TransferRoom - Take Home

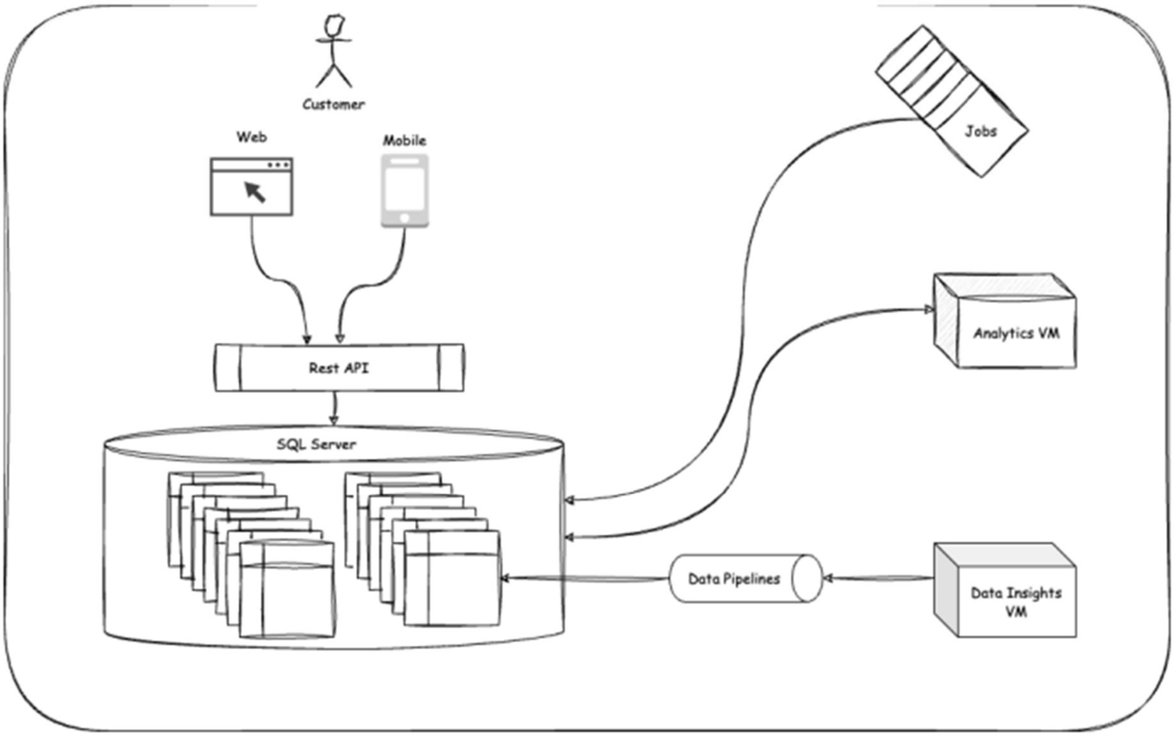

Figure 1 TransferRoom Take Home – Current Architecture Diagram

Current Architecture (Problems):

- **Analytics and Application workloads on same database**
  - Large analytical queries should never take down the whole platform / DB or affect the natural running of the app where analytics are not relevant
- **Lack of central orchestrator**
  - This can make lineage hard to understand and often hurts debugging and discoverability
  - Alerting tends to be quite all over the place or more expensive to maintain
- **Lack of SQL Templating Framework**
  - This often leads to spurious DDLs, inconsistent write dispositions / methods (often leading to duplicates etc..), lineage issues, less testing, longer iteration cycles due to poor local development workflow etc...
- **No consistent means of getting API data into 'warehouse'**
  - Likely means time is wasted repeatedly trying to convert nested chunks of JSON into tables (and evolving the schemas, etc...) manually
- **Lack of CDC (Change Data Capture) stream**
  - **NOTE – CDC is basically a big log that shows when a table has had an insert, update or delete**
  - This effectively means that some amount of data is being continuously 'lost'. Specifically, knowing the exact state that a dimension table was at

some given time rather than it's current state (e.g. user goes active, to inactive, back to active)

  - SCD-2 (slowly changing dimensions) are an important concept in data / analytics engineering which are tables specifically designed to allow analytics that may require knowing the state of a dimension at some given point in time like a user attribute (e.g. where they live)
  - It's possible that the analysts are taking snapshots of the table as an alternative which is ok, but this is not as reliable and can still result in data being lost. Having a CDC stream / copy means you can recreate the state of the DB at any point in it's history (from the day you start running it)

- **Lack of dedicated OLAP database (warehouse)**
  - OLAP databases store data in a columnar fashion rather than row based like OLTP ones (e.g. Postgres, MySQL etc…) which makes things like aggregations of large numbers of rows much faster
  - Backfilling, be it due to mistakes, changing a metric or wanting a new metric backdated, is an essential component of any data platform, and you want this to generally be as fast and easy as possible. Using an OLTP database means that this is generally going to take significantly longer which means it takes longer to fix production issues once the issue is found, and even longer if multiple runs are needed due to incorrect fixes
  - The slowness issue also affects local development as slower calcs mean that it is more likely for a dev to get distracted while waiting / find something else to do (this is very costly), and also that they will fundamentally have a longer time debugging
  - OLTP databases also tend to be less feature-rich from an analytics standpoint compared to OLAP databases (→ lost productivity for analysts / data engineers)

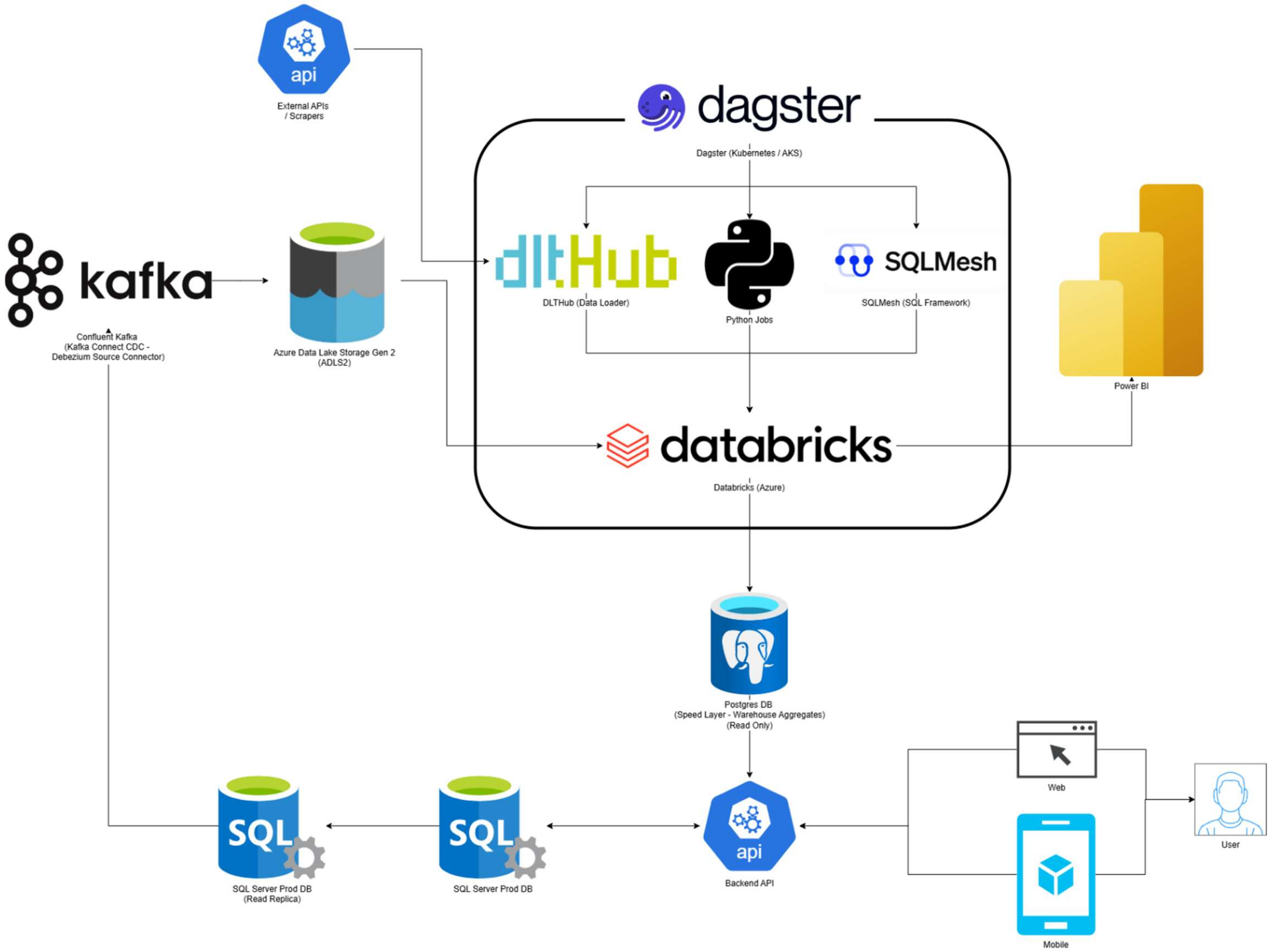

Figure 2 TransferRoom Take Home – New Architecture Diagram

Proposed Solution(Benefits):

- **Separated Application Workloads and Analytics Workloads (**Read Replica + CDC Stream via Kafka Connect / Debezium**)**
  1. **Production DB Performance**
     - The primary goal is to ensure the **main production database** functions solely as an **application DB**, minimising the risk of performance issues caused by analytics jobs
     - Phase 2 could include investigating inefficiencies in API-to-database interactions (e.g., reducing multiple API calls into a single optimized query)
  2. **Change Data Capture (CDC) Stream via Read Replica:**
     - The **CDC process** should run on the **read replica** (database mirror) to avoid stressing the production DB.

        - CDC logs (or transaction logs used for CDC tables) are typically retained for **3-7 days**, so a system must **continuously read and store** them elsewhere in a **cheaper storage format**
    3. **Tool Choice:**
        - **Debezium (via Kafka Connect)** is the most well-known tool for this purpose
        - **Managed Kafka (Confluent Cloud)** has been picked instead of self-hosting Kafka, as it's more reliable and cost-effective for this specific use case
        - **Kafka Connect** enables smooth data transfer from the source (SQL Server) to a target system
        - Alternatives Considered:
            - **Estuary (SaaS product)** – Possibly a good option, but have concerns about company size/reliability. Worth reconsidering only if Confluent Cloud is too expensive
            - **Azure Data Factory (ADF)** – Might support this, but reliability is a concern. We may test it in the future, but for now, would want to prioritize proven, stable solutions to avoid data loss risks.
        - Alternatives like **AWS DMS** and **Estuary** exist, but Debezium remains the most proven solution (and the first one isn't an option anyway as we are on Azure)
    4. **Storage Format Considerations:**
        - Data will be stored in **Azure Data Lake Storage (ADLS2)** in **Parquet format**.
        - **Why Parquet?**
            - It's a **columnar storage format**, offering:
                - **High compression** (efficient storage)
                - **Fast reads & loads** (ideal for data warehouses)
                - **Great support** (due to being industry standard)
        - Once in ADLS2, data can easily be used by **Databricks** (which natively integrates with ADLS2) or most modern execution engines for that matter
    5. **Storage Format Considerations:**
        - This setup can be done **entirely via the UI (on Confluent Cloud)**, making deployment straightforward
- **Dedicated OLAP Database / Warehouse (**Databricks**)**
    1. **Tool Choice:**
        - **Databricks** has strong **Azure integration** and often gets **priority access to new features**
        - **Snowflake** is great for **multi-cloud** environments but doesn't have the same deep Azure integration

- **Azure Synapse** is another option but is generally considered a **weaker version / knock-off of Databricks**, so it's not being considered
- **BigQuery** (GCP-based) is excellent but **not relevant** for this setup
  - BigQuery Omni exists but lacks key features, so it's not a better option

2. **Databricks Usage Philosophy:**
   - Even though **Databricks has many features**, it's best for us to **use as few features as possible**
   - **Primary use case: SQL data warehouse** with occasional **ML workloads in PySpark** when needed
3. **Data Lakehouse & Access Control (Unity Catalog):**
   - **Unity Catalog** is the **schema and access control layer** for data stored in **Azure Data Lake Storage (ADLS2)**
   - **How it works:**
     - Acts as an **abstraction layer** over **blob storage**, making it behave like a traditional warehouse
     - Uses **Delta Lake** (Databricks' ACID-compliant table format) to maintain **reliability** over raw **Parquet files**
     - **Growing support for Iceberg** (due to Tabular Acquisition), but **Delta Lake remains a 1st class citizen**
4. **Performance & Scalability Considerations:**
   - **ShellCorp's expected data size** is likely in the **terabytes (TBs)**, not **petabytes (PBs)**
   - This means that **standard best practices** should be enough to ensure **strong performance**
   - Unlike **PB-scale setups**, **no extensive tuning** should be needed, so **setup and deployment will be quick**

- **Dedicated Central Orchestrator (**Dagster**)**
  1. **Avoid No/Low-Code:**
     - **Azure Data Factory (ADF)** and other **no/low-code pipeline tools** (e.g., Databricks Workflows, Snowflake Tasks etc...) can be fine in certain situations, especially when a company has **limited data engineering expertise**
     - However, **service-native workflow tools** tend to be **limiting** in areas such as:
       - **Testing** capabilities
       - **Alerting** flexibility
       - **Dynamically created DAGs**
       - **Data catalog integrations**
       - **Etc...**
  2. **Tool Choice:**

- **Airflow** is the older market leader, but **Dagster has gained strong momentum** due to its modern approach
- From experience, **Dagster has proven to be an excellent orchestration tool**, offering:
    - **Customizable alerting**
    - **Dynamic DAG structures**, useful for:
        - APIs
        - Looping over **database tables**
        - **Custom/unsupported integrations**
    - **Global asset lineage**
        - Supports **manual runs based on lineage**, similar to **DBT's asset+ syntax** (run asset plus all child dependencies)
    - **Metadata & Repository Integration**
        - Publish **metadata** such as **owners, descriptions, and group names**
        - Custom-built tools can:
            - **Emit metadata & lineage** for all assets/classes used
            - Use **LLMs to prevent code drift** (e.g., automatically generating documentation, column descriptions)
    - **Local Testing & Cost Control**
        - Test workflows **locally** without **hitting Databricks or incurring costs**
        - Use configs to **scale up/down resources** easily
        - Perform **unit tests before running full jobs**
    - **First-Class Support for Asset/Data Quality Checks**
    - **Reusable Utility Functions**
    - **Supports Lightweight Python Jobs**
        - Great for tasks like **OpenAI calls** or **small data transfers**.
        - Avoids unnecessary **Databricks cluster usage**, allowing full control over **memory and compute allocation** (just a function decorator needed)
    - **Replaces Non-Application related Jobs in Azure**
    - **Excellent Support for DBT / SQLMesh**

3. **Deployment Strategy:**
    - Dagster will be **deployed via AKS (**Azure Kubernetes Service**)**
    - **Dagster Cloud is being ignored** as it can be a bit of a rip-off unfortunately

- **SQL Transformation / Templating Framework (**SQLMesh**)**
    1. **Why use a SQL templating framework?**
        - More Accessible for Analytics Engineers:
            - SQL is more widely known than Python/PySpark, making it easier for **BI analysts** to contribute

- Python models can still be used when PySpark is necessary
- **Onboarding new hires is easier** since many have experience with DBT
  - DBT is largely industry standard these days (SQLMesh is just a newer alternative with some benefits that I will outline later)

- **Improved Local Development Workflow:**
  - Allows **basic orchestration of SQL models** (tables) **without** needing to spin up expensive or slower environments (e.g., Databricks, Dagster)
  - Faster **iterations on code**
- **First-Class Support for Unit Tests & Data Quality Checks:**
  - Data quality checks are **already strong in Dagster**, but **unit testing SQL is often neglected**—this helps enforce best practices
- **Community-Made Extensions & Integrations:**
  - Popular DBT extensions (e.g., **dbt-utils, dbt-expectations**) simplify data quality checks and macros
- **Enforces Standardization:**
  - Helps prevent **inconsistent SQL/PySpark implementations**
  - Improves **lineage tracking** for transformations

2. **Why SQLMesh > DBT?**
   - **DBT Core Has Become Stagnant:**
     - Since DBT's **large VC funding rounds (**$200M Series D**)**, innovation has slowed, and many features are now **locked behind their expensive cloud offering**
       - SQLMesh, though also VC-backed, has **less funding pressure (**$20M Series A**)** and is **actively solving major DBT pain points**
   - **Python Macros Instead of Jinja Macros:**
     - DBT **only** supports Jinja macros, which lack proper testing and type support
     - SQLMesh allows **Python macros**, which can be **type-hinted and tested**, making them **more reliable** for complex pipelines
   - **SQL Checks at Compile Time:**
     - SQLMesh **understands SQL natively**, allowing it to check for **errors without hitting the data warehouse**
     - This saves money and is **much faster** than sending queries to the warehouse
   - **First-Class Unit Testing:**

- SQLMesh **executes unit tests with DuckDB**, avoiding unnecessary warehouse queries
- Even **CTEs (Common Table Expressions) can be tested independently**

- **Native Table Diffing (Prod vs. Dev):**
  - **Compare dev vs. prod data** using a simple CLI command
  - Makes it easy to see how changes will affect production (normally a painful manual process)
- **Automated Detection of Breaking vs. Non-Breaking Changes:**
  - Similar to Terraform, SQLMesh detects whether a change is **breaking or non-breaking**
  - Helps answer:
    - **Do we need to backfill this metric?**
    - **Do downstream models need to be rerun?**
- **Native Batching & Start Date Support:**
  - SQLMesh allows defining **batch sizes** (e.g., 7 days) for backfills, preventing **resource overload** from massive queries
  - **Start dates** improve code hygiene, clarifying **when data is trusted/usable**
- **Partition Awareness:**
  - SQLMesh **tracks filled vs. missing partitions**, reducing the risk of **data gaps in tables**
- *(There are many other benefits, but these highlight why SQLMesh is a strong alternative to dbt)*

3. **Business Logic & Data Architecture**
   - The **medallion architecture** (bronze → silver → gold) would likely be followed:
     - **(Bronze) Raw layer** → Direct data ingestion.
     - **(Silver) 1+ staging layers** → Data cleaning, transformation
     - **(Gold) Mart layer** → Final business logic layer

- **Standardised Process for Pulling in API Data and Pushing to Warehouse** (DLTHub)
  1. **Why Standardize API Data Ingestion?**
     - Many **API sources** need to be integrated, and having a **consistent method** for:
       - **Fetching API results**
       - **Handling schema evolution**
       - **Managing write disposition**
       - **Generating warehouse-specific DDL**
     - A standardized approach **reduces effort** when adding new data sources

2. **Why Use DLTHub?**
   - **Lightweight Python Library** → Makes API ingestion **simpler & more efficient**
   - **Huge efficiency gains** → Helped **delete 7 AWS services** (Glue Crawler, Glue Jobs, AWS DMS, and Airbyte) in a month at my current job.
   - **Prebuilt "verified connectors"** → Similar to **Shadcn UI**, but for **data pipelines & API sources**:
     - Many common APIs have **ready-to-use connectors** (just **copy-paste into your repo**)
     - For unsupported APIs, DLTHub provides **helper utilities** that make calling APIs and handling data **trivial**
   - **Automated Features:**
     - **Handles paginated JSON responses**
     - **Infers data types automatically**
     - **Manages nested JSON (configurable)**
     - **Controls write disposition (e.g., insert+delete updates)**
   - **NOTE –** This tool is not 100% necessary, but it's a light abstraction that saves time and effort. Since it streamlines API ingestion significantly, there's no reason not to use it.

- **Dedicated Analytics DB / Speed Layer for Fast, Isolated Reads of Metrics (**Postgres**)**
1. **Why Not Use the Data Warehouse for Application Queries?**
   - Using the **data warehouse (Databricks)** as a backend for the application would be:
     - **Too expensive**
     - **Too slow (high latency)**
   - Instead, a **dedicated analytics DB** can be used to **serve pre-aggregated metrics quickly** to the application
2. **How this Works**
   - A **job in Dagster** will **regularly copy** all **business (gold) layer tables/views** into a **separate Postgres database (dedicated for this purpose)**
   - The application will have **read-only access** to this Postgres database
   - Benefits of this Approach:
     - **Safer & More Controlled Data Access:**
       - The analytics DB **only allows reads**, preventing accidental data modifications
       - Ensures a **consistent, controlled write rate** into Postgres.
       - Heavy computation is done **in Databricks** beforehand, keeping **Postgres lightweight**
     - **Performance & Cost Efficiency:**

        - **Warehouse (Databricks) does the heavy lifting** → Postgres is used only for **fast queries**
        - Keeps **application queries cheap and responsive**
3. **Alternative Tool Choices**
    - **SQL Server** – Could also be used instead of Postgres with **no major issue**
        - Downside is losing the ability to use Postgres extensions like pgvector that allow for embeddings and vector search
    - **ClickHouse** – Would be a **good option for large-scale, high-granularity filtering**, but isn't necessary at this point
        - **Best to start with Postgres**, then upgrade **only if needed**
    - **Cube.dev** – This is like the data models in **Power BI** or **Looker** but isn't tied to any single BI tool. It sets up how your tables relate to each other and uses strong caching, so you can quickly slice and dice your data
        - This setup is great for **AI applications** because it reduces the chance of getting incorrect metrics. Think of it as an auto-generated API with clear descriptions for metrics and data details, ensuring you get accurate, vetted information. This is a **step up from Text-to-SQL models**, which almost always produce unreliable results with boundless possibilities to be wrong in ways that are not obvious (and may have no business relevance)
        - Cube would introduce some complexity, and I haven't used it before, but it seems promising. It might be worth exploring in the future, especially after we've tackled our current issues. Plus, with Cube, **both Power BI and our web back-end could use the same API** and data model, keeping everything consistent
            - There are other more basic steps you can take toward consistency (e.g. using wide tables in the data warehouse 'gold' layer), so this is not essential, but is something worth noting

Near Real-Time Capabilities (Dagster + SQLMesh + Databricks + Postgres)

1. **Suitability for ~10-Minute Latency**
    - If **near real-time updates (~10 min latency)** are required, this architecture can support it efficiently.
    - While **DBT** could handle this, **SQLMesh's automatic partition tracking** simplifies **incremental processing**
    - In **Dagster**, a simple **job scheduled every 10 minutes** can trigger **all SQLMesh-managed tables**, followed by the **Postgres copy as the final step**
    - **Key Requirement:**

        - Everything must be processed in **incremental chunks** (i.e., no full refreshes) to avoid exceeding the **10-minute window**
2. **Workflow Breakdown**
    - **MERGE Statements for CDC Log Tables (First Step of the Job):**
        - Ensures that all **Change Data Capture (CDC) log tables** contain **active versions** of each table
        - **SCD-2 tables** (slowly changing dimensions) also need to be processed
        - These tables act as the **raw data sources for SQLMesh**.
    - An alternative:
        - The raw tables **could be updated in real-time instead** (either using spark structured streaming or streaming tables)
        - This might incur **slightly higher costs**, but the **difference vs. running every 10 min is unclear** (may be about the same cost)
    - **SQLMesh Handles Incremental Processing:**
        - SQLMesh ensures that **only new partitions are processed**, preventing unnecessary re-computation
    - **Postgres Copy (Final Step of the Job):**
        - After transformations, results are **pushed to Postgres** for **fast, isolated reads** by the application.
3. **Handling Actual Real-Time Requirements (<1 Minute)**
    - If **true real-time tables** are needed, the best option is **Databricks Streaming Tables**
    - **Potential Issues:**
        - **Higher costs** (since it would require leaving a Databricks cluster running)
        - **Databricks acting as a partial backend**, which may not be desirable
        - **Streaming tables have some limitations** and can be tricky for specific use cases
        - **Micro-batching (~10 min updates) is simpler and likely sufficient** for most use cases
        - Unless **strictly required**, full real-time streaming **should be avoided** due to its complexity and cost implications

BONUS SECTION – Migration Plan

Proposed Plan:

1. **Identify All SQL Snippets + Identify All Stored Procedures (and Azure Jobs) + Convert Pandas Snippets to SQL (DuckDB / MSSQL)**

- The first obvious step is to identify where everything actually is and understand the scope of the variety of transformation 'dialects', but also so that we can setup a system of scripts that allows me to consistently keep track of what has changed week to week very easily (will talk about this more later). The transformation 'dialects' outline are as follows:
  - **Pure SQL → Table:** Very straightforward
    - If it's inside a python script, we may need to move these out into .sql files, but this is relatively trivial
  - **Highly, dynamically generated SQL** → Table: Bit of a pain
    - Will need to come up with a standard file structure and naming convention so that I can have a script that can easily identify and run all said dynamically generated SQL (presumably done with Python), that can output them to a directory as compiled .sql files
  - **Pandas → Table:** Bit of a pain
    - We will want to follow a consistent file structure and naming for these first so that we can once again output them to a directory of 'pandas tables' with a script
  - **Stored Procedures / Azure Jobs → Table (or Modifying Table):** Probably most painful
    - Due to the nature of how finnicky these can be (and the fact that a lot of this stuff is likely being directly done on production I suspect), I think the best option is to a have a script that can easily identify all active stored procedures (should be possible with a single SQL query on the information schema), so that we can be aware of them and make any adjustments in the new architecture as needed
      - I would probably also advise the team to minimise any changes to / usage of those as much as possible so that we have less awkward stuff to worry about
    - If we have jobs in Azure / ADF that just do stuff like pull in API data → Table etc..., I will probably just leave this closer to the actual migration date because these types of jobs are less likely to change as much (I will find out I suppose, but this isn't a major concern – will know better once I see it)

- **Setup SQL Server Read Replica + Confluent Cloud (Kafka Connect CDC) + ADLS2 + Databricks**
  - The SQL Server read replica should be pretty quick to setup, and Confluent Cloud should mostly be the same
  - There is some stuff we will need to run on the read replica to enable CDC, but after that it's just a case of putting the credentials in on Confluent Cloud and making sure that IT / Platform have whitelisted the IP or done whatever is needed to give out Confluent Cloud account access to the SQL Server Replica (we will also need want a user specifically made for Kafka Connect). Same will need to be done for blob storage / the data lake as well (may be a case of IAM roles or what have you for this – not sure if it's the same as AWS, but can't be too difficult)
    - Depending on how things work at the company, the more involved I can be in the process of doing this, the better, as it takes pressure off IT / Platform and should reduce the likelihood of blockers that will delay the migration
  - Databricks should be pretty trivial to get up and running. Presuming that this (like all the other stuff) is done via Terraform
    - If it's a ClickOps situation, then that will be a different situation, but it probably would be a good opportunity to introduce it if it wasn't being used already (based on what I can see from the old Site Reliability Engineer posting, I think Terraform is at least mostly being used nowadays if it wasn't before)
    - As part of this process, one of the main things to check is just that you can connect to Databricks from your PC and that it can connect to the Azure bucket etc...
- **Setup Dagster on AKS (Azure Kubernetes Service) + Repo + CI/CD**
  - This shouldn't be too bad. Historically I have only deployed this on ECS on AWS which is quite a bit easier, but I don't think this is particularly complicated to setup (just need to test it for some time to be happy with the reliability of the deployment, and there will be ample time for that as we will probably have this running for at least a few months before doing the migration)
    - This would be done via terraform and naturally would need to have all the typical IAM permissions stuff done so that it has access to the data lake, to Databricks, SQL Server etc...)
  - Once this is setup, I would probably test out doing something lightweight like copying over some of the API load jobs to push into Databricks, and also maybe test out setting up a streaming table for getting the CDC logs and converting them into individual raw tables (either that or just setup a

more standard pipeline that runs every 5-10 minutes and merges new data in)

  - Once I can get some jobs running locally in Dagster fine, I would then make sure to get CI/CD setup so that we have any easy way of deploying new Dagster images

- **Setup SQLMesh**
  - This should be quite basic as it would just be another directory inside the Dagster rep, but the main goal here would just be making sure that the Dagster integration is setup with it fine, that CI/CD is looking fine, and also that things like the PySpark models work as intended
    - I should note that in the long-term, SQLMesh would be in a separate repo most likely, but for the purposes of the migration, it's easier to have it in the same repo to start with for reasons that will be explained shortly

- **Setup Testing Script + Script for Pulling in Latest Code from Legacy Repo**
  - The testing script would look similar to what I did at my current role for going from Redshift → Snowflake. Basically, the data copied over to Databricks is going to have all the tables from SQL Server. As such, when we run all of our SQLMesh code into different schemas, we can easily have a script that compares the outputs of the SQLMesh created tables with the ones copied over from SQL Server (we only care about derivative tables, not comparing the raw ones) so that we have an easy way to check for correctness
    - I don't follow TDD most of the time (I usually test in the middle of writing my code), but this is case where it works wonders as it lets you aggressively modify the SQL code (and other code) until you get to working state
  - The next important part will be using all the work done on the legacy repo and having a script that I can run locally that will look in the relevant directories for the .sql files in the legacy repo + run any 'migration' python scripts in that repo to make the 'pandas tables' directories up to date, and the dynamically generated SQL files + run the relevant SQL query to get all of the active stored procedures → copying those into an organised structure within the Dagster repo
    - The reason for doing this is that if you commit this code you have copied over, the next time you run this locally, you will be able to hover over in git and see very easily what has changed since you last worked in the repo. This makes it easy to know what you will need to adjust in Dagster + SQLMesh, such that you can hopefully try and get back to working state in the same commit (or squash after) and the push the adjusted code with the latest changes from

    - the legacy repo. This is probably the easiest way to be able to deal with the fact that analysts will be making changes on a weekly or even daily basis once the migration starts
    - It's because of things like this that once you officially start the migration, you want to be in and out as fast as humanly possible (e.g. 1-2 months), as more delay means more business requirements → analysts making changes → more pain migrating (depending on how much the analysts have changed to meet the new business requirements). The other reason is because it's much more plausible to agree on slower velocity for a 1-1.5 months than for like 3+ (which is never going to happen). This is why the tidying before the migration is so crucial
- This process of iteration would continue until tests have all passed / a validation report has been made to explain any potential differences. You would then want to get analysts to start pointing to the new tables for their BI reports, and to also ask them to do a manual check before and after to make sure they haven't spotted anything that looks abnormal. The same would be true for the back-end engineering side who would simply be pointing to the read-only Postgres instance in this case (have skipped the step about loading stuff into the new Postgres instance because it's fairly trivial stuff)
    - You would gradually repeat this process for sets of tables at a time, and on the back-end side possibly for sets of customers / users at a time if that isn't hard for them to do (not 100% necessary, but is just risk management)
- ***MIGRATION DONE***