Project Deliverables

Part 1: System Design

Thought Process

- As I'm laying out the design I thought about a couple of things, the requirements & assumptions, and since the task is 60% architecture & 40% implementation I thought I can focus on high level architecture then a deeper dive into detailed architecture.

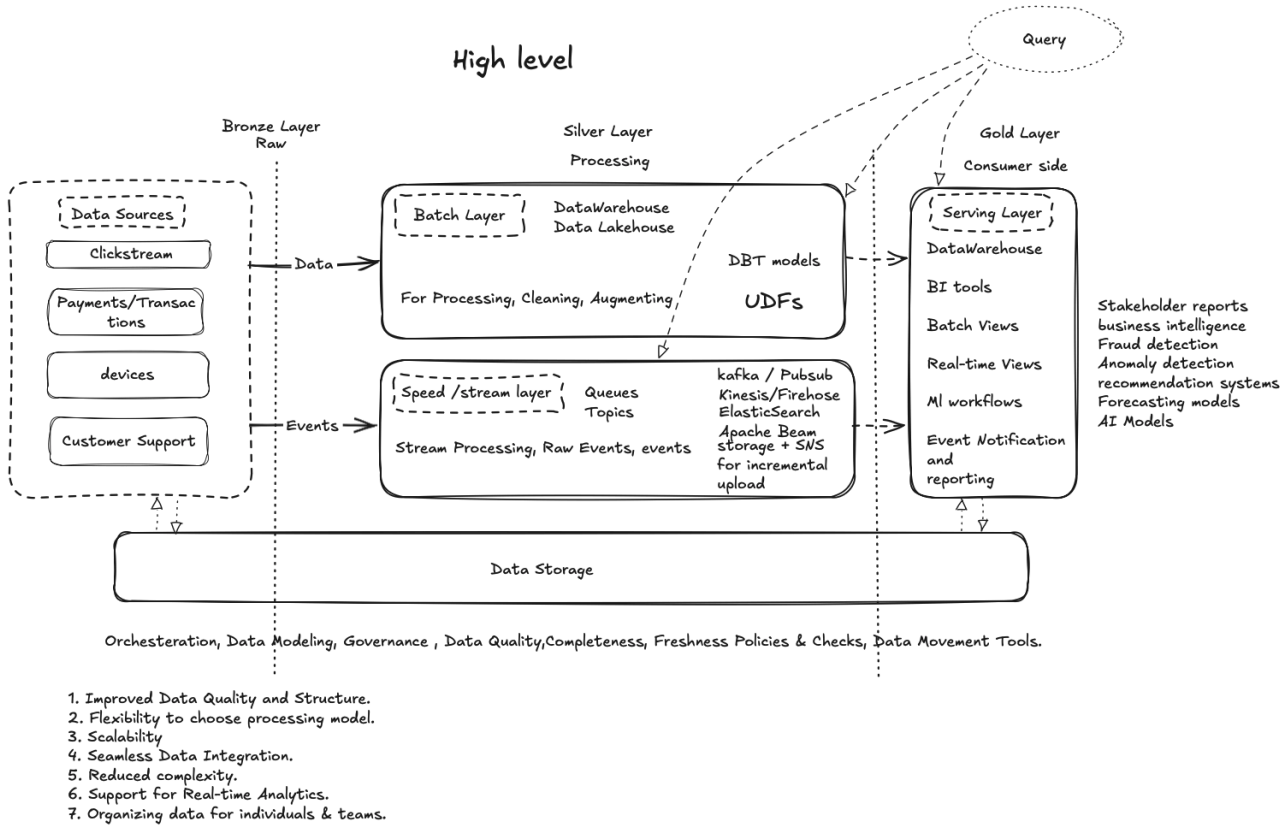
- Since I'm analyzing core platform user, payments, CS & product data, it would be good to stream that data to detect events and user activities earlier as i can't process these in batch for fraud detection or come up with detailed improved conversion rates without also applying batch transformations & modeling, also im given multiple choices between task1 & task 2. so i thought why not come up with a hybrid solution approach?

- Therefore; i decided to go with a hybrid approach between the lambda & medallion architecture for both streaming, batch processing and a data organisation / data-lake convenient architecture and see what the team thinks.

- I know that **Lambda architecture** contains a few key components:
    - **Data Sources:** Core platform (user, payments, CS, product) events
    - **Batch Layer:** Aggregations, historical modeling, ML features
    - **Streaming Layer:** Near real-time user activity and anomaly detection
    - **Serving Layer:** Analytical access for teams (BI, ML, security)
    - **Storage Layer:** Backing up all my data.
- And I thought ok, why not also add **Medallion architecture** labeling on these assets at the same time benefit classifying my organization data for distribution across different teams, along with good data governance and auditing labeling benefits.
    - **Data Sources / Ingestion Layer** (Bronze): Direct ingestion of events from all sources.
    - **Processing Layer** (Silver): Cleaned, processed, and enriched data
    - **Serving Layer** (Gold): Aggregates, session metrics, SCD customer profiles, and team-specific datasets

- This hybrid approach ensures **low-latency processing** for fraud detection while supporting **batch-based modeling** for insights and improved conversion metrics. It also enforces **data governance**, auditing, and team-specific access through clear Medallion labeling.

High Level Architecture Diagram



Behavior (Flow):

- Data consumed from data sources, & core logging events.
- Data cleaned processed in Batch Layer in ETL using Dataflow/Spark/Flink or ELT with a MPP database, or a query engine / data lakehouse, partitioned & clustered.
- Event Stream processed in Streaming Layer, Topics within Queues
- Data and Events are both Backed up by Data Storage
- Data then produced for the serving layer for production grade & high quality processing.
- Flow control of Data between my three operational Layers is done by my Orchestration, Data modeling, Transformation, Movement & Governance tools.

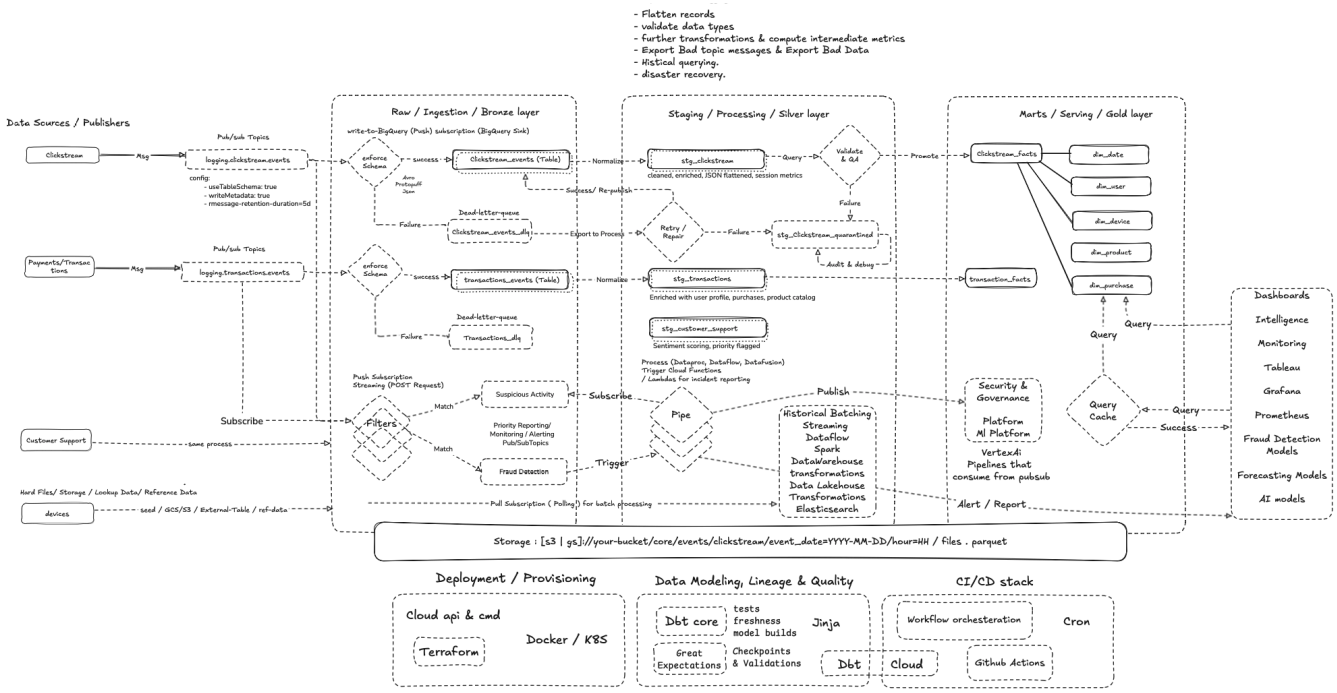
Benefits (Business Value):

1. Improved Data Quality and Structure.
2. Flexibility to choose a processing model.
3. Scalability
4. Seamless Data Integration.
5. Reduced complexity.
6. Support for Real-time Analytics.

7. Organizing data for platforms, reporting, individuals & teams.

Deeper Dive

With Design decisions and trade-offs



IMG_2

In a Deeper look, i have detailed components & necessary flow paths, a detailed flow outlining data flow through my 3 Important Data Layers.

- Raw/Ingestion
- Staging/Processing
- Marts/Serving

While still maintaining the high level details of data sources storage existence, data flow control tools, serving layer APIs and tools.

Detailed Behavior (Flow):

- For every **Data Source** set-up a **pub/sub** topic.
- For every **pub/sub** Topic create a **subscriptions**:
    a. **BigQuery Subscription** (Sink): a write-to-bigquery push subscription that writes and updates bigquery tables in real-time. And While doing so, ensure the following:
        i. Enforce Schema on the incoming event data.

        ii. Write metadata of the messages to the raw tables
        iii. Set message retention duration to a certain period
        iv. Configure Message retries.
        v. Configure Expiration period
        vi. Configure Message ack-deadline
        vii. Give the Subscription, DataEditor permissions to write-to-bigquery and Publisher.
    b. **Regular Pull Subscription** to take a closer look at the incoming data and be able and use it for batch processing.
    c. **Filtered Push Subscriptions** on certain fields such as {'status' : 'suspicious'} that will be sent to external end-points or alerting systems, security and machine learning platforms.
- For every main large scale **Subscription** create a **Dead-Letter-Topic**, and related **subscribers** and that will ensure the following:
    a. Publish **un-acked** or Schema adhering **failed** messages to a **dead-letter Topic**
    b. Publish lost messages to the DLT/DLQ topic.
    c. **DLQ** Subscribers to read data from DLT/DLQ process it and decide to:
        i. **Republish** on successful repair to the original big query subscription table.
        ii. **Quarantine** on failure to Quarantined assets for auditing & debugging
- Start **Processing & Normalizing** data into **Staging Layer** under prefix e.g. stg_dataset

- **Processing: Data Models, Query, Transform & Quality check** Validating and QA-ing my Staging Assets. Ensure the following is done within staging layer:
    a. Flattened Records.
    b. Validating Data Types.
    c. Deeper Normalization.
    d. Further Complex Transformations & compute joins for intermediate staging / silver grade assets.
    e. Quarantining Bad Quality Data for study, debug and research.
    f. **Export Partitioned Data** to storage for **backup**, **historical querying**, **disaster recovery**.
- The Raw Data is unpacked and processed in staging, we start promoting data into the **Production** grade Mart Models (**Serving layer**) (gold) for insights on core platform behavior points.
- **Kimball Model**: Facts and Dimensions (Star Schema)
    a. **Flexibility**: allowing your data to be easily sliced and granulated in any way the organisation wants to and you can **evolve schema** in response to business changes.
    b. **Performance**: it's designed for high speed querying and reporting due to simplicity and organization.

c. **Fast Time to Value**: focusing on specific business processes and building data marts incrementally, organisations can achieve quicker results compared to more complex, integrated approaches.
d. **Integrity**: by defending a clear grain using dimensions, the model ensures consistency and accuracy in data across different business processes.a
e. **Factors to influence Performance**:
    i. **Star Schema** was originally created for performance on relational database systems.
    ii. **Denormalizing** (Flattening) dimension tables sometimes is a key design pattern that enhances query performance.
    iii. **Conformed Dimensions**: Using conformed dimensions, which are consistent across different business processes, improves the ability to integrate and compare data, enhancing overall analytical performance.

    iv. **Business Process Focus:** By building data warehouses one business process at a time, organizations can incrementally deliver value, making performance improvements visible sooner.

f. **Limitations**:
    i. Schema complexity with change can toughen up because adapting to evolving business rules or source systems can be challenging as schema complexity can increase with each modification.
    ii. Can limit historical tracking: in complex kimball environments with multiple data marts, tracking historical changes across different processes can be difficult especially when data is already mature and granular.

- **To optimize dashboards and analytics performance**, I propose introducing a **Query Cache** as part of the serving layer.

  Why? :

  - Dashboards and BI tools often repeat similar queries (daily metrics, aggregates, KPIs).
  - Recomputing these queries directly against the warehouse increases cost and latency.
  - A cache layer provides faster responses and reduces load on the underlying systems.

Key Design Points:

- **Type:** In-memory distributed cache (e.g., Redis, Memcached, or BigQuery BI Engine for native integration). Elas
- **Scope:** Store results of frequent queries, pre-aggregations, and materialized views.
- **Refresh Strategy:**
    - Time-based TTL (e.g., refresh every 5 minutes for near real-time KPIs).
    - Event-driven invalidation (e.g., cache purge when new data lands in the gold layer).
- **Governance:**
    - Monitor cache hit ratio to ensure effectiveness.
    - Track query lineage so cached results remain consistent with warehouse schema evolution.
- **Benefits:**
    - Sub-second latency for BI dashboards.
    - Reduced warehouse query costs.
    - Improved user experience for analytics teams.
    - Adding Elasticsearch is a **good move if we expect a lot of ad-hoc search, scaling dashboards, log analytics, or real-time investigative queries**. With something like Grafana or Kibana **Dashboards.**

- **Deployment:**
    - **Terraform** is very useful and quick in setting up and destroying environments in a second speed, it provides. Different provider integrations from big clouds to different 3rd party cloud providers like dbt-cloud. And it is definitely part of a modern data engineering stack.
- **Data Orchestration & Modeling:** dbt-core is my main choice for data modeling and orchestration. Ever since dbt was open-sourced it's had great development around dependencies, integratable modules, utility libraries, and a strong community that researches and models data for both large and small organizations every day.

  Most of the time it's cost-effective because it runs directly on the warehouse engine — it doesn't process data itself. Google Dataform is an alternative that uses SQLX instead of Jinja, but it's not widely used and doesn't have as big of a community as dbt.

  Limitations:

  - Jinja templating can be complicated sometimes.
  - YAML config format can be tricky — indentation errors can easily cause failures.

- Modeling requires dependent assets to render successfully, so it can be strict on dependencies.

Terraform provider for dbt seems to be dbt Cloud–specific, but it's worth researching if it can run on Cloud Run or other platforms.

Complex workflows with Airflow/Prefect: dbt itself doesn't handle scheduling across multiple models or dependencies beyond its DAG compilation. Integrating it with Airflow or Prefect is a standard, correct approach.

Working with **Lightweight workflows with GitHub Actions or Cron** is my first approach to schedule and automate Dbt , but GitHub Actions or Cron only work well for **simple, linear pipelines**.

dbt Cloud: dbt Cloud does provide managed orchestration, logging, job scheduling, and resource management. Saying it has trade-offs in cost/flexibility is also correct.

Data Quality (Tests, Freshness) & Monitoring:

1. **Schema-level tests**
    - Ensure columns exist, types are correct, and nullability rules are respected.
    - dbt implementation: dbt tests like not_null, unique, accepted_values.
    - Example: dbt test --select payments validates transaction_id is unique and not null.
2. **Freshness / Latency checks**
    - Validate that the latest partition or batch is ingested on time.
    - dbt implementation: dbt source freshness checks max timestamp per table vs expected ingestion schedule.
3. **Business logic validation**
    - Cross-column or cross-table consistency:
        - e.g., total_amount = sum(item_amount) in transactions.
        - Clickstream counts vs known page loads.
    - dbt: custom tests written in SQL.
4. **Checkpointing & lineage validation**
    - Ensure all upstream sources have been successfully processed before computing downstream tables.
    - Tracks which partitions / offsets have been processed.

        - **Lineage** : **dbt** gives you resource and column level lineage, also gives you data lineage in BigQuery. But **BigQuery lineage** itself doesn't support all services in Google Cloud — it's limited to a few like Cloud Data Fusion, Cloud Composer, Dataflow, Dataproc, and Vertex AI.
        - An alternative could be **OpenLineage** but it requires a lot of consumer clients that consume data from different systems.
5. **Dead-letter / quarantine monitoring**
    - Monitor counts of events failing schema validation or arriving late.
    - Can trigger alerts if thresholds are exceeded.

Technology Justification:

Based on my hands-on experience, I tend to design and implement solutions by considering technology choices that align with the company's tech stack, my familiarity with the tools, and the suitability of each technology for the specific use case.

1. **Pub/Sub** provides globally scalable, serverless messaging with features like message retention, dead-letter topics, snapshots, and seek for replay — ensuring reliable event-driven pipelines.
2. **Push and Pull subscription models** support a wide range of processing patterns, with StreamingPull and flow control offering high-throughput, low-latency delivery.
3. **Batch publishing and flow control** optimize throughput and cost, while ordering keys preserve sequence when needed, making Pub/Sub flexible for real-time ingestion.
4. **BigQuery** acts as a fully managed warehouse, supporting partitioning, clustering, and streaming inserts from Pub/Sub for near real-time analytics without managing infrastructure.
5. **Cloud Functions** enable lightweight serverless compute that can trigger from Pub/Sub events, Cloud Storage updates, or HTTP calls — eliminating ops overhead and simplifying ETL tasks.
6. **Cloud Storage (GCS)** provides durable, cost-effective data lake storage, supporting JSON, Parquet, Avro, and integrates with BigQuery external tables for external and historical querying
7. **IAM and regional configuration** across Pub/Sub, BigQuery, and Storage allow secure, compliant, and cost-efficient setups — reducing cross-region latency and avoiding unnecessary egress fees.
8. **Query caching in BigQuery** and **Pub/Sub flow control** both help reduce compute spikes and cost while maintaining performance for dashboards and analytics.
9. **Labels, retention tuning, and resource management** provide transparency and cost governance, enabling you to break down billing and avoid overpaying for unused retention or egress-heavy queries.
10. When used together, this stack allows **real-time ingestion (Pub/Sub)**, **durable staging (Storage)**, **serverless transformations (Cloud Functions)**, and **low-latency analytics (BigQuery)** — a unified, cloud-native data platform.

Handling Late-Arriving Data:

Problem: Data may arrive out-of-order (e.g., clickstream delayed by network, payment retries).
Strategies:

1. Use event-time processing
    - Always timestamp events at source time (event_timestamp), not ingestion time.
    - Partition tables by event date, not ingestion date.
2. Late-data buffer
    - Introduce a staging/quarantine layer (e.g., stg_quarantined) to hold late data temporarily. For further debugging & repairing.
    - Example: Ingest all data into a staging bucket/Raw table first; process into Silver/Gold only after validation.
3. Windowed reprocessing
    - Allow your ETL to reprocess data for a sliding window (e.g., last 7 days) to fix late arrivals.
    - BigQuery: MERGE late-arriving data into partitioned tables.
4. Dead-letter queue (DLQ)
    - Store events that fail schema validation or are too late in a separate topic/bucket for manual or automated reprocessing.

GDPR Data Deletion Strategies:

1. **Logical Deletion (Soft Delete):**
    - Add a **deleted_at** timestamp to Raw, Silver, and Gold tables to mark records as deleted without physically removing them.
    - Pros: Simple, preserves historical analytics.
    - Cons: Data remains in storage; may not fully satisfy strict deletion requirements.
2. **Physical Deletion:**
    - **Use BigQuery DELETE** statements for partitioned tables.
    - For raw GCS/S3 files, either rewrite affected partitions or use formats like Delta/Iceberg that support efficient row-level deletes.
3. **Hashing / Pseudonymization:**
    - Redact sensitive identifiers with hashes (e.g., SHA256 (user_id)) to retain analytics while removing direct PII.

Best Practices:

- Document data flow to track all PII fields.
- Redact PII Data, in snowflake there's a masking feature for as an alternative DW
- Maintain audit logs of deletions.
- Apply partition-aware deletes for efficiency.
- Integrate deletion with retention and archival policies.

Part2: Implementation (Option B)

Approach:

- Following My Architecture, I Thought of creating an MVP that includes the core process of the architecture along with managing **Project Deployment, Data Modeling, Orchestration, Quality, Scheduling.**
- **Separation of concerns between infrastructure provisioning and data transformation where i let:**
    - **Terraform:** Handle **Raw Data Layer (Ingestion)**
    - **DBT:** Handle Transformations, Orchestration on **Staging > Mart (Processing and Serving) Layers**
- Creating the following:
    - **Terraform** config in **/Deploy** to provision the following:
        - For each schema in **/schemas**:
            - Spin up:
                - **Pub/Sub Topics**
                - **Dead-letter-Queue**
                - **Table as pubsub BigQuery Sink**
                - **Subscriptions**:
                    - **Push BigQuery subscription**
                    - **Pull Default** (source Topic)
                    - **Pull Default** (DLQ Topic)
                - Grant my **Google Service Account roles**:
                    - bigquery.dataEditor
                    - pubsub.publisher
                      Pubsub.subscriber
    - **publisher.py** class using the same client
    - **json_to_csv_converter.py** to convert **product_catalog.json** to .csv
        - **NOTE:** That is because I wanted to demonstrate how dbt seed works for uploading external or reference data, a good use case could be Iso-standards or data taxonomy or definition tables and in some cases **quarantine** data
        - Dbt seed can not seed tables in json , it only takes CSVs

Models:

1. **Staging Layer (Data Ingestion & Cleaning)**

- **stg_clickstream.sql** - Cleaned clickstream events with pub/sub metadata
- **stg_transactions.sql** - Processed transaction data with proper typing
- **stg_customer_support.sql** - Support ticket data
- **stg_product_catalog.sql** - Product reference data
- Key Features: BigQuery optimization, pub/sub lineage tracking, STRUCT handling ??

2. **Marts Layer - Kimball Star Schema**

- **fct_events.sql** - Central fact table unioning clickstream + transactions
- **dim_users.sql** - User dimension
- **dim_date.sql** - Date dimension with calendar attributes
- **fct_daily_summary.sql** - Daily aggregated metrics

3. Process daily transaction and clickstream data:

- **fct_user_purchases.sql** - Purchase patterns & RFM segmentation
- **fct_user_browsing.sql** - Browsing behavior & category preferences
- **dim_user_segments.sql** - Combined user segmentation
- **fct_user_journey.sql** - Category-level conversion funnel

4. **Advanced Features**

- **dbt snapshots:**
    - Dbt's way of creating SCD-Type2 to automatically tracks changes to source data over time.
    - What it does is creates versioned records with **dbt_valid_from** and **dbt_valid_to timestamps.**
    - **Running** with a dbt **snapshot** command perhaps daily or weekly.
- **dim_customers_scd2.sql** - our scd2 model for customer interaction.
- **customer_profiles_snapshot.sql** - dbt snapshot for automated SCD tracking.
- **stg_customer_profiles.sql** - Customer profile staging.
- **Generate_schema.sql macro:** because by default dbt adds environment e.g. dev,stg,prd as a postfix/prefix to dbt models when being referenced and created, resulting in something like project_name.stg_dev_stg, or similar , i've changed it to only put source model name. E.g. project_name.stg.stg_schema.

Schema Evolution:

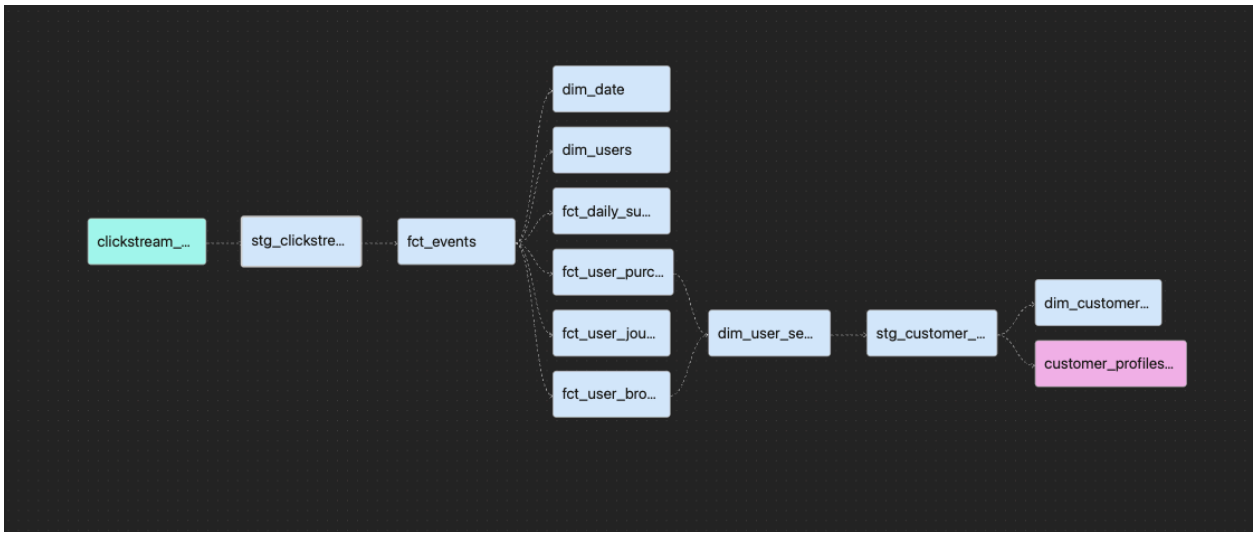
As data models grow with business complexity—especially in a Kimball approach where granularity and normalization increase—schema evolution must be carefully managed to avoid unnecessary complexity. Best practices include:

- **Flexible Storage:** Store less-critical or rapidly changing fields in a JSON column.
- **Additive Changes:** Always add new fields instead of dropping existing ones. BigQuery supports adding columns (including nested/RECORD types) without downtime, ensuring backward compatibility (old rows remain NULL).
- **Streaming Compatibility:** With Pub/Sub streaming into BigQuery, new fields are automatically recognized once the table schema is updated, so incoming messages seamlessly populate new columns.
- **Automation:** Use schema migration scripts to detect and add new fields as they appear.
- **Documentation:** Maintain clear records of schema changes to avoid unnecessary complexity and ensure transparency.

Quality Tests:

- Since my core events are coming from **marts/fct_events.sql** i've added some tests to it to ensure validation in **/tests**. That will get rendered by the `dbt test` command once it runs.
- As I'm running my dbt tests, i've found events with
    - **{user_id**: **null, event_id: evt_1234567896, session_id: ses_ccc333 }**.
    - that event_id and session_i doesn't have a user therefore i've added it to a **quarantine seed**. And therefore isolate it from my **staging_slickstream** model.

dbt lineage:



Cost Overview & Approach Discussions

- Pub/Sub: Around $0.40 per million messages. For 100M events per month, that's about $40.
- BigQuery Storage: $20/TB for active data, $5/TB for long-term storage. With 5TB raw and 10TB processed, that's roughly $250/month.
- BigQuery Queries: $5 per TB scanned. If we scan 30TB a month, expect about $150.
- Dataflow: For light dev/test workloads, $200–400/month. Full production streaming and batch pipelines can run $2K–5K/month.
- To help manage both **budget and performance**, dbt profiles can be configured with BigQuery-specific settings:
    - **`maximum_bytes_billed`** – limits how much data a query can scan, preventing unexpectedly high costs.
        - Example: 1 GB for production, 100 MB for development.
    - **`job_execution_timeout_seconds`** – automatically cancels queries that run too long to save compute costs (e.g., 5 minutes).
    - **`job_retries`** – controls how many times failed jobs are retried; here set to 1 to balance reliability and cost.

These settings can be applied to **dbt target profile**, giving us **predictable costs, faster feedback, and safe query execution** across environments.

Budgeting Approach

- I kept costs under control by actively monitoring usage, using a cloud cost calculator to plan, and leveraging AI-based estimations to make spending projections more visible.
- This approach ensured scaling without surprises.

Improvements:

Design:

- Add deeper dive into staging models, transformations, ELT & ETL pipelines, message alerts. Different team integrations, 3rd party API integrations across my Data Architecture & along with monitoring and governance tools, security and auditing approaches.

Implementation:

- Add github actions on PR for Tests and run on dev & staging projects.
- Add pre-commit hooks on commit, push and PR.
- Add Linting SQL-fluff Sql-linter
- Better Lineage and docs
- Deeper Look at the data
- There's always room for improvements in a project.

For The Team

- This design reflects my own process and thinking—I drafted 100% of the Design and 90% of this document myself, taking the time to document the approach in detail for the team. I used AI sparingly to enrich trade-off analysis, validate implementation ideas, and assist with a few SQL queries and extra tests. All diagrams were hand-crafted in Excalidraw. (with great technology, comes great responsibility).
- Thanks for the opportunity to put this together—I'm excited to hear your thoughts, questions, and feedback, and to keep the discussion going with the team.

Reference Docs

Lambda & Kappa Architecture
https://hazelcast.com/foundations/software-architecture/lambda-architecture/
https://hazelcast.com/foundations/software-architecture/kappa-architecture/

Dbt
https://github.com/dbt-labs/jaffle-shop
https://docs.getdbt.com/docs/core/connect-data-platform/bigquery-setup#maximum-bytes-billed
https://docs.getdbt.com/reference/resource-configs/bigquery-configs
https://docs.getdbt.com/best-practices
https://docs.getdbt.com/docs/core/connect-data-platform/bigquery-setup#dataset-locations

Pubsub & Big Query
https://cloud.google.com/pubsub/docs/subscription-properties?authuser=3#bigquery
https://cloud.google.com/pubsub/docs/create-subscription
https://cloud.google.com/pubsub/docs/create-bigquery-subscription
https://cloud.google.com/pubsub/docs/audit-logging
https://cloud.google.com/pubsub/docs/reliability-intro