiguous or vague questions: if the question is unclear, the system may need to ask follow-up questions or offer filter suggestions to clarify the user's intent.

Supporting filter corrections, if the user sees an unexpected result, they should be able to quickly adjust filters like time range or priority without retyping the whole question.

Multi-turn dialogue: to identify the root cause in a process the AI agent needs to understand the question and comments from the user, suggest metrics or vector search to help to identify the problem. A carefully engineered prompt with multi-shot training or RAG can help to improve the analytical competence of the AI agent.

2. Historical Data Challenges

Data storage: all the tables are stored in ClickHouse using a columnar format, which is highly compressed and optimized for analytical queries. The data is partitioned by date to enable efficient filtering by time. This partitioning is critical for fast queries over recent or specific time windows.

Data cleaning: a lightweight rule-based cleaning is to be executed to extract structured data from raw tickets. The comment sentiment classification can be done with small models such as facebook/bart-large-mnli. Only the comments without identified products or processes are batch analyzed by AI agent. The vectorization is also limited to the part of comments without classifiable information such as product and process so that the processing time and space can be limited.

Response time: daily aggregates are precomputed by priority and status using materialized views. Default time window and alert on large time window to scan can be enforced to avoid full-table scans unless the user explicitly requests a broader range. For vector search the number of results retrieved (Top-K) can be capped to a reasonable number to balance relevance and performance. ClickHouse supports HNSW index which enhances the rapidity of vector similarity search. Each incoming request is assigned a unique idempotence key derived from its body content. This key is stored in Redis with a short expiration time (typically 5 to 15 minutes). If an identical request arrives within this window, the system retrieves the cached response from Redis and returns it directly, avoiding redundant processing. In case of excessive running time, the system falls back to coarse-grained reports. It is also possible to use ClickHouse's max_execution_time setting to abort long-running queries, or allow the user to specify it. All slow queries can be logged and used to improve materialized views or indexing strategies.

3. Business Data Risks

In ClickHouse, access control is implemented through row-level policies and column-level permissions. Sensitive fields, such as personal information in comments, are exposed only through masked views to prevent unauthorized access.

In S3 or MinIO, bucket policies are configured to allow access only to backend service roles, preventing frontend clients from directly accessing raw data. Object lock mechanisms are used to make raw snapshots immutable.

In dbt, data contracts are defined in schema files to specify field names, types, and validation rules. dbt validates model outputs against these definitions during runtime, preventing errors caused by upstream schema changes or type mismatches. Data models are cited in the response and logged for audit.

Cube is used to expose only a curated set of metrics. Dimension values are constrained through enumerations to prevent arbitrary or injected values. Cube supports API tokens, and allows access to pre-aggregated data only, ensuring that raw data is not exposed.

AI agents apply input filtering, tool invocation constraints, and output validation to prevent unauthorized access, data leakage, and misuse of system resources, while ensuring that LLMs operate within clearly defined boundaries.

Input filtering blocks prompts that attempt to execute non-whitelisted SQL queries, export raw personally identifiable information (PII), or access data from other organizations without permission. Sensitive patterns such as email addresses, phone numbers, and national ID numbers are detected using regular expressions or abstract syntax trees. Prompts that attempt to extract entire datasets or print database schemas are discouraged.

Each sub-agent has a single, well-defined capability. No agent is permitted to trigger arbitrary operations or access data outside its predefined scope. Sub-agents must return structured JSON that is validated against a schema. Only valid schemas trigger execution. SQL generation is not free-form, only white-listed query is permitted, and direct natural-language-to-SQL execution is prohibited.

Output filtering ensures that generated content does not expose sensitive information. Text outputs are scanned for PII using regular expressions and dictionaries. If potential PII is detected, the system either masks it or informs the user that they lack the necessary permissions. For charts and tables, the system checks that only approved fields are included.

In the backend architecture, the API gateway handles authentication, rate limiting, idempotence, and auditing. Authentication is implemented using the fastify-jwt plugin, which validates JWT tokens. Role-Based Access Control (RBAC) is enforced through a permission checker function that evaluates access rights based on user roles, organizational affiliation, and scopes. Rate limiting is managed using the fastify-rate-limit plugin. Limits can be applied per IP address or per user and should be configurable at the route level—especially for endpoints like /api/query and /api/chart/render, which are resource-intensive and prone to abuse.

Audit logs are written to a dedicated ClickHouse table. Each log entry should include fields such as timestamp, user ID, IP address, HTTP method, URL, status code, request body, SQL templates used and bound parameters, vector search parameters and Top-K results, cost metrics (tokens, execution time). This provides a comprehensive trace of API activity for monitoring, debugging, and compliance purposes.

4. Complex Business Questions

To answer complex questions like customer dissatisfaction the AI agent identify the products and processes, suggests the relevant metrics and runs the data model. Then within the scoped dataset, the system performs a Top-K vector search over comment embeddings to retrieve the most relevant user feedback. The retrieved metrics and comments are passed to an LLM for a summarization and a suggestion to find root cause and identify the plan of action. The comment table with concerned products and processes help the AI agent better retrieve the relevant information.

With all the agents, the system realizes a proactive analytical agent designed to trace meaningful business signals and assist users in navigating complex, multi-stage investigative workflows. This agent leverages a curated expert prompt corpus that encodes domain heuristics, historical operator patterns, and best practices for troubleshooting. This agent is able to propose existing metrics that are relevant to the user's question, suggest new composite or derived metrics when existing ones are insufficient, manage multi-turn diagnostic conversations, progressively narrowing root causes.

For example, if there is a problem with the final product at the exit of the assembly line or identified at the customer, the AI agent can help to identify which parts potentially dysfunction. With the preference and agreement of the user, the AI agent can search for tickets / logs relevant to a special part, and identify what deviations in product or in process has been applied to this part, for example a product modification or a process rework etc. This helps the identify the potential reasons quicker, even for relatively new employee on this project.

Another example would be the introduction of a design modification, the AI agent can suggest the impacted performances and processes according to the information saved in the tables or engineer designed prompt, which helps the user to better verify the impacts.

This proactive analytical agent helps also on the completion of the log system by identifying the missing critical information. With the help of engineered prompt or RAG into design documents, this agent can evolve to be a product and process mining agent and helps to improve the product and process in the company level.

5. Workflow Efficiency

To improve efficiency for recurring business tasks—like generating a weekly report every Monday—the system supports customized reports which defines the metric (e.g. ticket count), dimensions (e.g. priority, status), filters (e.g. queue, assignee), and time window (e.g. last week), and customized visualization specification which defines the visualization using a Vega-Lite JSON schema (e.g. bar chart, line chart).

The system also supports versioning. If a user edits a saved report (e.g. changes the metric or chart type), a new version (e.g. v2) is created, while the original (v1) is preserved and can be rolled back if needed.

For operations on browser, the AI agent will call the headless browser and help to click on the necessary buttons to automate with one request in the chat.

6. Data Visualization Needs

The challenges can be reproducibility, rapidity and cost.

Reproducibility is achieved by standardizing all visualizations using the Vega-Lite specification. This avoids inconsistencies caused by client-side rendering logic or ad hoc charting libraries. Every chart is generated from a structured viz_spec JSON, ensuring that the same input always produces the same output.

Another problem associated to the reproducibility is the injection prevention when LLMs are involved. To avoid security risks or malformed charts, the system does not allow LLMs to freely generate Vega-Lite JSON. Instead, chart generation is strictly based on a whitelisted schema that maps approved metrics and dimensions to predefined chart templates. This ensures that only safe, validated visualizations are rendered.

For the rapidity and cost, Server-Side Rendering (SSR) is used to generate charts as PNG or SVG images on the backend. This allows charts to be directly embedded in chat responses. Downsampling for Large Datasets is necessary when visualizing long time series or high-volume data. The system applies algorithms like Largest-Triangle-Three-Buckets (LTTB) to reduce the number of data points while preserving the visual shape of the trend. This keeps charts readable and performant without losing key insights.

Interactive Features are supported on the frontend when needed. If the user opens the chart in a browser or dashboard, the system can send the full Vega-Lite spec to the client, enabling features like hover tooltips, filtering, zooming, and brushing. This hybrid approach allows static previews in chat and rich interactivity in web apps.

7. Operational Cost Management

The system uses different budget controls to respond to the cost management when users ask questions with vastly different computational demands. If the query matches

a previously computed result, the system returns it instantly from cache. Token usage and sample size are limited to control cost. For complex, long-form analysis (e.g., “Why are customers unhappy with the new feature over the past year?”), the system prompts the user to confirm entry into a “fine-grained analysis mode.” This mode may involve large vector searches, long LLM completions, and higher cost. Each request, user, and organization has a defined budget cap for example maximum token usage per response, daily or monthly cost limits etc. If a request exceeds these limits, the system responds with a message like: “This query exceeds your current budget. Please narrow the scope or upgrade your plan.” to build the consciousness of the user on the cost.

Embedding models and LLMs are accessed through a unified interface. This allows the system to switch providers based on cost, latency, or quality. For example, use a cheaper embedding model for bulk vectorization, use a smaller LLM for short summaries and standard output, use a premium model only when high accuracy is required. This modular design ensures that the system can adapt to changing pricing or performance needs without rewriting core logic.

8. Technology Evolution

LangChain supports different LLMs with unified APIs, which allows the system to switch between providers (e.g., OpenAI, Anthropic, local models) by changing configuration, without modifying business logic.

Prompt templates, tool schemas, and report definitions are version-controlled. This allows gradual rollout of improvements (e.g., better phrasing, stricter filters) via A/B testing. Older versions remain available for rollback or audit, ensuring stability and traceability.

Natural language processing modules—such as sentiment analysis or topic classification—are designed to be interchangeable. For example, a zero-shot classifier can be replaced with a fine-tuned model without affecting upstream logic. This modularity enables experimentation and upgrades without disrupting the pipeline.

The new data ingestion can be realized with the streaming tool kafka in case of high concurrency. Kafka saves raw data to S3 or MinIO and kafka Steams app executes basic data cleaning process to ensure idempotence before writing to ClickHouse.

If vector search is not frequently combined with analytics, it is possible to migrate the vector data base to a dedicated tool suh as Qdrant or Weaviate. These systems are designed for high-performance vector search at large scale and offer native support for features like distributed indexing and optimized recall.