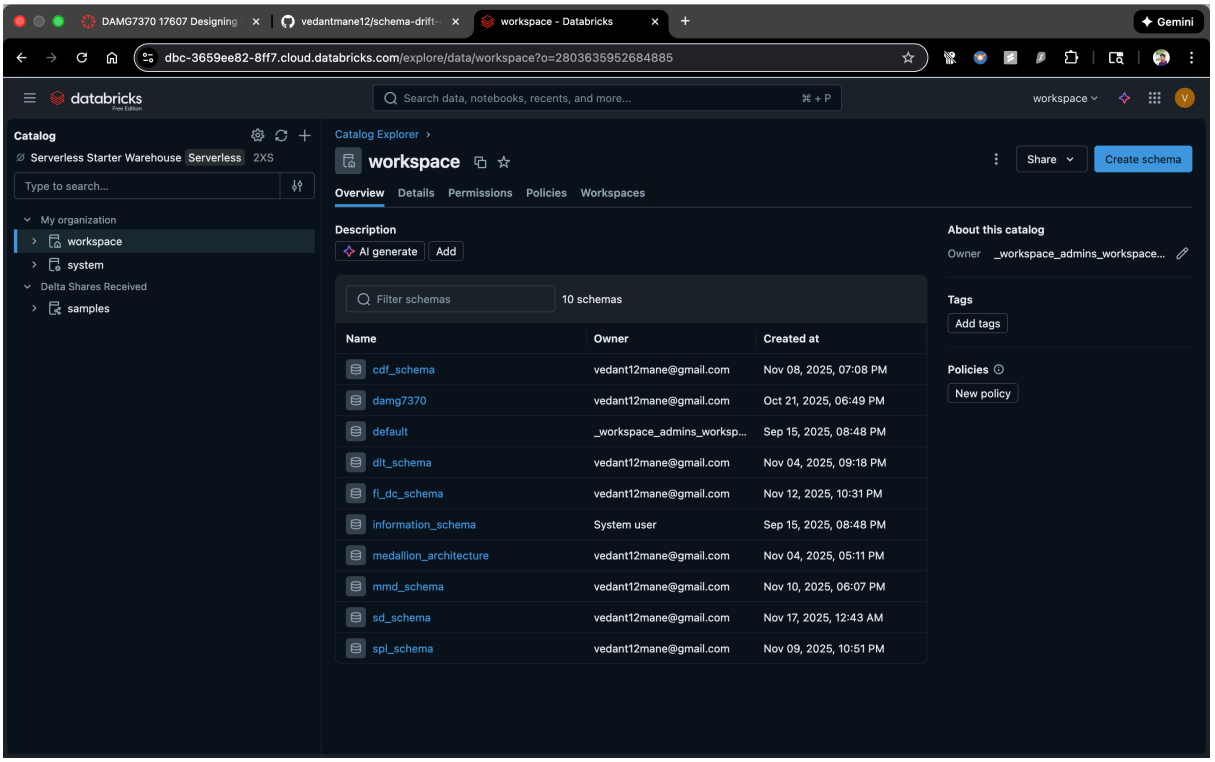
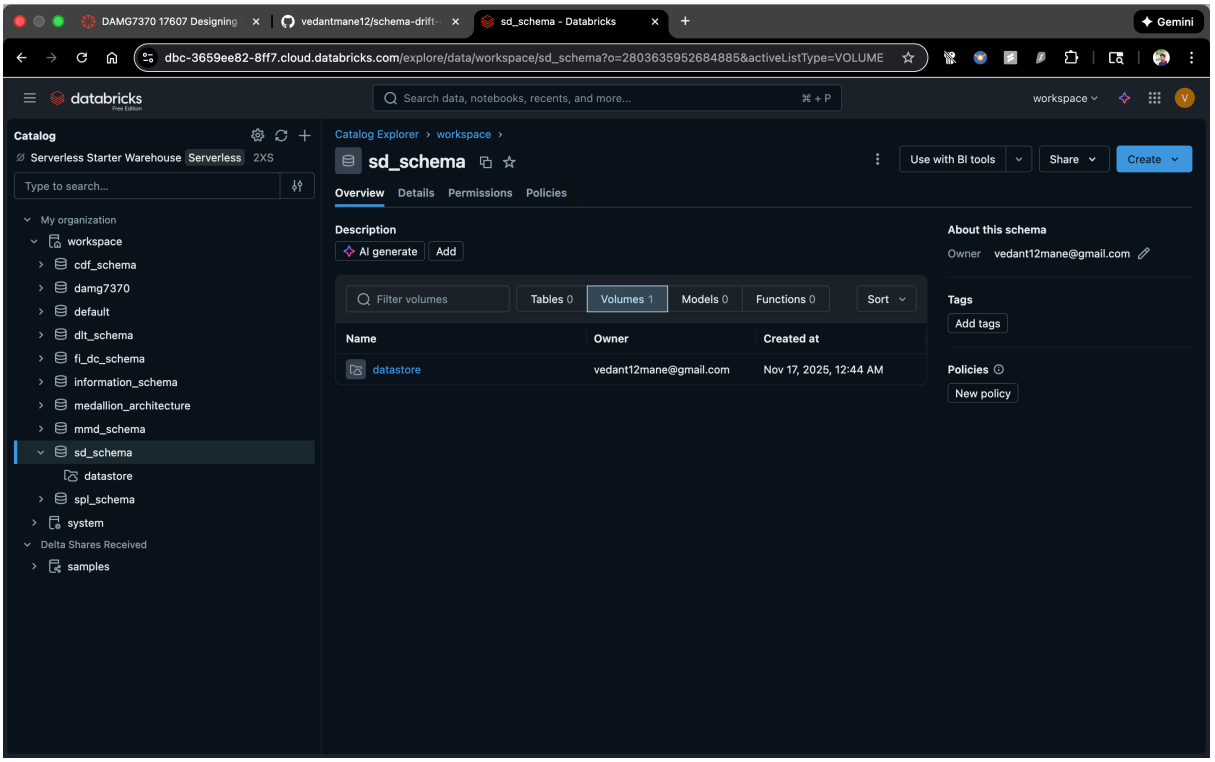
Assignment 09 - Schema Drift Implementation using Python in Databricks

Catalog: workspace

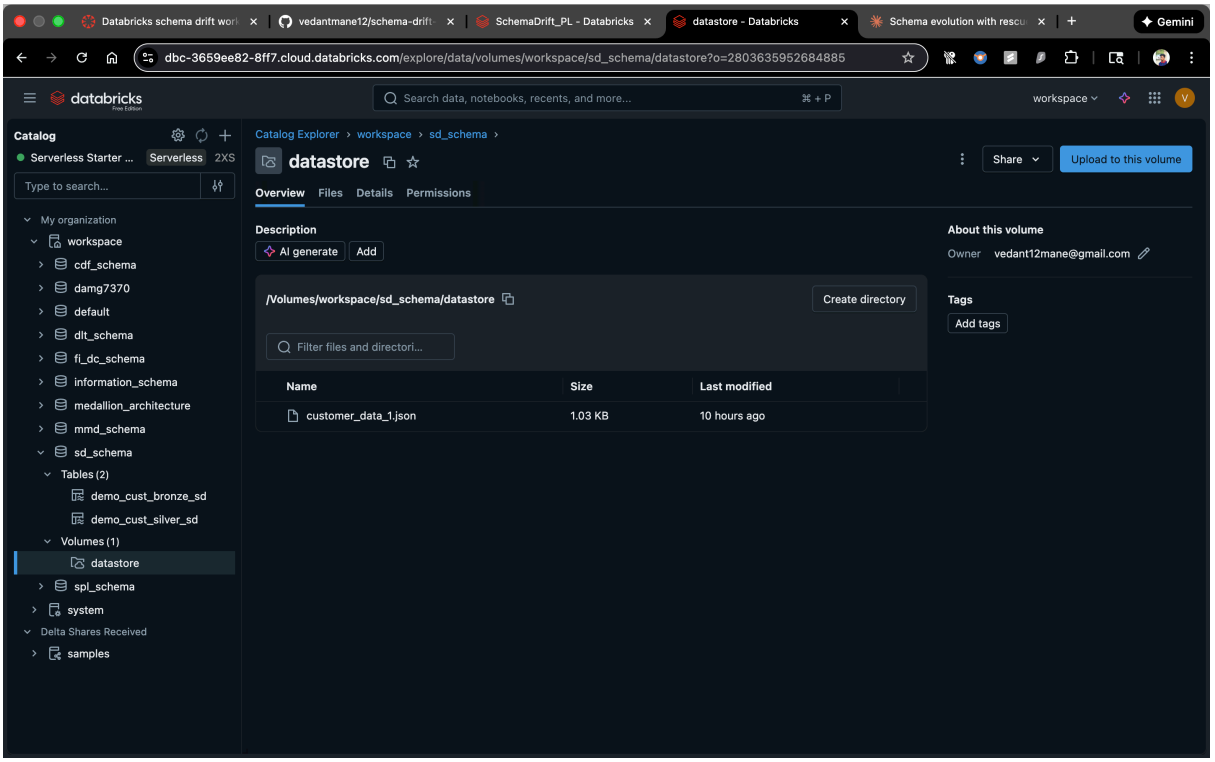


Schema: workspace.sd_schema

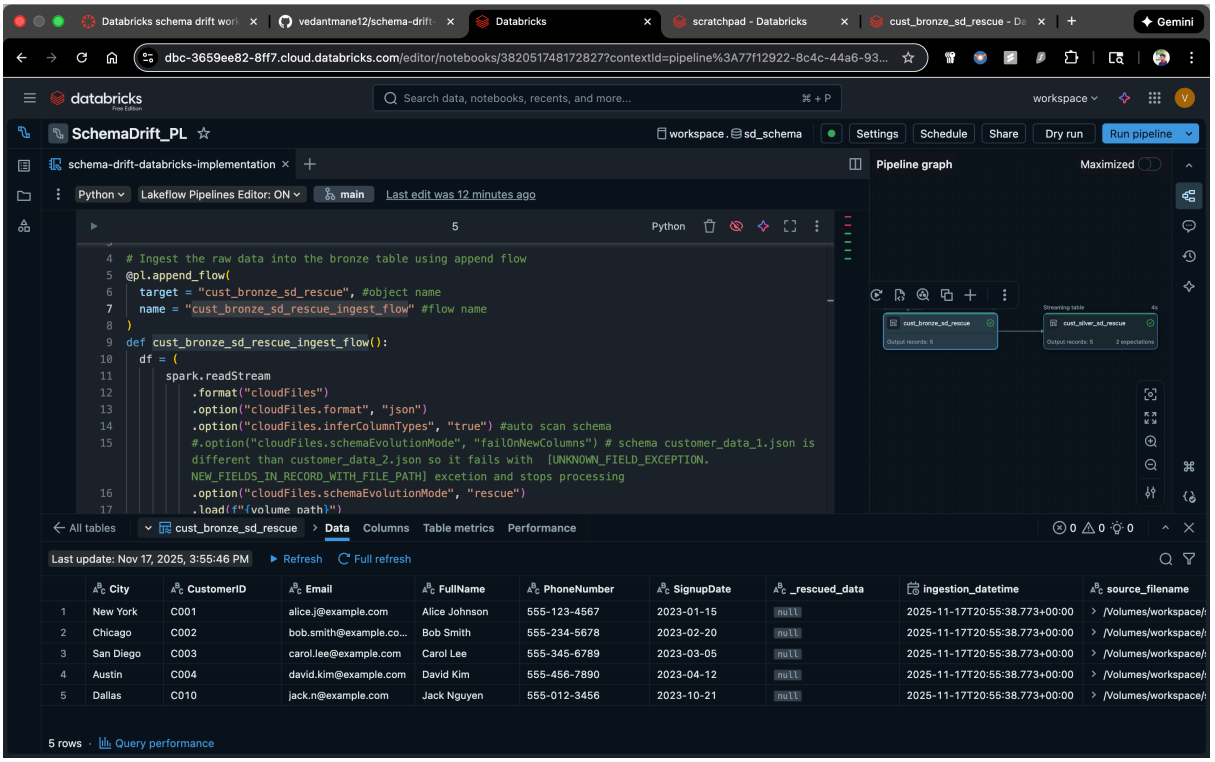


Pipeline Run: customer_data_1.json

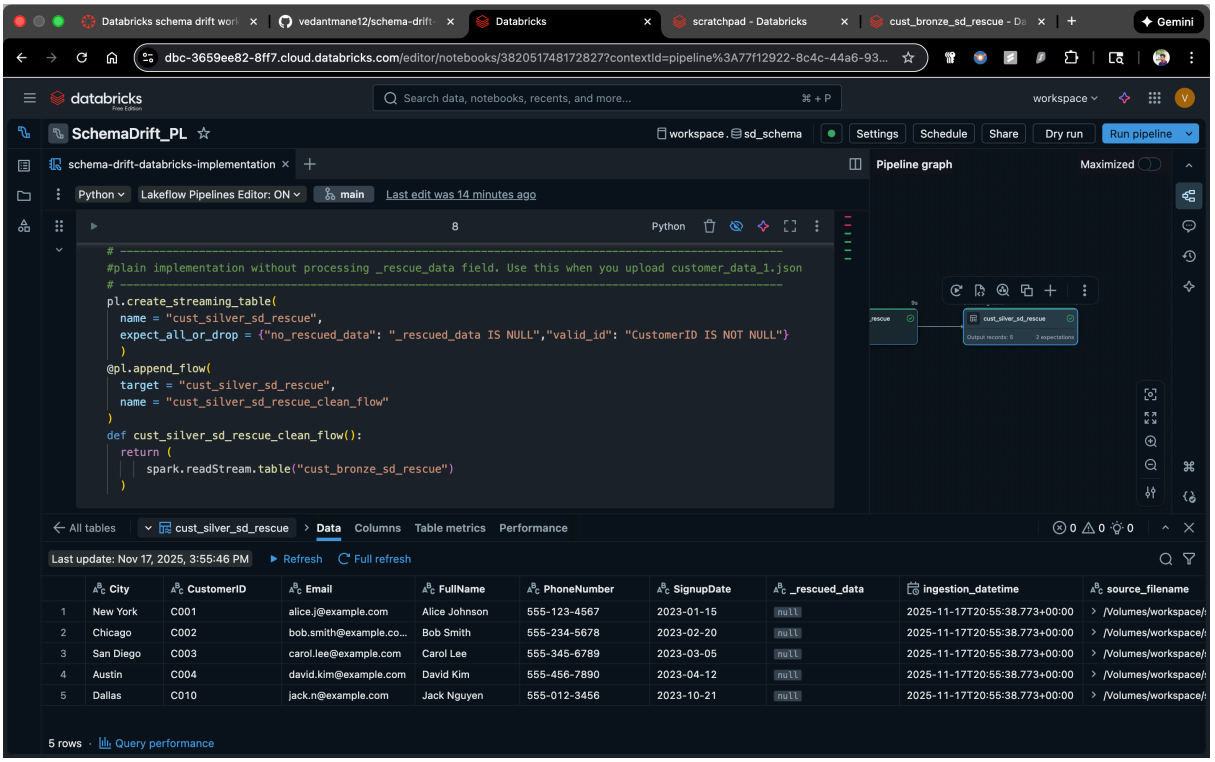
Volume: workspace.sd_schema.datastore



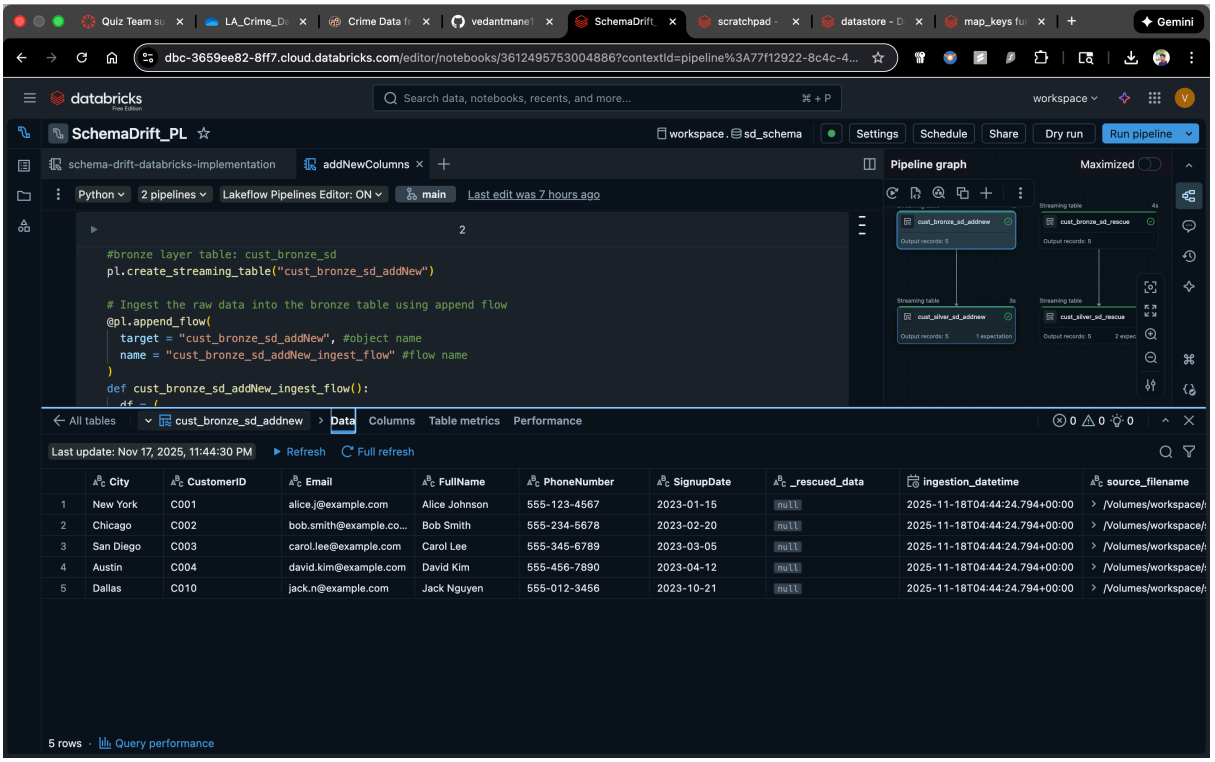
Bronze Layer Implementation (Rescue Data)



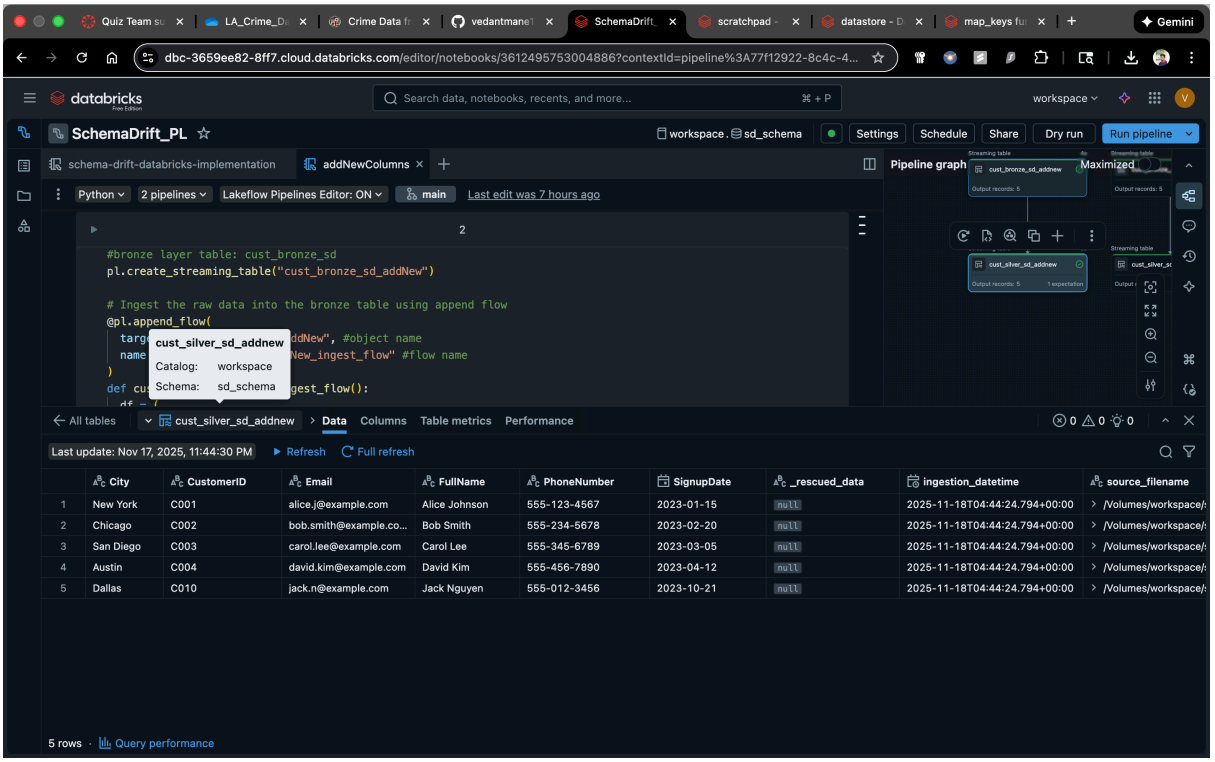
Silver Layer Implementation (Rescue Data)



Bronze Layer Implementation (AddNewColumns Data)

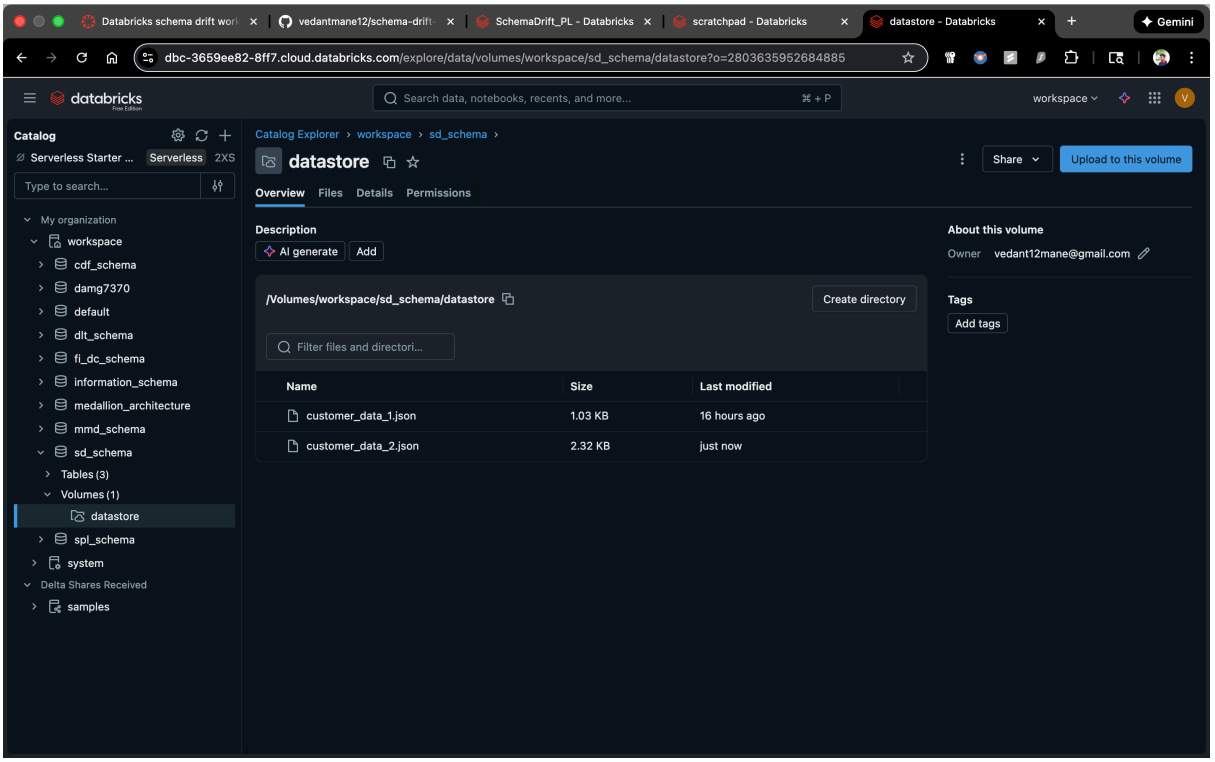


Silver Layer Implementation (AddNewColumns Data)

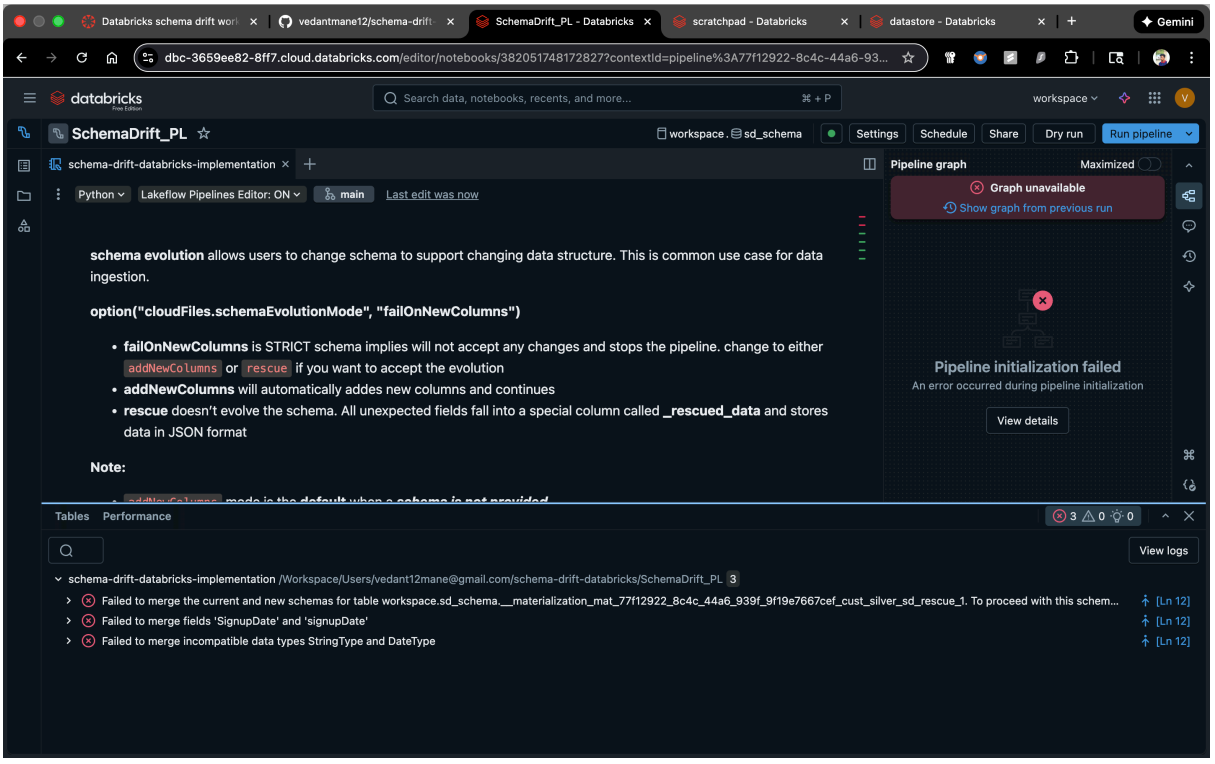


Pipeline Run: customer_data_2.json

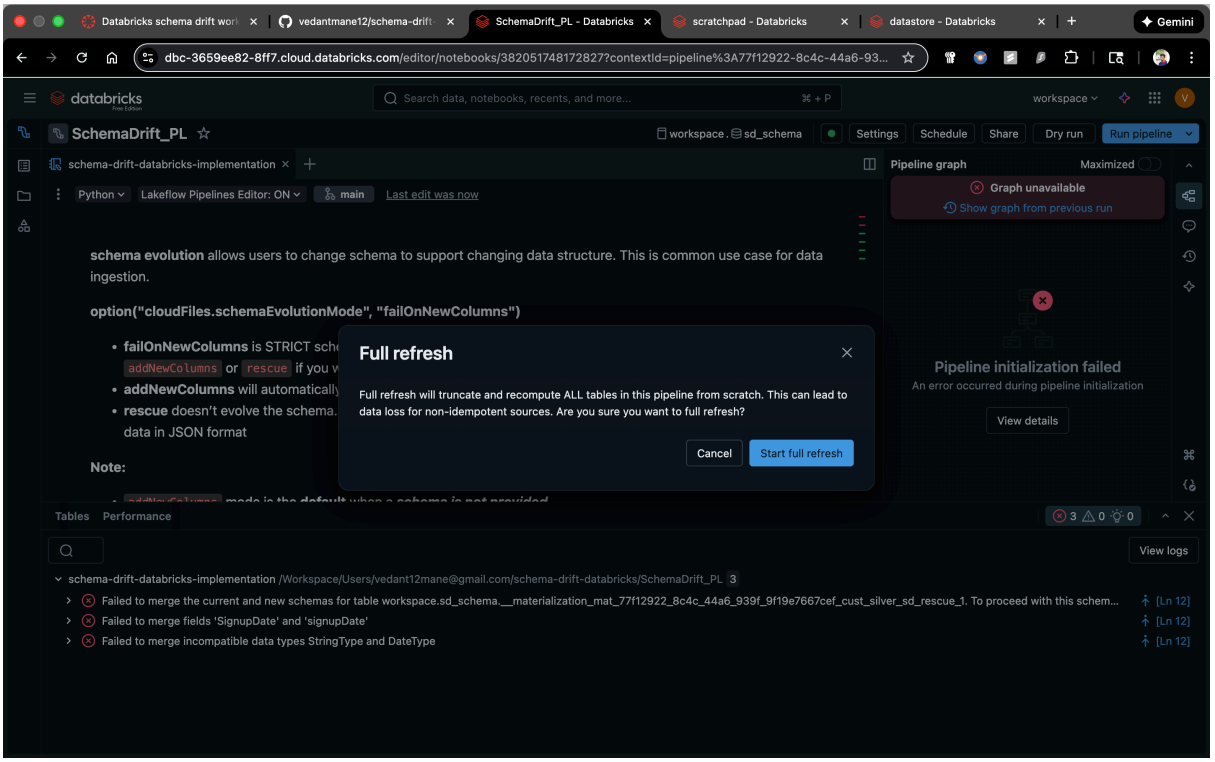
Volume: workspace.sd_schema.datastore

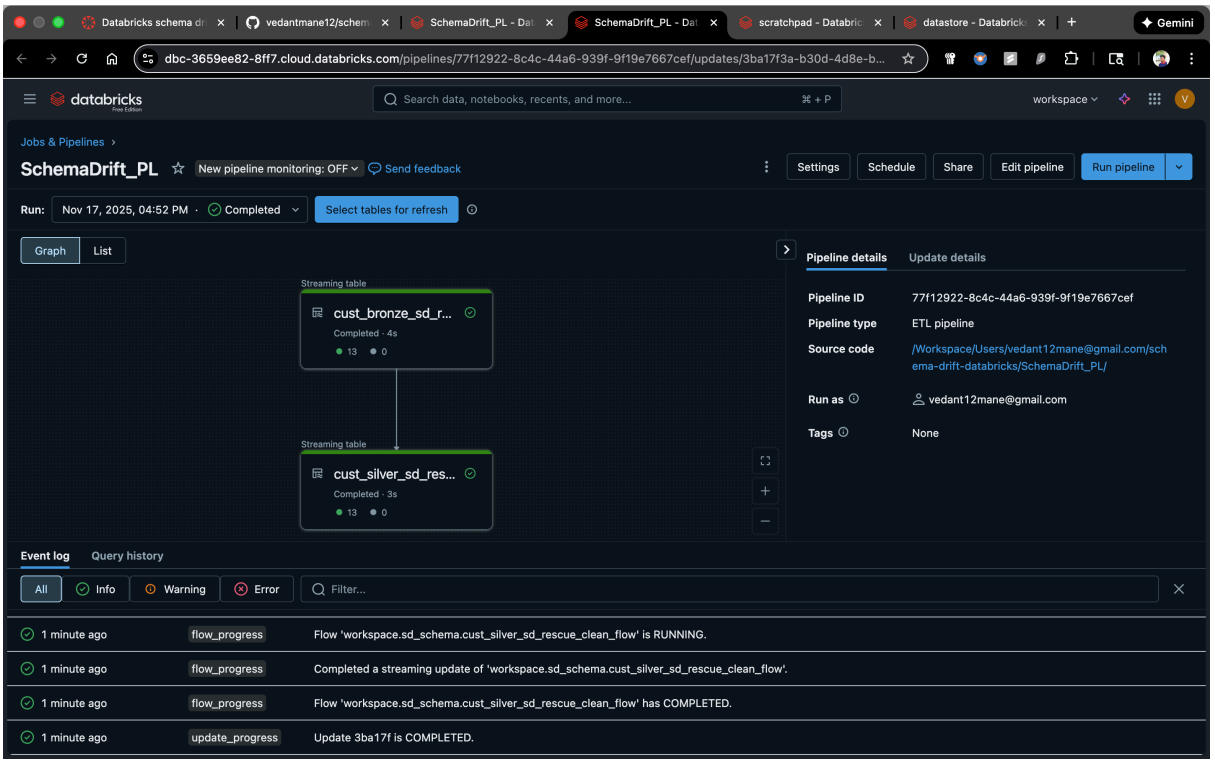


Pipeline Run: (Normal)

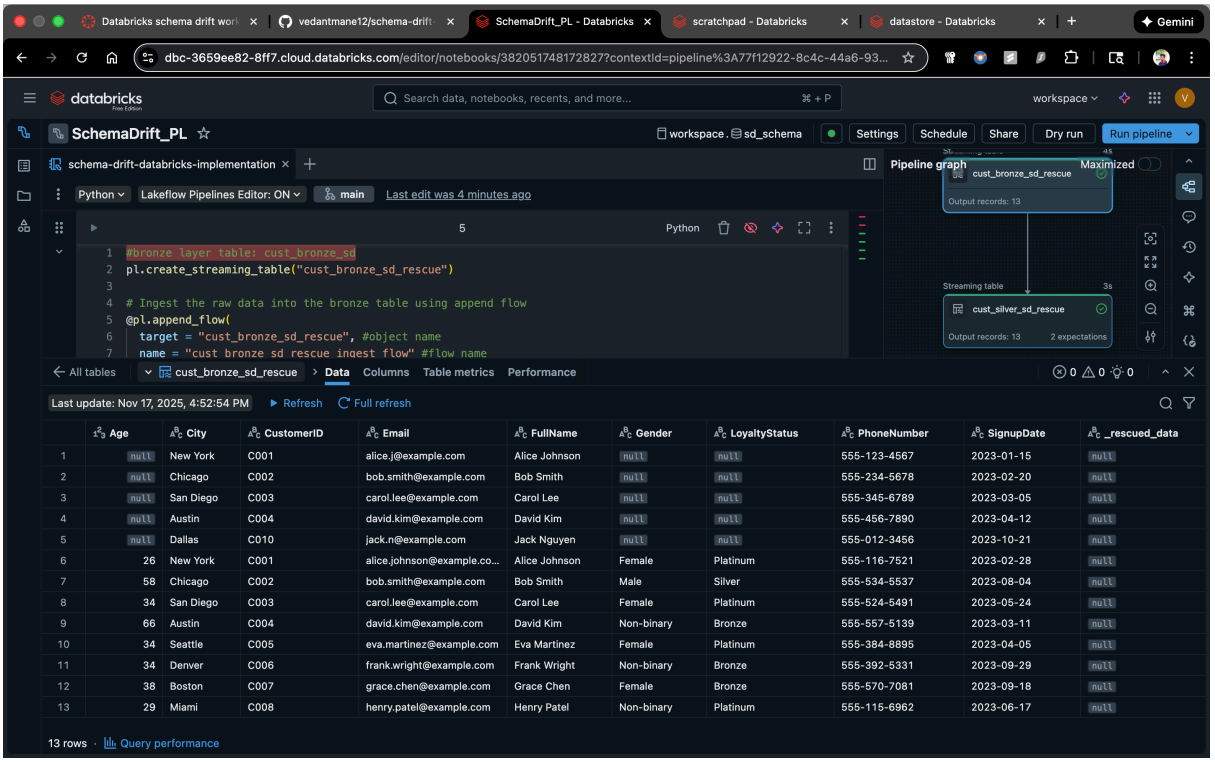


Pipeline Run: (Full Table Refresh)

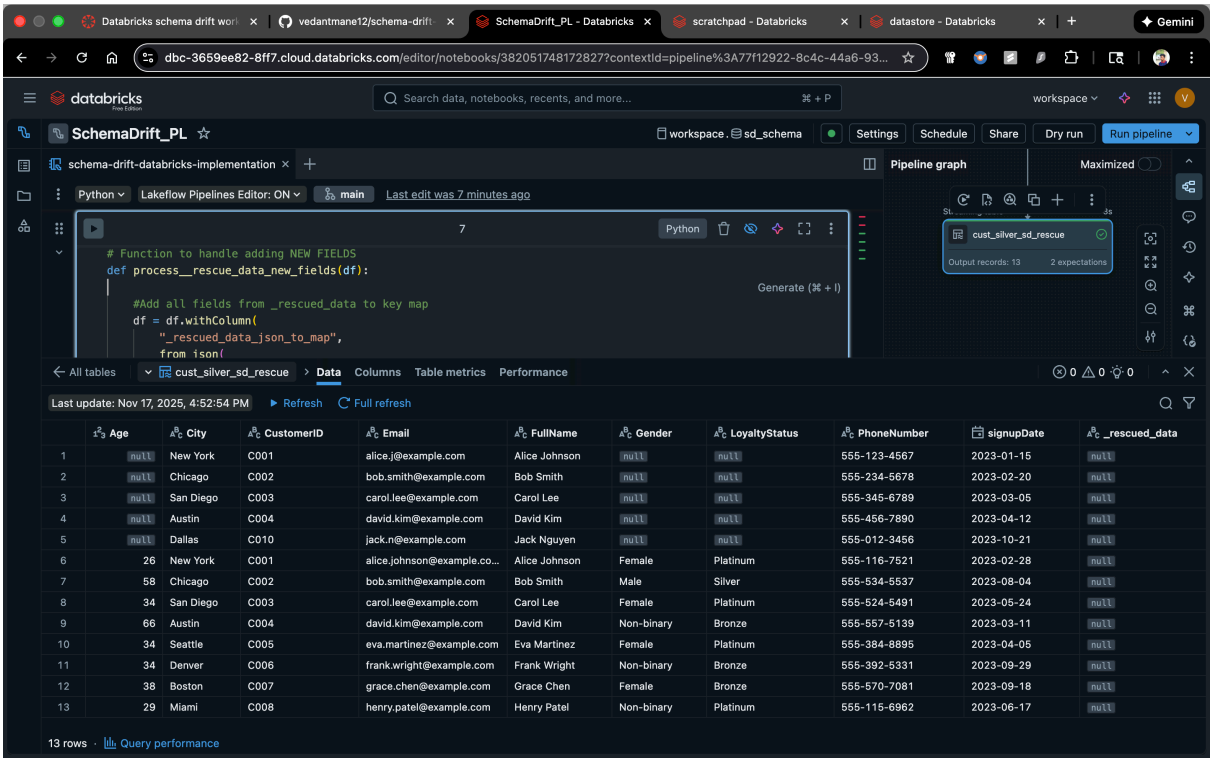




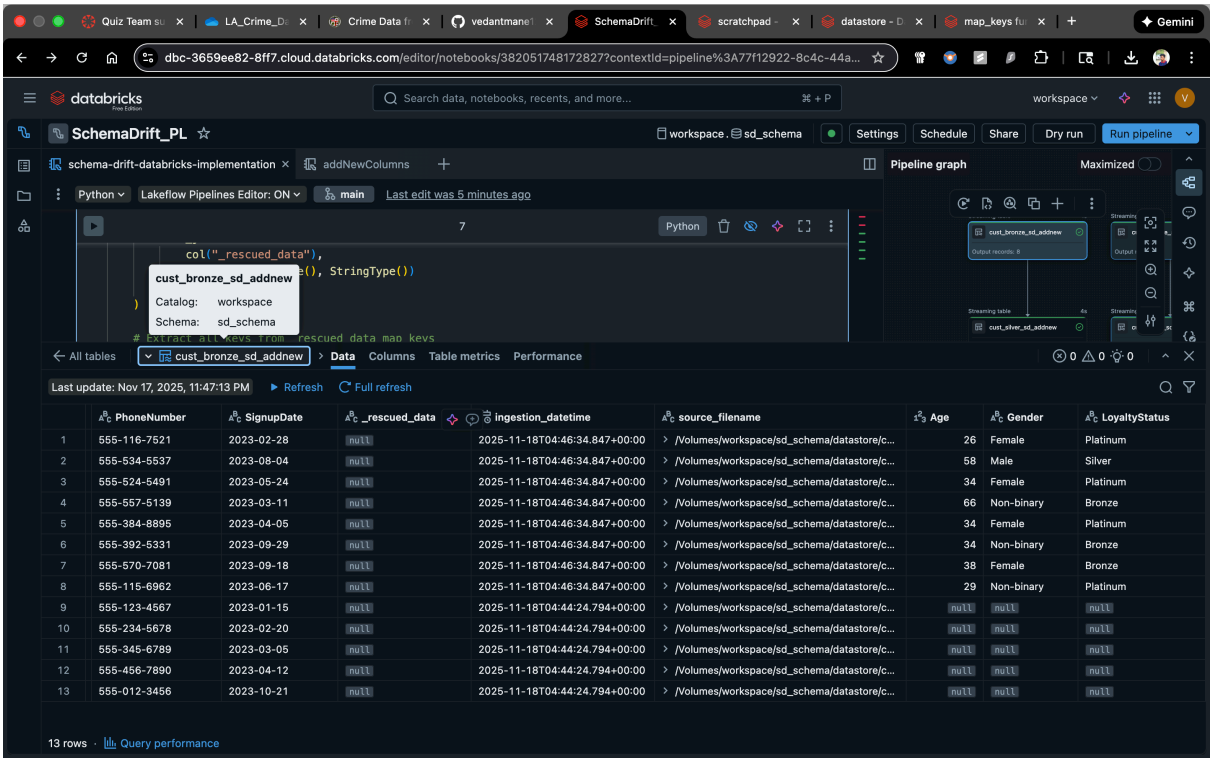
Bronze Layer Implementation (Rescue Data)



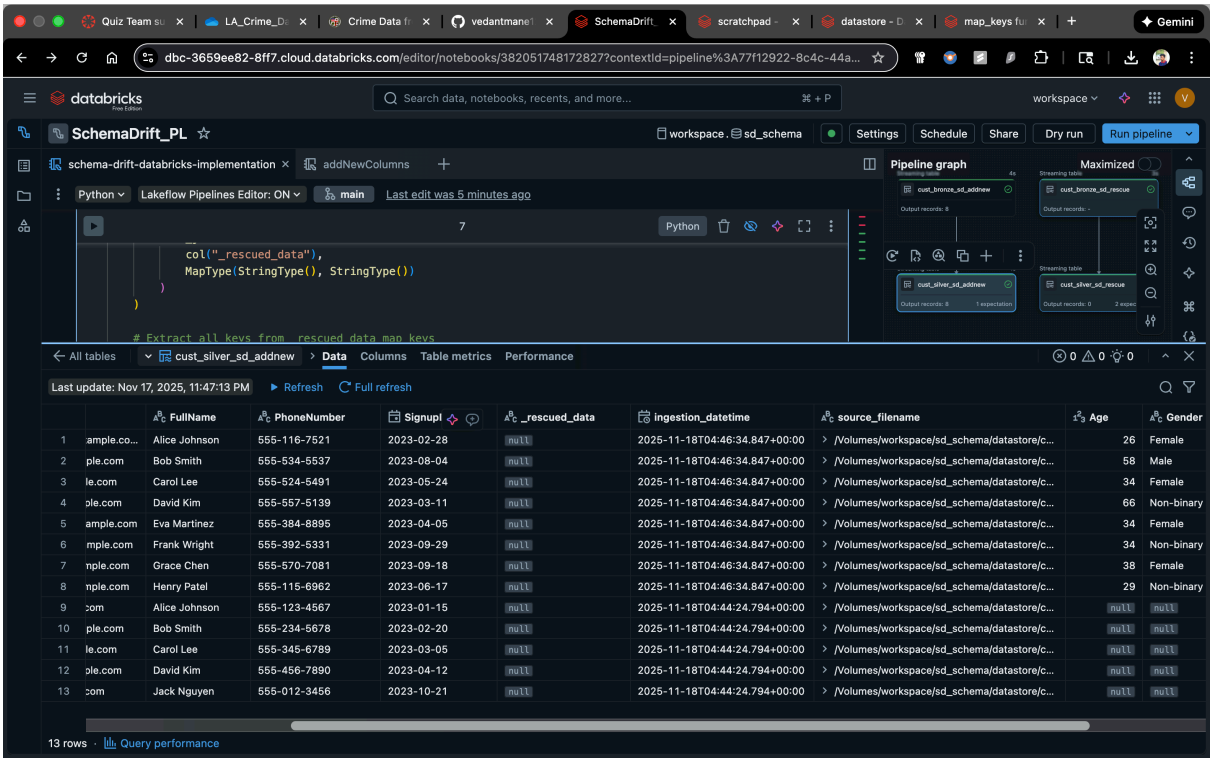
Silver Layer Implementation (Rescue Data)



Bronze Layer Implementation (AddNewColumns Data)

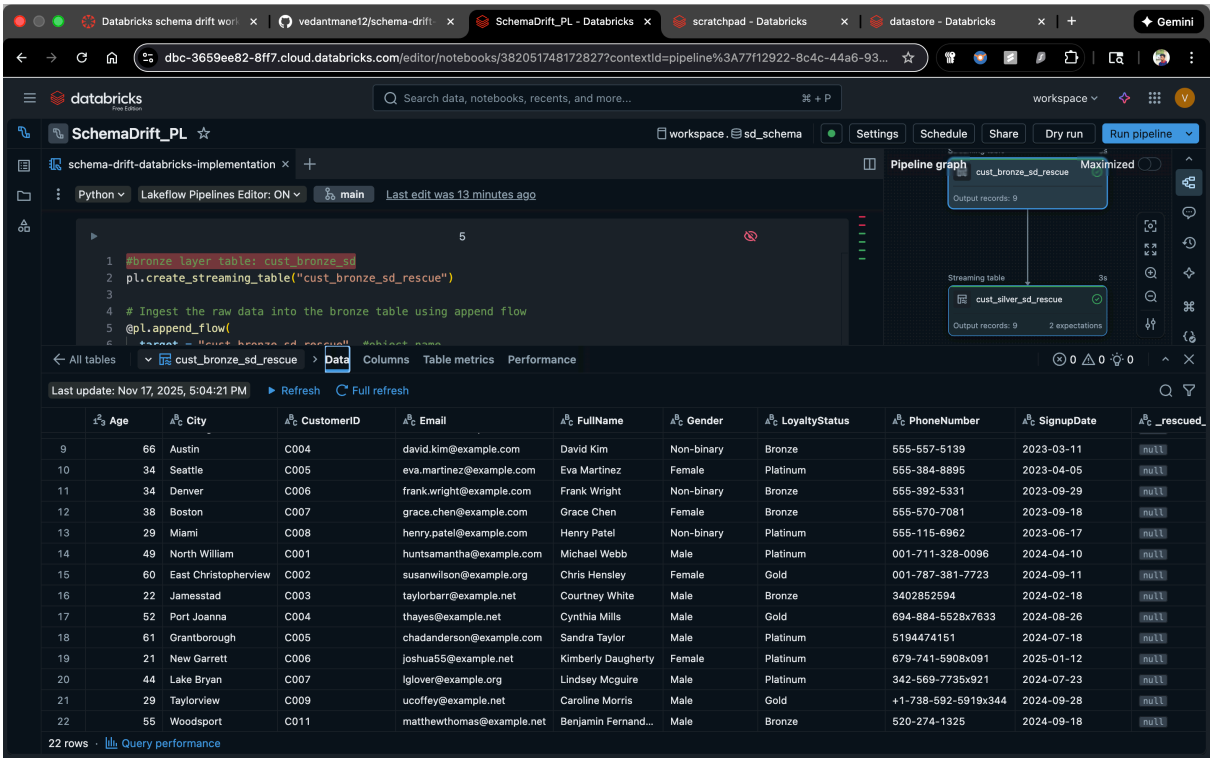


Silver Layer Implementation (AddNewColumns Data)

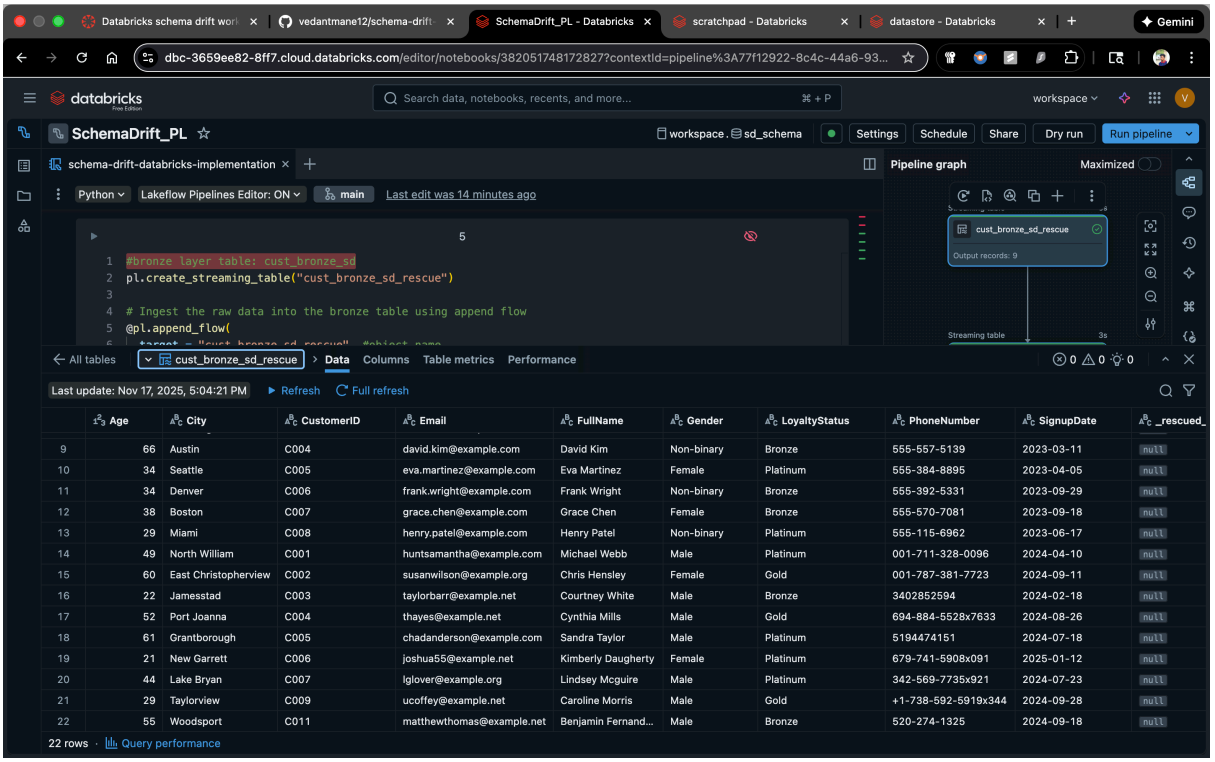


Pipeline Run: customer_data_3.json

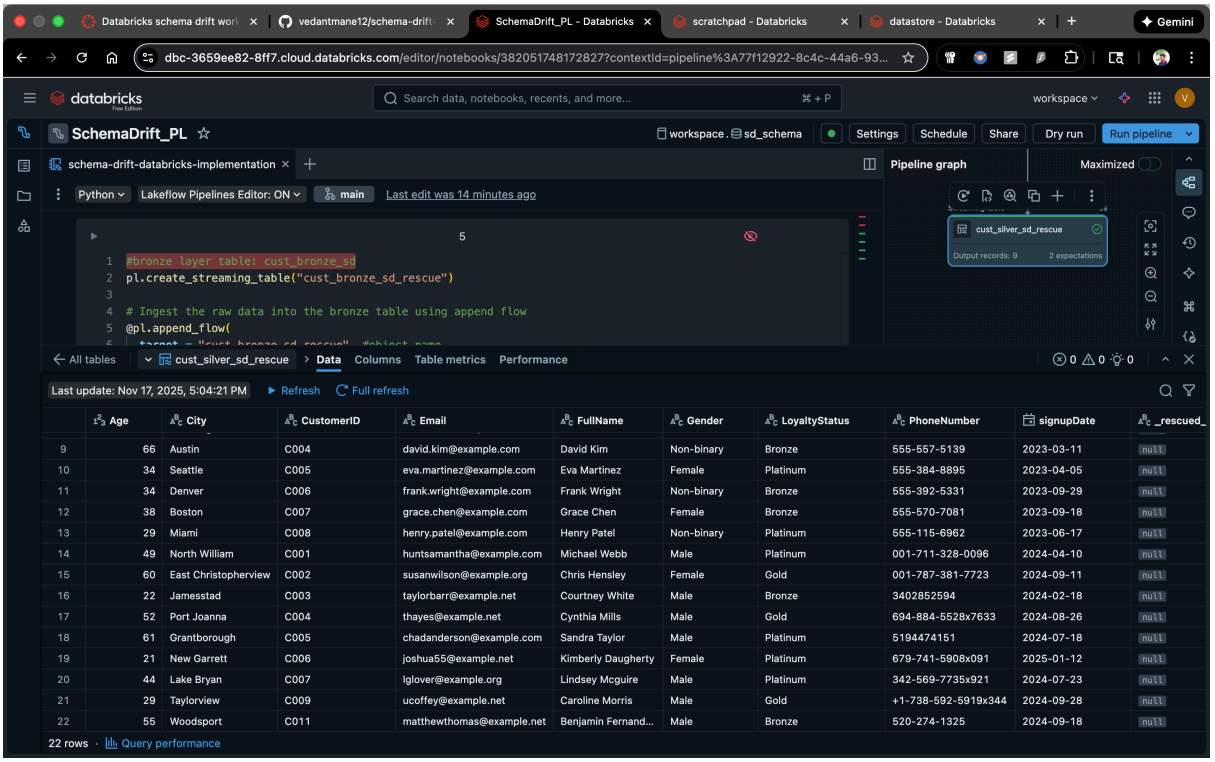
Volume: workspace.sd_schema.datastore



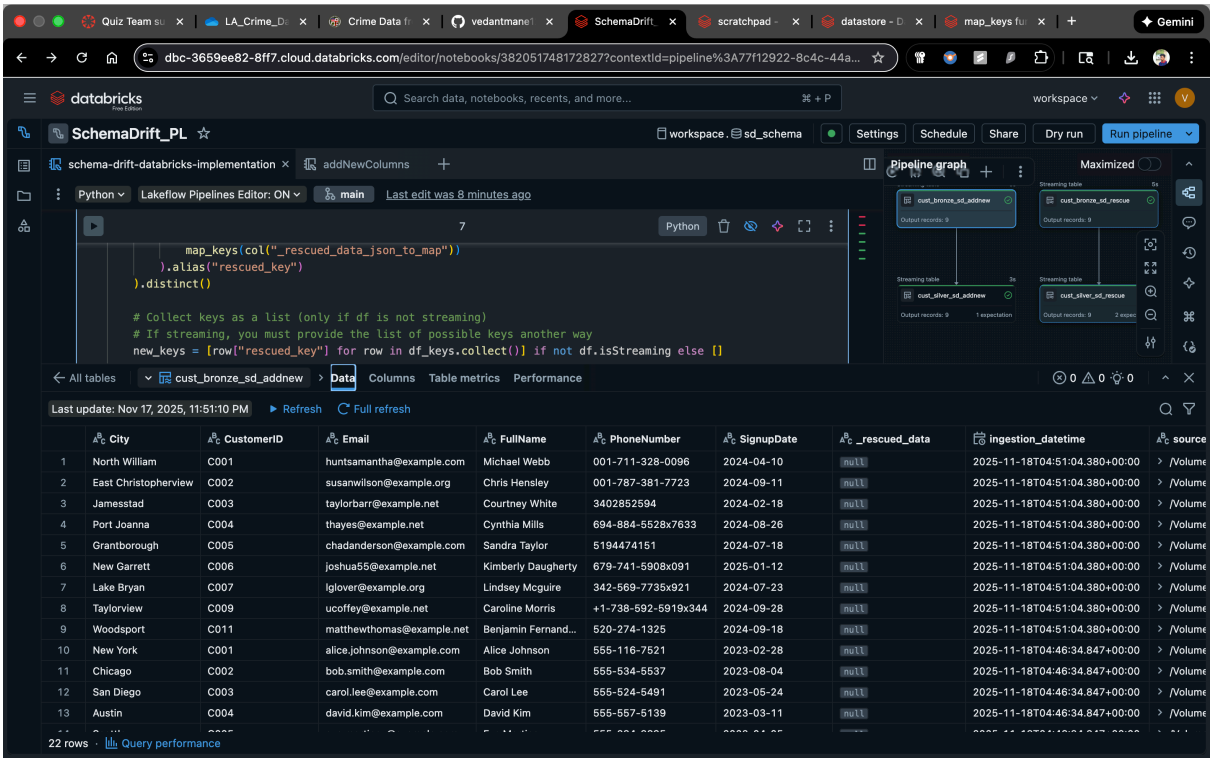
Bronze Layer Implementation (Rescue Data)



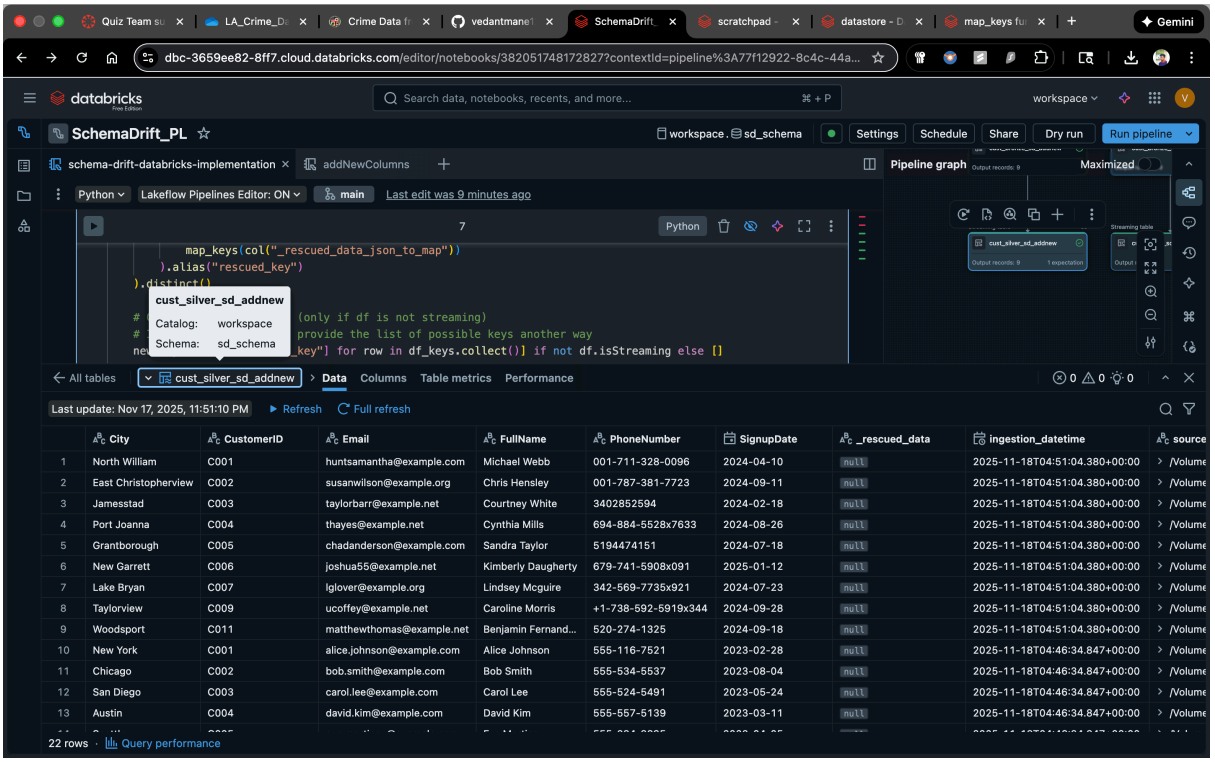
Silver Layer Implementation (Rescue Data)



Bronze Layer Implementation (AddNewColumns Data)

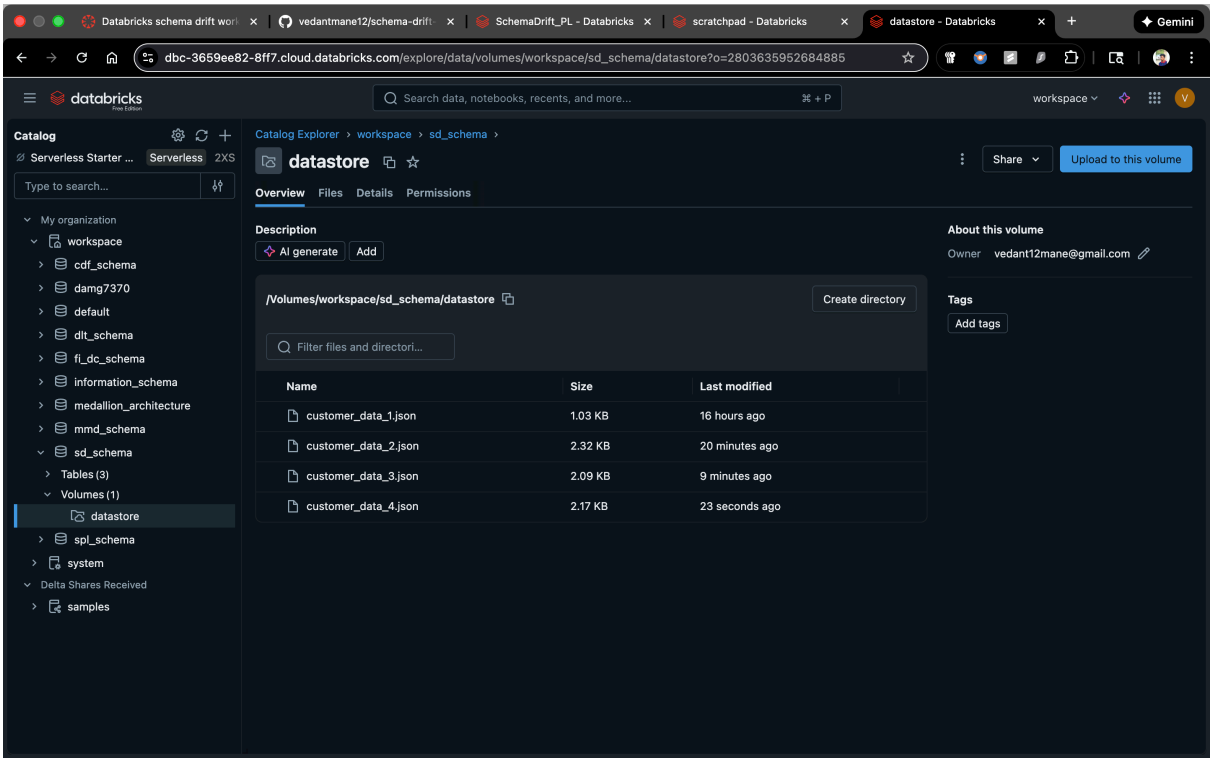


Silver Layer Implementation (AddNewColumns Data)

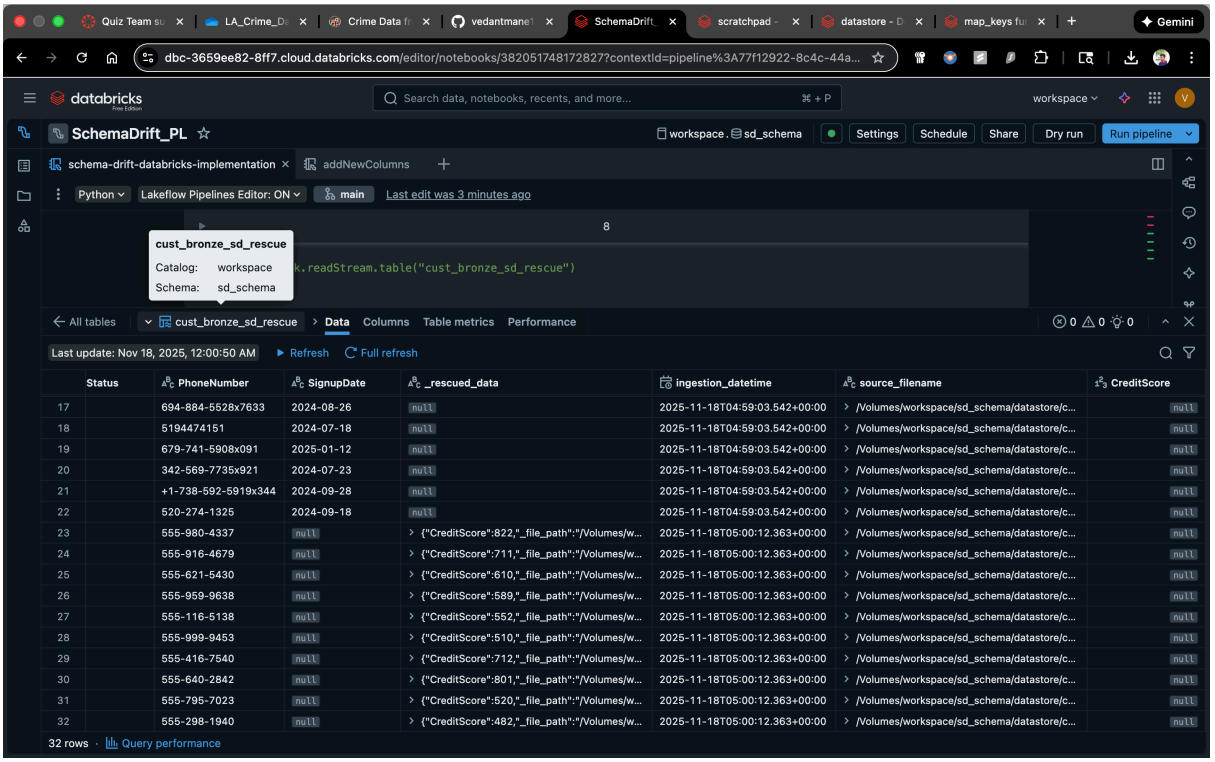


Pipeline Run: customer_data_4.json

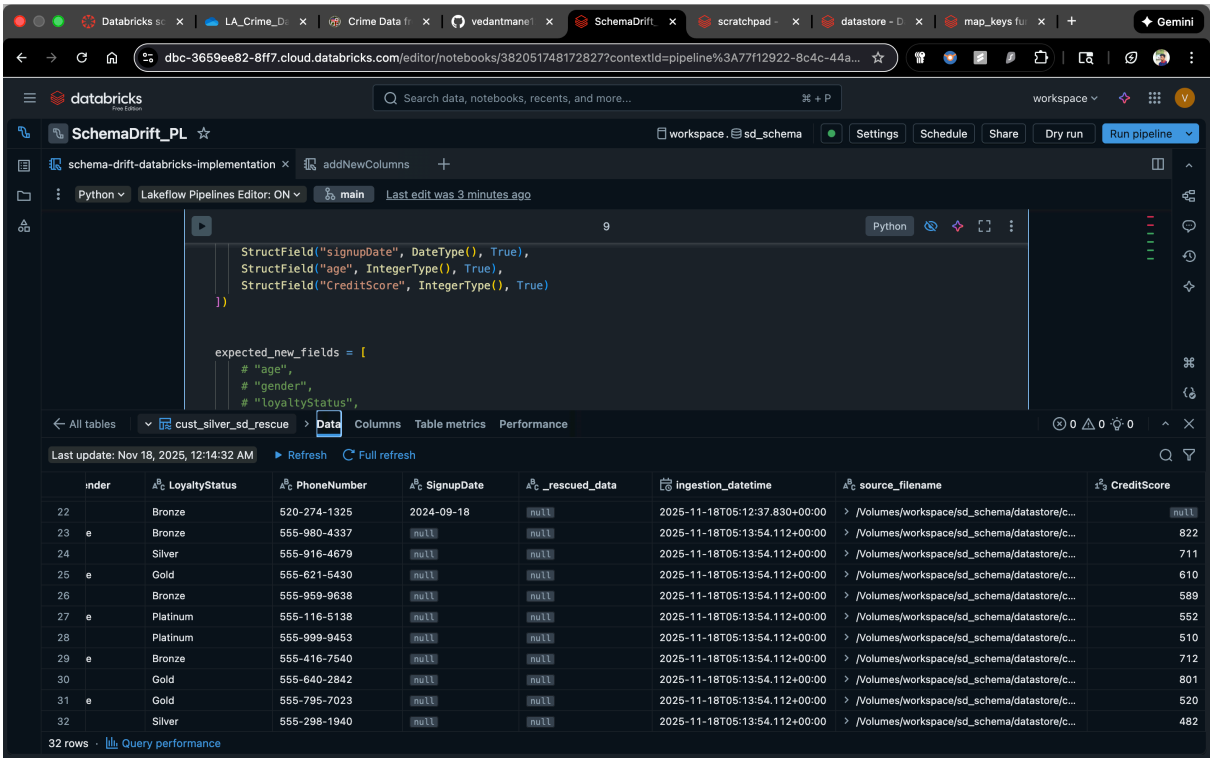
Volume: workspace.sd_schema.datastore



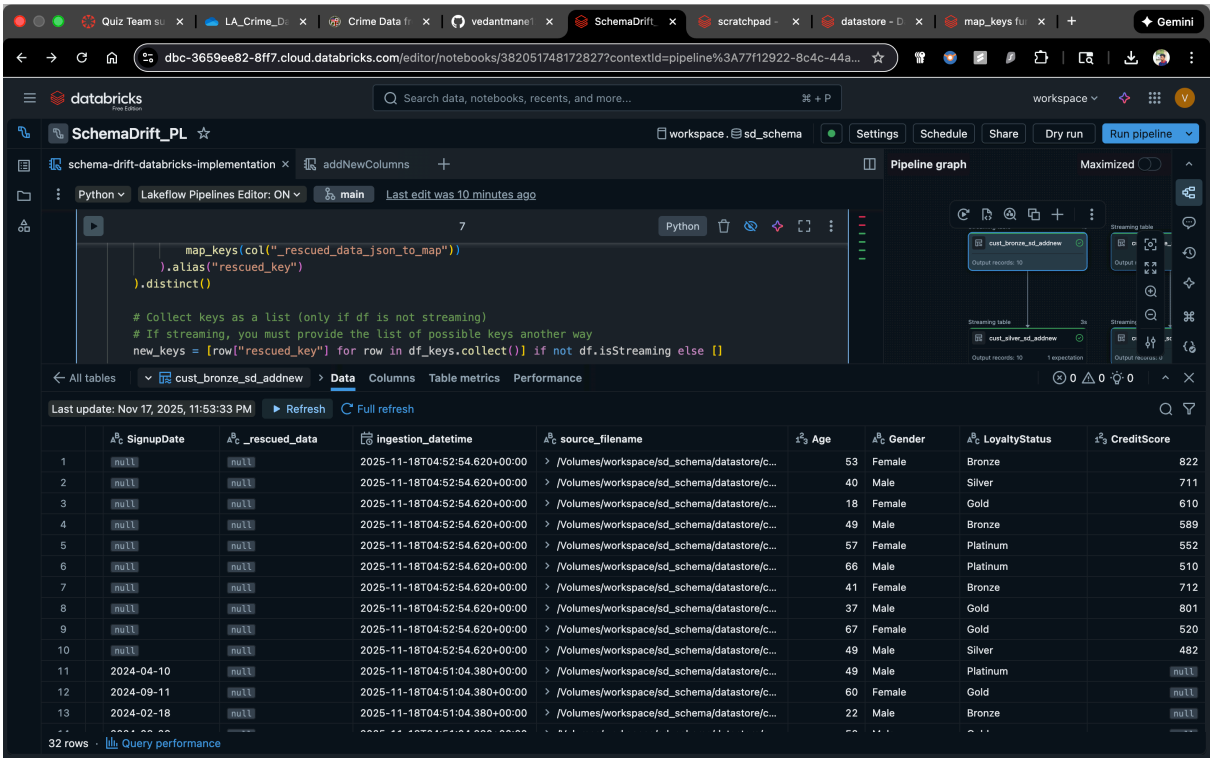
Bronze Layer Implementation (Rescue Data)



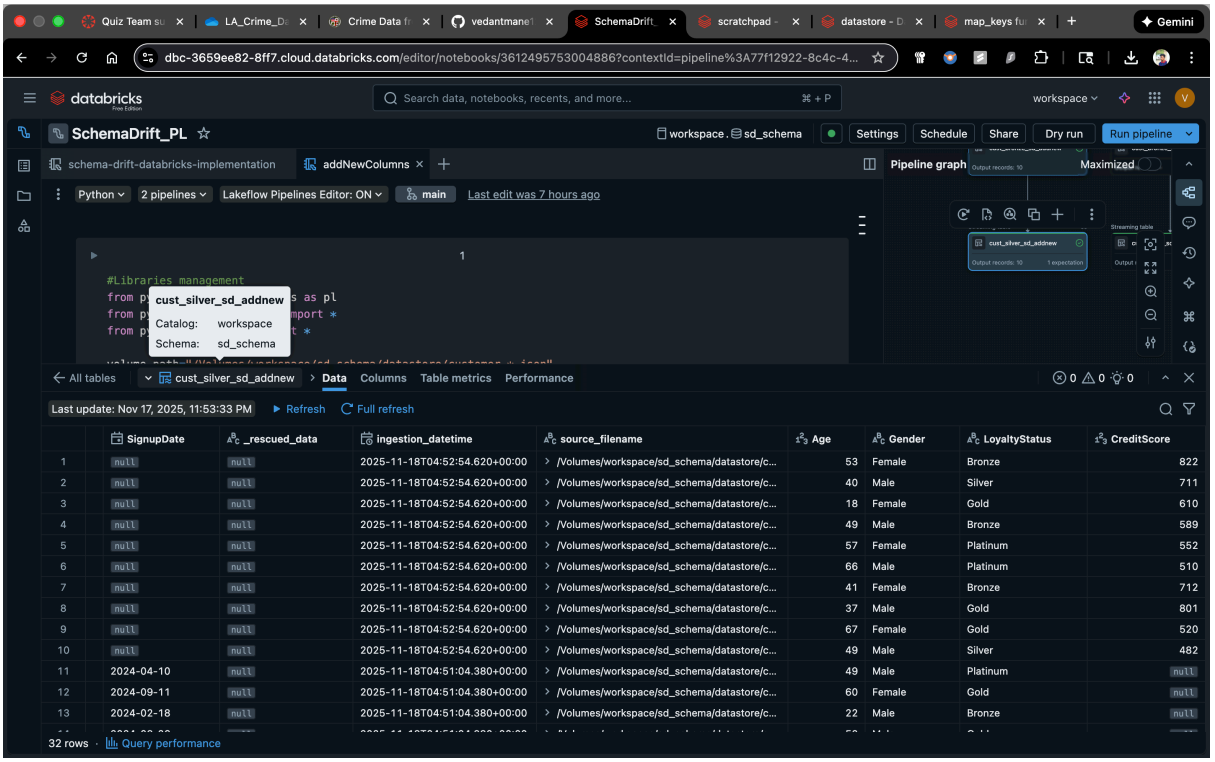
Silver Layer Implementation (Rescue Data)



Bronze Layer Implementation (AddNewColumns Data)



Silver Layer Implementation (AddNewColumns Data)



Part 1: Workshop

1.1 Understanding Schema Evolution

Schema evolution is a critical capability in modern data engineering that allows data pipelines to adapt to changing data structures without breaking. In real-world scenarios, source systems frequently add new fields,

modify data types, or restructure their output formats. A robust data pipeline must handle these changes gracefully.

Databricks provides three schema evolution modes through the `cloudFiles.schemaEvolutionMode` option:

1. **failOnNewColumns** - Strict mode that rejects any schema changes and stops the pipeline
2. **rescue** - Captures unexpected fields in a special `_rescued_data` column as JSON
3. **addNewColumns** - Automatically adds new columns to the table schema (default when no schema is provided)

1.2 Implementation with "rescue" Mode

Overview

The rescue mode implementation maintains a strict initial schema and captures any unexpected fields in a special `_rescued_data` column. This approach provides complete visibility into schema changes while maintaining strict data quality standards.

Bronze Layer Configuration

The bronze layer was configured to read streaming JSON files from the volume path using Auto Loader (cloudFiles). The key configuration parameter `cloudFiles.schemaEvolutionMode` was set to "rescue", which means any fields not present in the initial schema would be captured in the `_rescued_data` column rather than being added as new columns.

Additional metadata columns were added:

- **ingestion_datetime**: Timestamp of when the record was ingested
- **source_filename**: The file path from which the record originated

This metadata is crucial for tracking data lineage and troubleshooting data quality issues.

Silver Layer Configuration

The silver layer applied data quality expectations using DLT's `expect_all_or_drop` feature. Two critical expectations were implemented:

1. **no_rescued_data**: This expectation enforces that `_rescued_data IS NULL`, meaning no unexpected fields should be present. This is a strict quality check that will drop any rows containing rescued data.
2. **valid_id**: This expectation validates that `CustomerID IS NOT NULL`, ensuring every record has a valid identifier.

The combination of these expectations creates a strict data quality boundary - only records that match the expected schema and have valid identifiers will pass through to the silver layer.

Processing Rescued Data

Two custom functions were implemented to handle rescued data:

Function 1: process__rescue_data_new_fields()
This function extracts new fields from the `_rescued_data` JSON column and adds them as proper columns to the dataframe. It converts the JSON string into a map structure, then iterates through expected fields to extract values. If a field already exists in the dataframe but has a value in rescued_data, it uses the rescued value (handling the case where a field was added after initial processing).

Function 2: process__rescue_data_datatype_change()
This function handles data type changes for existing columns. When a column's data type changes in the source (for example, SignupDate changing from String to Date), the new values appear in `_rescued_data`. This function checks for each column whether it has a value in rescued_data, and if so, casts it to the appropriate target data type. After processing, it sets `_rescued_data` back to NULL to indicate the data has been properly handled.

Workflow with Rescue Mode

The typical workflow when using rescue mode involves:

1. **Initial Load**: Upload customer_data_1.json with base schema - pipeline succeeds
2. **Schema Change Detection**: Upload customer_data_2.json with new fields (Age, Gender, LoyaltyStatus) - new fields go into `_rescued_data`
3. **Pipeline Failure**: The expectation `_rescued_data IS NULL` causes the pipeline to fail because rescued data is present
4. **Full Table Refresh**: Must perform a "Full Table Refresh" to restructure the tables
5. **Code Update**: Uncomment the processing functions in the silver layer to handle rescued data
6. **Successful Processing**: After refresh, the functions extract fields from `_rescued_data` and the pipeline succeeds

1.3 Implementation with "addNewColumns" Mode

Overview

The addNewColumns mode provides automatic schema evolution by adding new columns to the table schema as they appear in the source data. This mode is more flexible and requires less manual intervention compared to rescue mode.

Bronze Layer Configuration

The bronze layer configuration was similar to rescue mode, but with `cloudFiles.schemaEvolutionMode` set to "addNewColumns". This setting tells Auto Loader to automatically add any new columns it discovers to the table schema. The same metadata columns (ingestion_datetime and source_filename) were added for consistency.

Silver Layer Configuration

The silver layer with addNewColumns mode uses a simpler approach. Instead of strict expectations about rescued data, it only validates that `CustomerID IS NOT NULL`. The transformation logic includes conditional data type casting:

Conditional Type Casting Logic:
The silver layer checks for the presence of specific columns before applying type conversions. This conditional approach is necessary because:

- Not all source files may contain all columns
- Schema evolves over time as new files are uploaded
- Applying type casting to non-existent columns would cause errors

The implemented type conversions were:

- **SignupDate**: Cast from String to DateType for proper date operations
- **Age**: Cast from String/inferred type to IntegerType for numeric operations
- **CreditScore**: Cast from String/inferred type to IntegerType (this was the bug fix)

Workflow with AddNewColumns Mode

The workflow is much simpler:

1. **Initial Load**: Upload customer_data_1.json - schema established with 6 base columns
2. **Automatic Evolution**: Upload customer_data_2.json - Age, Gender, LoyaltyStatus automatically added as columns
3. **Continued Processing**: Upload customer_data_3.json - no schema changes, data flows normally
4. **Final Evolution**: Upload customer_data_4.json - CreditScore added (after fix)

No pipeline failures occur, and no full table refresh is needed for new columns (only for data type changes of existing columns).

1.4 Detailed Observations and Behavioral Differences

Observation 1: Pipeline Stability

Rescue Mode:
When customer_data_2.json was uploaded, the pipeline immediately failed because the expectation `_rescued_data IS NULL` was violated. The new fields (Age, Gender, LoyaltyStatus) were captured in the `_rescued_data` column, causing all rows from this file to be dropped. This failure is by design - it alerts data engineers that the schema has changed and requires attention.

The failure provides strong governance but disrupts data flow. In a production environment, this means data stops flowing until the issue is addressed, which could impact downstream processes and analytics.

AddNewColumns Mode:
When customer_data_2.json was uploaded, the pipeline continued running without any failures. The new columns were automatically added to both bronze and silver tables. Data continued to flow seamlessly, with NULL values appearing in the new columns for records from earlier files.

This automatic handling provides operational continuity but may allow unexpected schema changes to propagate through the pipeline without immediate visibility.

Observation 2: Schema Change Handling

Rescue Mode:
All schema changes are captured in `_rescued_data` as a JSON string. This creates a complete audit trail of what fields were unexpected. However, this data is not immediately usable - it requires custom parsing logic to extract and properly type the fields.

The format in `_rescued_data` looks like:

`{"Age":"26","Gender":"Female","LoyaltyStatus":"Platinum"}`

To make this data usable, we need to:

1. Parse the JSON string into a map
2. Extract each field individually
3. Cast to appropriate data types
4. Handle cases where both the regular column and rescued_data contain values

AddNewColumns Mode:
Schema changes result in actual new columns being added to the table. Each new field becomes a first-class column with proper typing (based on Auto Loader's inference). For existing records that don't have these fields, the values are NULL.

This creates a sparse table structure where newer columns have many NULL values for older records, but the data is immediately queryable without any parsing.

Observation 3: Data Type Changes

Rescue Mode:
When a column's data type needs to change (like SignupDate going from String to Date), rescue mode can capture the new values in `_rescued_data`. The custom function `process_rescue_data_datatype_change()` handles this by:

1. Checking if the column exists in `_rescued_data`
2. Casting the rescued value to the target type
3. Using the rescued value if present, otherwise using the original column value

This allows for gradual data type migration, but requires a full table refresh to apply the change to the entire table.

AddNewColumns Mode:
Data type changes cannot be handled automatically with addNewColumns mode. If a column needs to change type, you must:

1. Perform a full table refresh
2. Update the silver layer transformation logic to cast the column

3. Reprocess all historical data with the new type

The conditional casting approach in the silver layer (checking `if "Age" in df.columns` before casting) ensures the code doesn't fail when processing older data that might not have the column yet.

Observation 4: Code Complexity

Rescue Mode:
The implementation requires significantly more code:

- Two custom functions (200+ lines) to process rescued data
- Complex JSON parsing logic
- Map extraction and type casting
- Careful handling of edge cases where both regular column and rescued_data contain values

This complexity increases maintenance burden and the potential for bugs. However, it also provides fine-grained control over how schema changes are handled.

AddNewColumns Mode:
The implementation is straightforward:

- Simple conditional checks for column existence
- Direct type casting without parsing
- Minimal custom logic (about 20 lines)

The simplicity reduces maintenance overhead and makes the code easier to understand and debug.

Observation 5: Full Refresh Requirements

Rescue Mode:
Full table refresh is required whenever:

- Schema changes are detected (to restructure tables to accommodate rescued data)
- Data type changes occur
- Processing logic is updated

The refresh requirement means:

- Entire bronze and silver tables are reprocessed
- Downtime during refresh operation
- Computational cost of reprocessing all historical data

AddNewColumns Mode:
Full table refresh is only required when:

- Data type changes occur for existing columns
- Processing logic fundamentally changes

New columns are added automatically without refresh, which means:

- No downtime for additive schema changes
- Lower computational cost
- Faster adaptation to schema changes

However, when a refresh IS needed (for type changes), both modes require the same level of effort.

Observation 6: Data Quality Control

Rescue Mode:
Provides strongest data quality control:

- Explicit approval needed for schema changes (via code updates)

- Complete visibility into unexpected fields
- Ability to reject or quarantine non-conforming data
- Audit trail of all schema deviations

The expectation `_rescued_data IS NULL` acts as a circuit breaker that stops problematic data from flowing through the pipeline.

AddNewColumns Mode:
Provides weaker data quality control:

- Schema changes are accepted automatically
- Unexpected columns are added without explicit approval
- No built-in mechanism to alert on schema changes
- Relies on downstream validation to catch issues

However, the expectation `CustomerID IS NOT NULL` still provides basic data quality validation.

Observation 7: Storage and Performance

Rescue Mode:

- Bronze table has fixed column count with `_rescued_data` containing JSON
- JSON storage is less efficient than native column storage
- Queries against rescued data require JSON parsing functions
- Silver table structure changes after processing rescued data

Storage pattern:

```
Initial: 8 columns (6 business + 2 metadata)
After schema change: 8 columns (6 business + _rescued_data + 2 metadata)
After processing: 11 columns (9 business + _rescued_data [NULL] + 2 metadata)
```

AddNewColumns Mode:

- Column count increases with each schema change
- Native column storage is more efficient
- Direct column queries without parsing
- Sparse table structure with many NULLs

Storage pattern:

```
Initial: 8 columns (6 business + 2 metadata)
After schema change: 11 columns (9 business + 2 metadata)
After another change: 12 columns (10 business + 2 metadata)
```

Observation 8: Use Case Suitability

Rescue Mode is ideal for:

- Highly regulated industries (finance, healthcare) requiring strict data governance
- Scenarios where schema changes must be reviewed and approved
- Data contracts with external providers where schemas should be stable
- Audit and compliance requirements needing complete change tracking
- Production systems where unexpected schema changes indicate upstream problems

AddNewColumns Mode is ideal for:

- Agile development environments with frequent iterations
- IoT or sensor data where new metrics are regularly added
- Semi-structured data sources with variable schemas

- Rapid prototyping and experimentation
- Data lakes ingesting data from multiple evolving sources

1.5 File-by-File Schema Evolution Timeline

customer_data_1.json (Base Schema)

This file established the initial schema with 6 business columns:

- CustomerID: Unique identifier for each customer
- FullName: Customer's full name
- Email: Contact email address
- PhoneNumber: Contact phone number
- City: Customer's city of residence
- SignupDate: Date when customer signed up (String format)

Plus 2 metadata columns:

- ingestion_datetime: System timestamp
- source_filename: Source file path

Total columns in Bronze: 8Total rows: 5 customers

This file processed successfully in both rescue and addNewColumns modes as it defined the baseline schema.

customer_data_2.json (First Schema Evolution)

This file introduced 3 new fields while retaining all original fields:

- Age: Customer's age (Integer)
- Gender: Customer's gender (String: Male/Female/Non-binary)
- LoyaltyStatus: Customer tier (String: Bronze/Silver/Gold/Platinum)

Additionally, some data values changed for existing customers (C001-C004), demonstrating that schema evolution often accompanies data updates.

New columns added: 3Total columns in Bronze (addNewColumns): 11Total rows added: 8 (includes 3 new customers C005-C007, C008)

Rescue Mode Behavior:

- New fields captured in `_rescued_data`: `{"Age":"26","Gender":"Female","LoyaltyStatus":"Platinum"}`
- Pipeline failed due to expectation violation
- Required full table refresh and code update to process

AddNewColumns Mode Behavior:

- Age, Gender, LoyaltyStatus automatically added as columns
- Old records (from customer_data_1.json) show NULL for these columns
- New records have values populated
- Pipeline continued without interruption

customer_data_3.json (Data Updates, No Schema Changes)

This file maintained the same schema as customer_data_2.json but with significant data updates:

- Updated values for existing customers (C001-C009)
- Added 1 new customer (C011)
- Demonstrated that not every file upload needs schema changes

Schema changes: NoneTotal columns remain: 11 (in addNewColumns mode)Total rows added: 9

Both Modes Behavior:

- Processed seamlessly in both modes
- No schema evolution logic triggered
- Data simply appended/updated based on streaming semantics
- Demonstrated stable operation when schema doesn't change

This file validated that both approaches handle normal data ingestion efficiently when schema is stable.

customer_data_4.json (Second Schema Evolution + Field Removal)

This file introduced interesting changes:

- Added new field: CreditScore (Integer) - customer's credit score
- Removed field: SignupDate no longer present in new records
- Updated values for all existing customers (C001-C010)

New columns added: 1 (CreditScore)Total columns in Bronze (addNewColumns): 12Columns removed from new records: 1 (SignupDate)Total rows added: 10

Critical Observation about Field Removal:
SignupDate was NOT removed as a column from the table - it remained as a column but with NULL values for records from customer_data_4.json. This is expected behavior: schema evolution in Delta Lake is additive only. Once a column exists, it persists in the table schema even if future records don't populate it.

AddNewColumns Mode Behavior (Before Fix):

- CreditScore added to bronze layer successfully
- CreditScore did NOT appear in silver layer (BUG)
- Reason: Silver layer transformation lacked conditional check for CreditScore

AddNewColumns Mode Behavior (After Fix):

- CreditScore added to bronze layer
- CreditScore properly cast to IntegerType and appeared in silver layer
- Fix involved adding the conditional casting logic

Rescue Mode Behavior:

- CreditScore captured in `_rescued_data`
- Required updating the `expected_new_fields` list to include "CreditScore"
- Required full refresh to apply changes
- After processing, CreditScore extracted from rescued data and properly typed

1.6 Comprehensive Comparison Matrix

CriteriaRescue ModeAddNewColumns ModeAutomatic Schema EvolutionNo - manual intervention requiredYe s - fully automaticPipeline Failures on Schema ChangeYes - intentional failure for governanceNo - continues s eamlesslyNew Field HandlingStored in_rescued_data as JSONAdded as new native columnsData Type Chang e HandlingCustom function requiredCustom function requiredFull Table Refresh FrequencyEvery schema cha ngeOnly for data type changesCode ComplexityHigh (200+ lines of custom logic)Low (20 lines of conditional casts)Data Quality EnforcementVery strict - circuit breaker patternModerate - basic validation onlyAudit Trail Complete - all changes in_rescued_dataNone - changes happen silentlyQuery PerformanceSlower - requires JSON parsingFaster - native column accessStorage EfficiencyLower - JSON string overheadHigher - native c olumn storageOperational OverheadHigh - requires monitoring and interventionLow - mostly self-managingS chema GovernanceStrong - explicit approval neededWeak - automatic acceptanceDevelopment AgilitySlower - requires code updatesFaster - automatic adaptationProduction StabilityCan cause outages during schema c

hangesMaintains continuous operationDebugging ComplexityHigher - more code paths to traceLower - straightforward logicTesting RequirementsExtensive - test rescue data processingModerate - test type conversions Documentation NeedsHigh - complex logic to documentLow - simple logic to documentTeam Skill RequirementsAdvanced - JSON processing, complex PySparkIntermediate - basic PySparkCost (Compute)Higher - more processing for parsingLower - direct column operationsCost (Storage)Similar - JSON vs sparse columns trade-offSimilarRegulatory ComplianceExcellent - strong controls and audit trailFair - requires additional monitoringBest for Team SizeLarger teams with dedicated data engineersSmaller teams, self-service analytics

1.7 Key Insights and Recommendations

Insight 1: No Universal "Best" Approach

Neither mode is universally superior - the choice depends on organizational context:

Choose Rescue Mode when:

- Operating in regulated industries (financial services, healthcare, government)
- Data contracts are formal agreements requiring change management
- Schema changes are rare and should be exceptional events
- Team has advanced PySpark skills
- Audit and compliance requirements are stringent
- Budget allows for dedicated data engineering resources

Choose AddNewColumns Mode when:

- Operating in agile, fast-paced environments
- Source systems evolve rapidly
- Team prefers simplicity over strict control
- Self-service analytics is a priority
- Time-to-insight is critical
- Budget favors minimal operational overhead

Insight 2: Hybrid Approaches Are Possible

Organizations don't have to choose one mode for all pipelines. Consider:

- **Tier-based approach**: Use rescue mode for critical tier-1 data, addNewColumns for tier-2/3
- **Environment-based**: Rescue in production, addNewColumns in development
- **Source-based**: Rescue for external vendors, addNewColumns for internal systems
- **Maturity-based**: Start with addNewColumns, graduate to rescue as governance matures

Insight 3: The Real Challenge is Data Type Changes

Both modes struggle with data type changes for existing columns. Neither mode can automatically convert a column from String to Integer without a full table refresh. This is a fundamental limitation of Delta Lake's schema evolution.

Mitigation strategies:

- Define clear data type contracts upfront
- Use String types for flexibility (cast in silver/gold layers)
- Plan for periodic full refreshes during maintenance windows
- Consider creating new columns (e.g., Age_v2) rather than changing types

Insight 4: Metadata is Crucial

Both implementations included `ingestion_datetime` and `source_filename` metadata columns. This proved invaluable for:

- Debugging schema evolution issues
- Understanding data lineage
- Identifying which file introduced which schema changes
- Time-based analysis of data evolution

Recommendation: Always include comprehensive metadata regardless of schema evolution mode chosen.

Insight 5: Testing Schema Evolution is Complex

Schema evolution testing requires:

- Sequential file uploads in specific order
- Validation at each stage
- Checking both success and failure scenarios
- Verifying data type conversions
- Ensuring NULL handling is correct

This makes automated testing challenging. The assignment's sequential file approach (customer_data_1 → 2 → 3 → 4) provided a realistic testing scenario.

Part 2: Fixing the issue

2.1 Problem Identification

Issue Description

When `customer_data_4.json` was uploaded, the new field **CreditScore** was successfully appearing in the **bronze layer** table but was **completely missing** from the **silver layer** table.

Impact

- Critical data loss between bronze and silver layers
- Business users unable to access CreditScore for analytics
- Schema evolution not propagating through the pipeline as expected

Initial Verification

Bronze Layer Check:

- CreditScore column: ✅ Present
- Data values: 10 non-NULL records with scores ranging from 482 to 822
- Column type: String (inferred by Auto Loader)

Silver Layer Check:

- CreditScore column: ❌ Completely absent
- Total columns: 11 (should be 12)
- Data loss confirmed

2.2 Root Cause Analysis

Investigation Steps

Step 1: Bronze Layer Review
Confirmed that bronze layer was correctly configured with `cloudFiles.schemaEvolutionMode = "addNewColumns"`. Auto Loader

successfully detected and added the CreditScore column.

Step 2: Silver Layer Transformation Review
Discovered that the silver layer transformation function only had explicit handling for:

- SignupDate (cast to DateType)
- Age (cast to IntegerType)

Step 3: Identified the Gap
The transformation logic had **NO conditional check** for CreditScore, meaning it was read from bronze but never explicitly processed or passed through to the output.

Root Cause Explanation

The silver layer uses **explicit column transformation**. Only columns that are explicitly processed get propagated to the output dataframe. Since CreditScore was not included in any transformation logic, it was implicitly dropped during the DataFrame operations.

Key Insight: While `addNewColumns` mode automatically adds columns in bronze, it does NOT automatically propagate them through custom transformations in downstream layers. Each transformation layer needs explicit logic to handle new columns.

2.3 The Solution

Fix Applied

Added a conditional block to handle CreditScore in the silver layer transformation:

Logic Added:

- Check if "CreditScore" column exists in the bronze dataframe
- If present, cast it from String to IntegerType
- Include it in the output dataframe

Why This Fix Works

1. **Conditional Check Prevents Errors**: The `if "CreditScore" in df.columns` check ensures the code doesn't fail when processing older files (customer_data_1, 2, 3) that don't contain CreditScore
2. **Explicit Processing Ensures Propagation**: By explicitly casting CreditScore with `withColumn()`, we ensure it's included in the output dataframe written to silver table
3. **Type Casting Provides Data Quality**: Converting to IntegerType ensures CreditScore can be used in mathematical operations and aggregations
4. **Maintains Code Consistency**: Follows the same pattern as existing Age and SignupDate transformations

Before Fix

Silver layer transformation had:

- SignupDate handling ✅
- Age handling ✅
- CreditScore handling ❌ (MISSING - causing the bug)

After Fix

Silver layer transformation has:

- SignupDate handling ✅
- Age handling ✅
- CreditScore handling ✅ (ADDED - resolves the bug)