Data engineering Azure Microsoft Fabric documentation

Why did I choose to do this project?

I wanted to use Terraform to create a lift-and-shift foundation of services and infrastructure that I could easily redeploy in a few simple steps whenever I return to the project in the coming months. Additionally, I wanted to explore FinOps—a relatively new cloud operating model—and dive deeper into Microsoft Fabric capabilities.

Table of contents

Section	Topic
1	Introduction to what is FinOps
2	In-depth discussion on the project
3	Technical implementation data engineering implementation
4	Step by step implementation of the FinOps project

Section 1

1)Introduction to what is FinOps

1.1 What is FinOps in general

FinOps ("Finance" and "DevOps") is an operational framework and cultural practice that brings financial accountability to the variable spend model of cloud computing. Rather than serving as a traditional cost-cutting mechanism, FinOps is designed to maximize business value and drive revenue growth by helping engineering, finance, and business teams collaborate on cloud spend decisions. It relies on a continuous lifecycle consisting of three core phases:

- **Inform:** Giving organizations visibility, telemetry, cost allocation, and benchmarking insights.
- **Optimize:** Reducing waste, right-sizing resources, and taking advantage of cloud pricing models.
- **Operate:** Establishing continuous governance, tracking metrics, and aligning business and technical objectives.

1.2 What is FinOps in Microsoft

(https://learn.microsoft.com/en-us/cloud-computing/finops/overview)

Within the Microsoft ecosystem, FinOps is deeply integrated into Azure services, enterprise tools, and governance frameworks. Microsoft is an active premier member of the FinOps Foundation, aligning its cloud services with open standards like the FinOps Open Cost and Usage Specification (FOCUS). Microsoft operationalizes

FinOps through native capabilities such as:

- **Microsoft Cost Management:** A suite of native tools for tracking, monitoring, and allocating multi-cloud and Azure spend.
- **Microsoft Fabric:** Serving as an advanced analytics platform to process scale-out billing data, data lakehouses, and custom cost reporting.
- **Azure Advisor and Azure Policy:** Providing automated recommendations and policy enforcement to maintain continuous financial and operational governance.

1.3 The importance of FinOps

As organizations scale operations in the cloud, unmanaged infrastructure leads to massive financial waste and operational silos between tech teams and finance departments. FinOps is critical because it:

- **Eliminates Silos:** Fosters a collaborative culture where engineers understand financial impacts and finance teams understand technical drivers.
- **Drives Accountability:** Pushes cost ownership out to the engineering and product teams building the workloads, making cost a core metric of operational efficiency.
- **Maximizes ROI:** Ensures that every dollar spent in the cloud directly supports business growth, innovation, and performance optimization rather than vanishing into idle or over-provisioned resources.

1.4)Introduction as to why I choose Microsoft fabric as an analytics platform and Azure as the cloud provider.

Azure was chosen as the cloud provider because its native Cost Management service makes it simple to automate daily billing data exports.

An **Azure Storage Account (ADLS Gen2)** securely stores these raw cost files and our Terraform infrastructure state.

Microsoft Fabric was selected as the analytics platform because it combines data engineering, lakehouses, and warehousing into a single, unified workspace.

Fabric's **Medallion architecture** (Bronze, Silver, and Gold layers) allows us to easily ingest raw billing files, clean them up, and model them into business-ready data.

Built-in features like **OneLake shortcuts** and **native Git integration** make it seamless to manage our multi-environment CI/CD workflows across dev, test, and prod.

Together, this stack provides a practical, low-overhead way to turn raw Azure cloud bills into clear, queryable financial insights.

Section 2

2 in-depth discussion on The finOps project

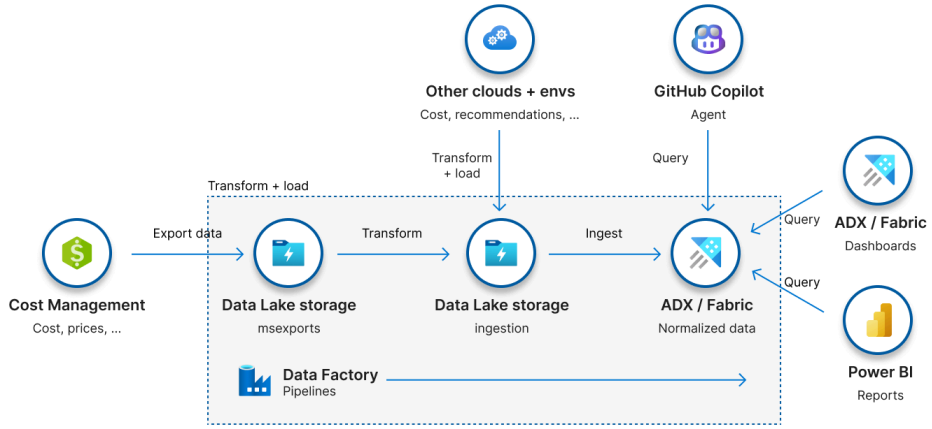
In this project, I built an FinOps platform on Azure microsoft fabric designed to automate how cloud costs are tracked, processed, and analyzed.

The primary aim was to bridge the gap between raw cloud billing data and actionable business insights by creating a seamless, automated pipeline. The workflow takes daily cost exports directly from Azure Cost Management, moves them through a structured data lakehouse, and transforms them into business-ready models for end users and reporting dashboards.

To manage the underlying cloud and analytics resources, I used **Terraform** as the Infrastructure as Code tool. Its declarative nature, combined with a highly modular and reusable design, was a major purpose of this project—allowing me to reliably

provision and manage reproducible environments across development, testing, and production.

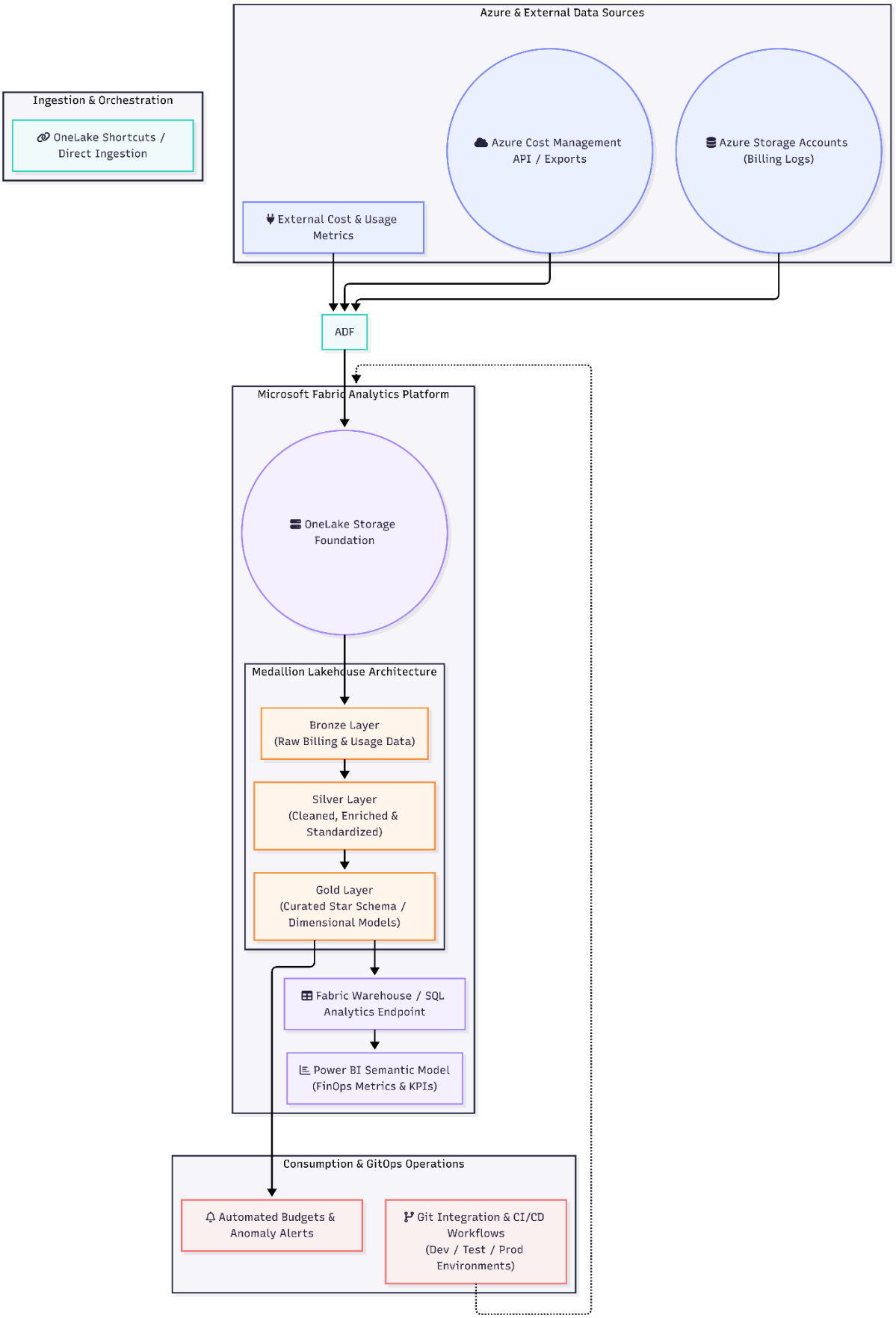
I was inspired by this FinOPs diagram (https://learn.microsoft.com/en-us/cloud-computing/finops/toolkit/hubs/finops-hubs-overview)

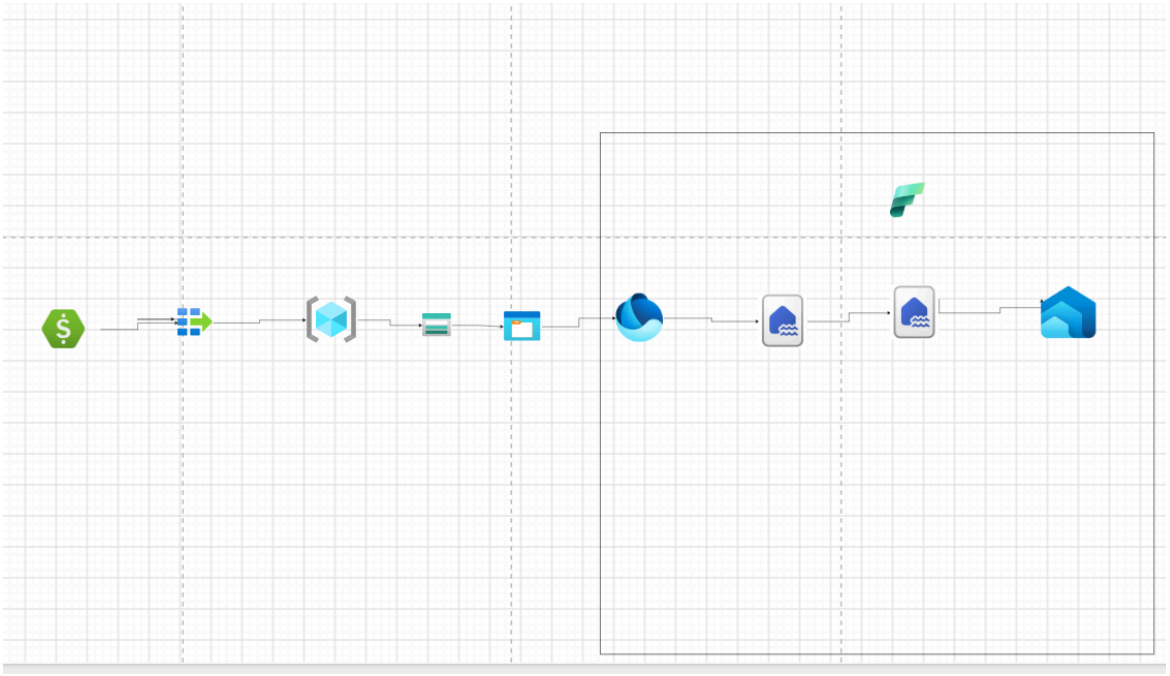


Section 3

Technical Implementation & Data Engineering Overview

Diagram on the finops ETL flow


Azure & External Data Sources
Azure Cost Management API / Exports
Azure Storage Accounts (Billing Logs)
External Cost & Usage Metrics
Ingestion & Orchestration
OneLake Shortcuts / Direct Ingestion
ADF
Microsoft Fabric Analytics Platform
OneLake Storage Foundation
Medallion Lakehouse Architecture
Bronze Layer (Raw Billing & Usage Data)
Silver Layer (Cleaned, Enriched & Standardized)
Gold Layer (Curated Star Schema / Dimensional Models)
Fabric Warehouse / SQL Analytics Endpoint
Power BI Semantic Model (FinOps Metrics & KPIs)
Consumption & GitOps Operations
Automated Budgets & Anomaly Alerts
Git Integration & CI/CD Workflows (Dev / Test / Prod Environments)



This project uses modern data engineering practices, Infrastructure as Code, and GitOps workflows to build a reliable, multi-environment FinOps platform.

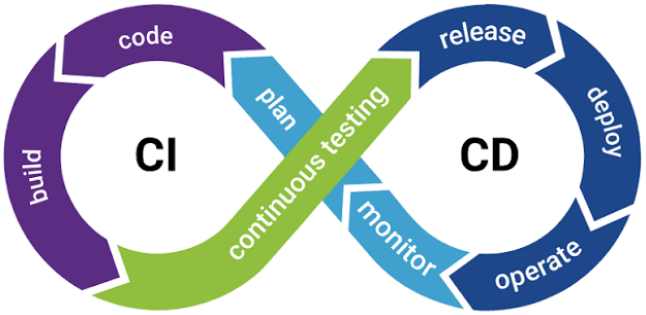
The technical stack is built on three core pillars:

- **Modular Infrastructure with Terraform:** Everything from Azure storage to Microsoft Fabric workspaces is provisioned using Terraform. By writing modular and reusable code combined with Terraform Workspaces (test,dev,prod) I ensured that deployments remain consistent and easy to replicate across environments and powershell to automate a few tasks.

- **Medallion Data workflow:** To process raw cloud bills, the data pipeline follows a structured three-tier lakehouse pattern:

    - **Bronze Layer:** Ingests raw CSV cost exports straight from Azure Cost Management into high-performance Delta tables.
    - **Silver Layer:** Cleanses the data, fixes data types, filters out bad rows, and standardizes the schema.

    - **Gold Layer:** Models the clean data into a Star Schema inside a Fabric Data Warehouse, making it fast and ready for reporting dashboards.



CI/CD

- **GitOps and CI/CD Automation:** Environment configurations and code are managed through GitHub, with specific branches mapped directly to each Fabric workspace (test, dev and prod). Moving code between environments is handled via standard pull requests, keeping deployments automated and organized.

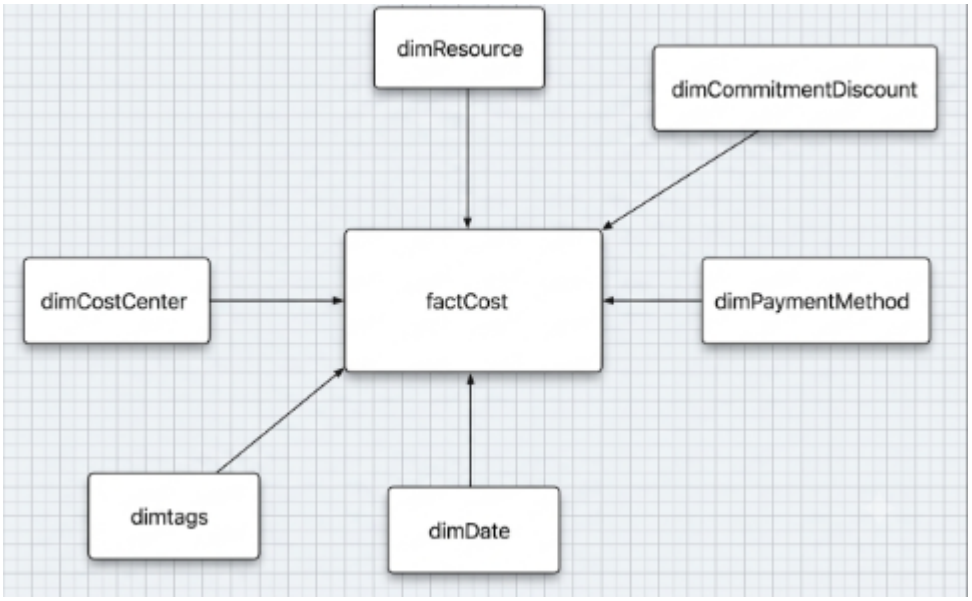
Data Modeling for Cost Management Data

Here is the google sheet with model (diagrams(https://docs.google.com/spreadsheets/d/1bJgjWkfDE1InX8WBE7q4YPKeAn5v7VFl0dl3qWFb58o/edit?gid=0#gid=0)

Conceptual & Physical Data Modeling Strategy

Transforming raw, denormalized Azure Cost Management files into high-performance analytical structures requires a rigorous transition from conceptual relationships to physical database schemas.

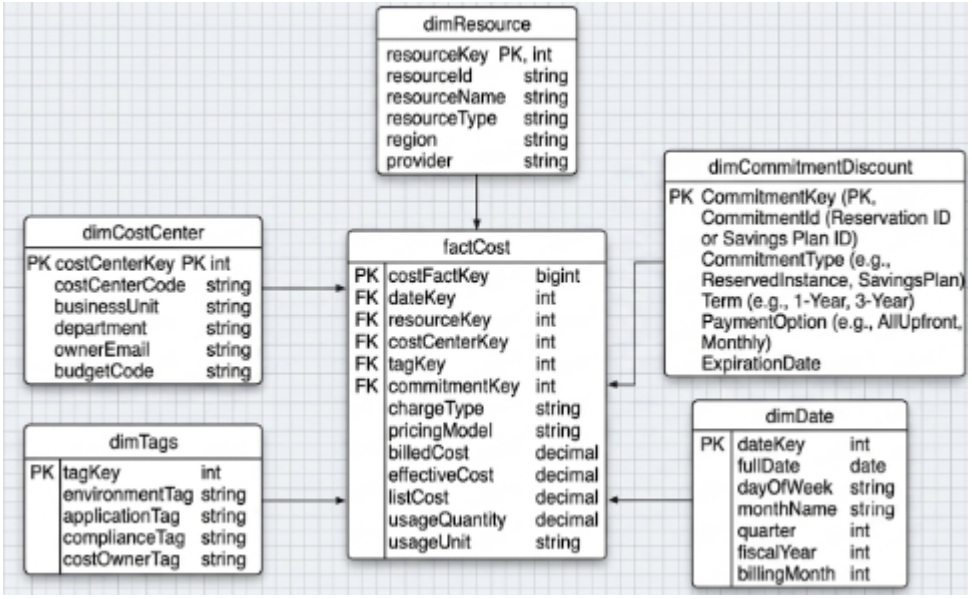
1. Conceptual Modeling



At the conceptual level, the objective is to decouple technical cloud telemetry from organizational reporting hierarchies. We map disparate cloud resource entities—such as subscription IDs, resource group names, meters, and resource types—to real-world business dimensions:

- **Cost Centers & Business Units:** Mapping usage to the respective departments or projects incurring the costs.
- **Environments:** Categorizing spend across development, staging, and production tiers.
- **Service Categories:** Grouping granular meters into logical product buckets (e.g., Compute, Storage, Networking, AI).

2. Physical Modeling (Dimensional Star Schema)



In the Gold layer, the physical schema is implemented as a **dimensional star schema** using optimized storage formats (Delta Lake / Parquet):

- **Fact Tables (`FactCloudCost`):** Designed at a granular transactional grain (e.g., line-item usage record). It contains additive or semi-additive numerical metrics such as `costInBillingCurrency`, `consumedQuantity`, `effectivePrice`, and `amortizedCost`. Foreign keys link this table to the associated dimensions.

- **Dimension Tables (`DimDate`, `DimResource`, `DimService`, `DimAccount`):** Store descriptive, slow-changing attributes to provide context to the facts. Strict naming conventions (such as `camelCase`) and enforced data types are applied across all attributes to ensure consistency for downstream consumers.

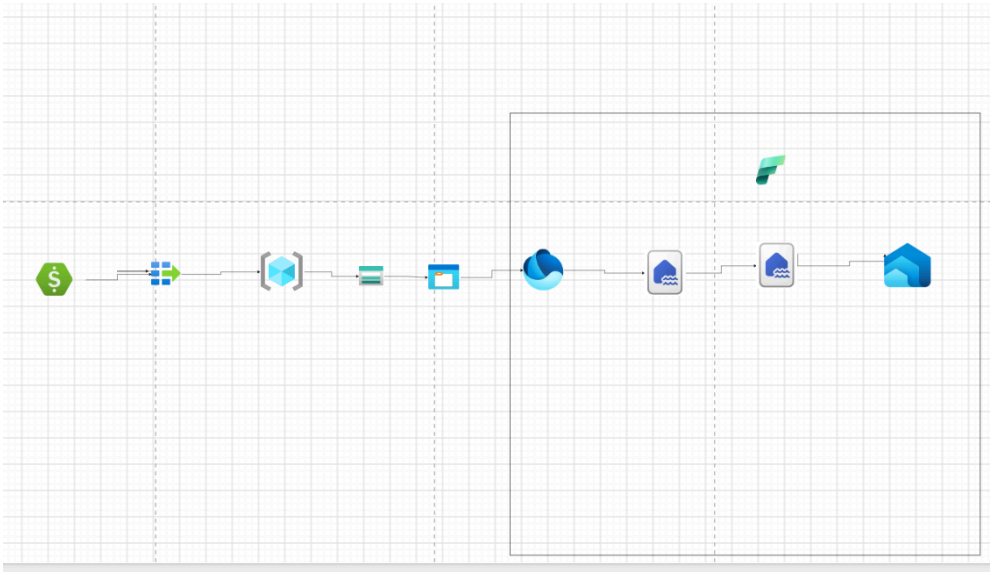
Core Business Questions Answered by the Model

The structured data model is explicitly tuned to answer critical financial and operational questions:

- **Cost Allocation & Attribution:** *Where and by whom is cloud expenditure being driven across various subscriptions and resource groups?* (Answered via joins between `FactCloudCost` and organizational dimension tables).
- **Budget Tracking & Variance Analysis:** *Which services or business units are deviating from historical cost baselines?* (Answered via time-series aggregations across `DimDate` and service dimensions).
- **Forecasting & Predictive Modeling:** *What will next quarter's multi-cloud expenditure look like based on historical consumption trends?* (Answered by exposing deeply versioned historical fact datasets to machine learning compute engines).

3. Technical Data Engineering Discussion & Principles

Diagram



Directory Structure Layout

```
FINOPS PROJECT/
├── costData/
├── finops-fabric-platform/
│   ├── Config/
│   │   └── fabric.json
│   ├── fabricContent/
│   │   ├── Bronze.Lakehouse
│   │   ├── bronzeFilesStorage.DataPipeline
│   │   ├── bronzeFilesStorage.Notebook
│   │   ├── bronzeTOSilver.DataPipeline
│   │   ├── finOpsBronzeTOSilver.Notebook
│   │   ├── finOpsSilverTOGOLDDIMFacts.Notebook
│   │   ├── Gold.Warehouse
│   │   ├── Silver.Lakehouse
│   │   └── silverTOGOLDDIMFacts.DataPipeline
│   ├── scripts/
│   │   ├── assignFabricCapacity.ps1
│   │   ├── authenticateFabric.ps1
│   │   ├── bootstrap.ps1
│   │   ├── createOneLakeShortcut.ps1
│   │   ├── createWorkspace.ps1
│   │   └── FabricAPI.ps1
│   ├── sql/
│   │   ├── procedures/
│   │   ├── views/
│   │   └── test.sql
```

```
│   ├── terraform/
│   │   ├── environments/
│   │   └── modules/
│   │       ├── costManagementExport/
│   │       ├── fabric/
│   │       ├── resourceGroup/
│   │       ├── storageAccount/
│   │       └── storageContainer/
│   └── .gitignore
```

Core Data Engineering Principles Applied

- **ETL (Extract, Transform, Load):** Systematic extraction of cloud billing files, transformation across distributed compute nodes, and loading into structured lakehouse layers.
- **Data Modeling:** Strict separation between raw ingestion, technical cleansing, and dimensional star schema design for business intelligence performance.
- **Medallion Architecture:** Multi-tier processing pattern (**Bronze** for raw ingestion, **Silver** for cleansed/standardized data, and **Gold** for curated business-ready semantic models).

- **Batch Processes:** Scheduled daily pipeline executions optimized for high-volume historical ingestion without exhausting cluster resources.

Infrastructure, Automation, & Tooling Standards

- **Terraform (Idempotency & Modularity):** Used to provision infrastructure reproducibly. By utilizing modular blocks, variables are cleanly abstracted to prevent hardcoding, enabling seamless management of cloud resources, storage accounts, and Fabric workspaces.
- **PowerShell Scripting:** Utilized for operational automation, handling secure Azure authentication contexts, managing capacity scaling, and executing deployment wrappers.
- **Git & CI/CD Workflows:** Ensures that code and workspace artifacts are persistently tracked, allowing seamless recovery, review, and migration across environments.

- **PySpark:** Chosen over standard Python to leverage distributed cluster computing capabilities across massive multi-gigabyte billing files, ensuring horizontal scalability.

Environment Segregation (Dev, Test, Prod)

Isolating workspaces into distinct development, testing, and production tiers ensures that code changes, schema alterations, and experimental notebooks do not disrupt operational reporting environments. This lifecycle mimics enterprise software engineering practices.

OneLake Configuration & Provisioning

- **Creation & Configuration:** OneLake acts as the unified logical storage foundation for Microsoft Fabric. It was provisioned automatically via Terraform modules that set up the underlying Fabric tenant workspaces and storage links.
- **Data Population:** Data Factory pipelines and native shortcuts populate OneLake by ingesting billing files and logs directly from Azure storage containers, making data instantly accessible to compute engines without duplication.

PySpark Notebook Implementation & Design Decisions

- **Why PySpark over Python?** Standard single-node Python struggles to process millions of cloud billing logs efficiently. PySpark distributes data processing across a cluster, allowing fast manipulation of large-scale Parquet datasets.
- **Design Practices:** Avoided hardcoded values entirely by passing configuration parameters and dynamic paths using notebook variables. Transformations are written declaratively to ensure clean execution lineage.

Medallion Tier Walkthrough

1. Bronze Layer (Raw Ingestion & Historical Storage)

- **Function:** Serves as the immutable landing zone for raw billing and log files.
- **Storage Format:** Stored as **Parquet** files to provide columnar compression and optimal query performance.
- **Metadata & History:** Adheres to reference Microsoft documentation for lakehouse design, maintaining an append-only historical record. This ensures that past billing states remain untouched and fully auditable over time.

2. Silver Layer (Cleansing & Standardization)

- **Notebook (`bronze_to_silver`):** Reads the raw Parquet files from the Bronze layer using PySpark.
- **Logic & Steps:** Executes structural data cleaning, strips out corrupted schema fields, performs deduplication checks, and handles missing or null values.
- **Focus:** This layer is strictly for data engineering hygiene; no semantic business logic or reporting dimensions are built here yet, as the data is still undergoing structural standardization.

3. Gold Layer (Dimensional Modeling & Serving)

- **Warehouse & Meaning:** The Gold layer hosts fully modeled, business-ready star schemas within Fabric warehouses or lakehouse SQL endpoints.
- **Notebook (`silver_to_gold`):** Scheduled daily batch PySpark jobs read from the Silver layer, transforming clean data into structured dimension (`dim`) and fact (`fact`) tables.
- **Logic:** Applies strict data type enforcement and naming conventions (such as `camelCase`). The resulting fact tables reach the exact granularity and business logic required for end users and analysts to build reliable reports.

Versioning, Git Integration, & Personal Engineering Standards

How Versioning Works in Microsoft Fabric & Git

Microsoft Fabric natively integrates with Git (such as GitHub), synchronizing workspace items (notebooks, pipelines, semantic models) as structured files. Code changes made in development are tracked, committed, and promoted through CI/CD pipelines to test and production workspaces.

Overcoming Infrastructure Loss & Subscriptions

A primary motivation for implementing this robust Terraform and GitOps pipeline stemmed from frustration with ephemeral student or trial cloud subscriptions—where finished projects vanish once resources are deleted or trials expire, leaving behind only static screenshots as proof of work.

- By committing infrastructure code via **Terraform** and workspace logic via **Git**, the entire architecture becomes portable and reproducible.
- Irrespective of subscription tiers or workspace lifecycles, the complete project state can be pulled down and redeployed anywhere at a moment's notice.

Section 4

Step by step implementation

Once Terraform was installed, we needed a way for it to talk to Azure securely. We used two main methods:

1. Local CLI (`az login`): The quick interactive way for daily development—opens a browser, authenticates, and passes your session token straight to Terraform.\

2. Service Principal / Env Variables: The automated way—using client IDs, secrets, and tenant IDs so Terraform and pipelines can authenticate headlessly without pop-ups.

```
Install-Module -Name Az -Repository PSGallery -Force

Connect-AzAccount
```

Once tools were sorted, we jumped into writing our actual Terraform modules inside the `environments/dev` folder to provision everything modularly.

Here is what the **`main.tf`** orchestration file looks like, tying together our resource group, storage account, containers, and cost management export:

```
data "azurerm_client_config" "current" {}

module "resourceGroup" {
  source = "../../modules/resourceGroup"

  resourceGroupName = local.dynamic_resource_group_name
  locationName      = var.locationName
  environmentName   = var.environmentName

  tags = local.commonTags
}

module "storageAccount" {
  source = "../../modules/storageAccount"

  storageAccountName = local.storage_account_name
  resourceGroupName  = module.resourceGroup.resourceGroupName
  locationName       = var.locationName

  tags = local.commonTags
}

module "storageContainer" {
  source = "../../modules/storageContainer"

  for_each = toset(var.containerNames)

  containerName    = each.value
  storageAccountId = module.storageAccount.storageAccountId
}

module "costManagementExport" {
  source = "../../modules/costManagementExport"

  subscriptionId     =
data.azurerm_client_config.current.subscription_id
  storageAccountId   = module.storageAccount.storageAccountId
```

```
  storageAccountName = module.storageAccount.storageAccountName
  containerName      = "costexports"
}
```

We then create a output.tf in the same directory.

```
output "resourceGroupName" {

  value = module.resourceGroup.resourceGroupName

}

output "resourceGroupId" {

  value = module.resourceGroup.resourceGroupId

}
```

I then created the variable.tf name and entered the correct variables with the correct types and descriptions.

```
variable "subscriptionId" {
  description = "Azure Subscription ID"
  type        = string
}


variable "baseResourceName" {
  description = "Base name for resources"
  type        = string
}

variable "locationName" {
  description = "Azure region"
  type        = string
}
```

```
variable "environmentName" {
  description = "Environment"
  type        = string
}
```

versions.tf I create this from the documentation and use of ai to help.

```
terraform {
  required_version = ">= 1.8.0, < 2.0.0"

  required_providers {

    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 4.81"
    }

}
```

Terraform.tfvar

```
subscriptionId     = "{secret key}"
baseResourceName   = "finops"
locationName       = "South Africa North"
environmentbase  =  test
```

Using official documentation and LLM assistance, we provisioned our infrastructure modules, including resource groups, storage accounts, and automated cost management exports,modularly via Terraform.

```
```

```
resource "azurerm_subscription_cost_management_export" "this" {
  name             = "finOpsDailyCostExport"
  subscription_id  = "/subscriptions/${var.subscriptionId}"

  recurrence_type  = "Daily"
  recurrence_period_start_date = formatdate("YYYY-MM-DD'T'00:00:00Z",
timestamp())
  recurrence_period_end_date   = "2030-12-31T23:59:59Z"

  export_data_options {
    type       = "ActualCost"
    time_frame = "MonthToDate"
  }

  export_data_storage_location {
    container_id     =
"${var.storageAccountId}/blobServices/default/containers/${var.container
Name}"
    root_folder_path = "azure-cost-data"
  }
}
```

I created Output.tf

```
output "costExportId" {
  value = azurerm_subscription_cost_management_export.this.id
}


output "costExportName" {
  value = azurerm_subscription_cost_management_export.this.name
}
```

I the created variable.tf

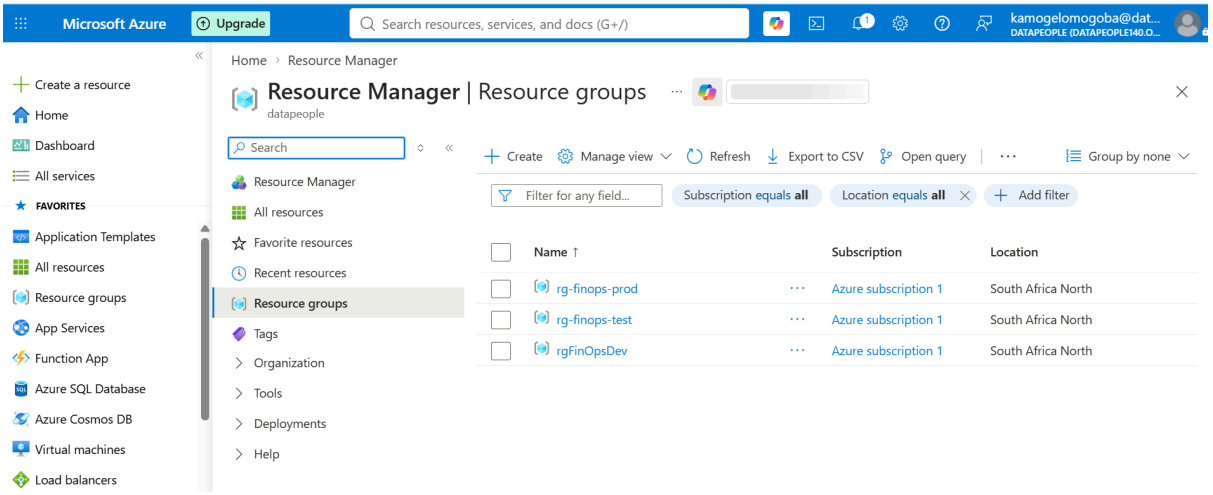
```
variable "storageAccountName" {
  description = "Storage account name"
  type        = string
}


variable "containerName" {
  description = "ADLS Gen2 filesystem name"
  type        = string
}
```

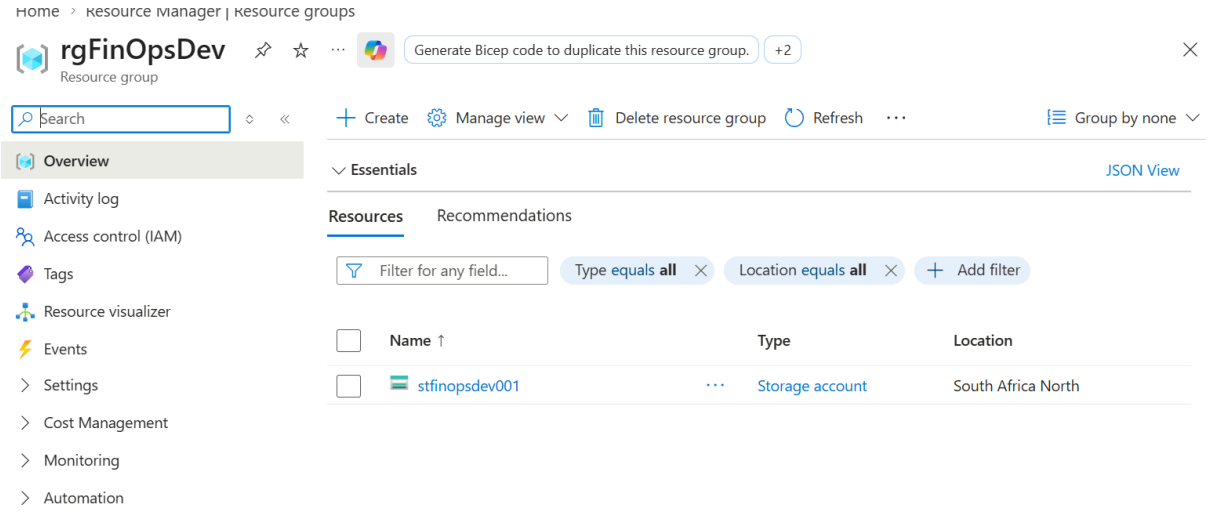
```
terraform workspace select -or-create prod
 terraform innit
terraform plan
terraform validate
```

The terraform workspace command is important when you have all three workspace up and running because it does not kill you current state to create the new state it checks if the state already exists if not it then creates a new workspace preserves current states(NB)

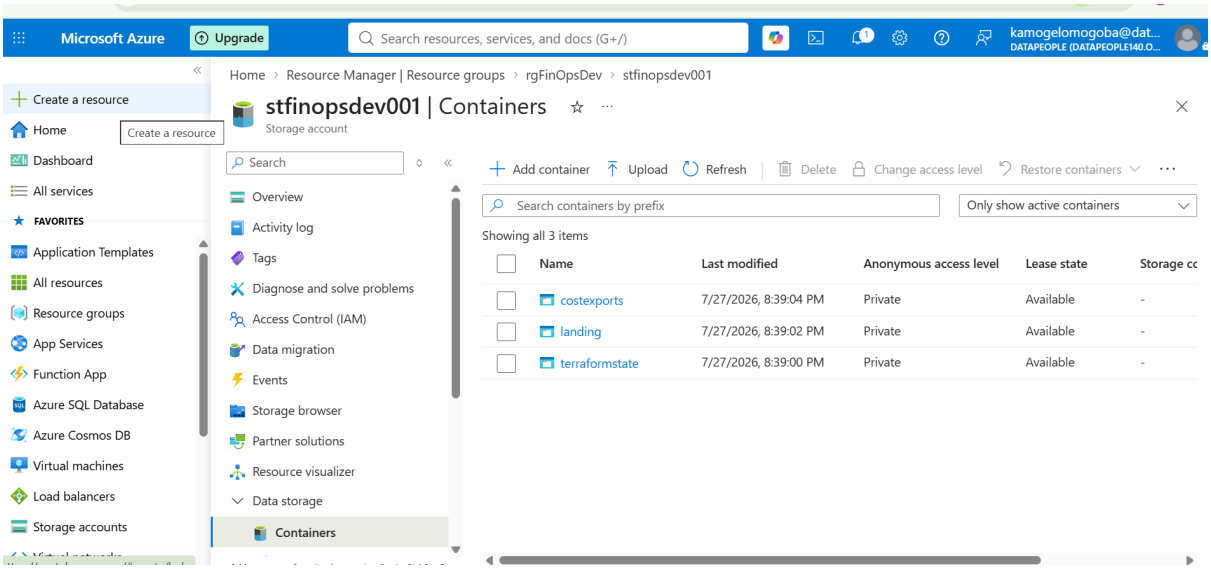
Using this modular Terraform template (splitting `main.tf`, `variables.tf`, and `outputs.tf` across dedicated folders), we successfully provisioned our resource groups, storage accounts, and containers, bringing us to the point where the resource group is live and visible in Azure.



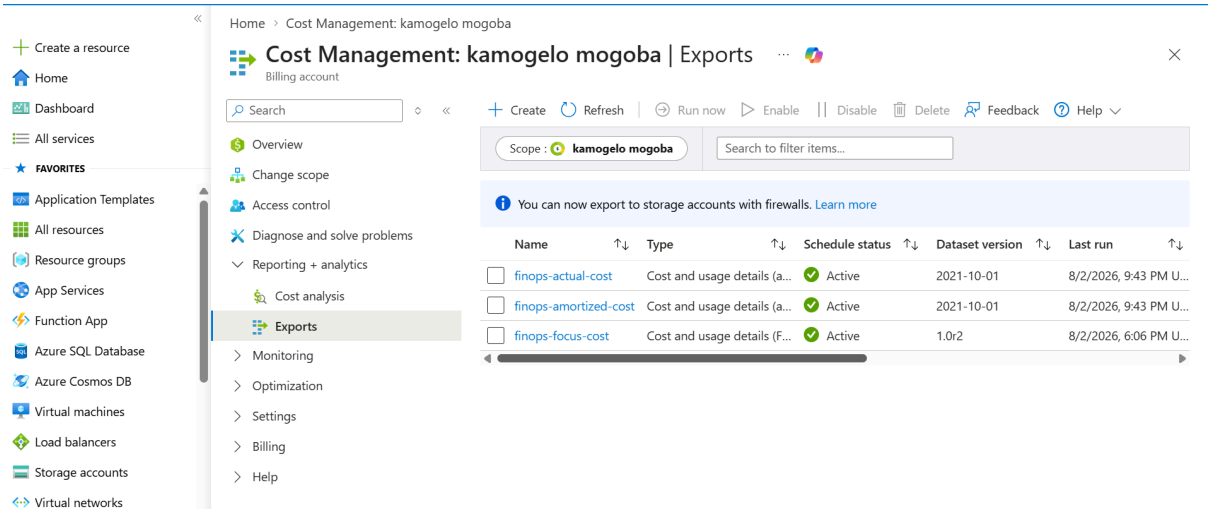
In your first resource group you will see the rg-finops-prod. Storage account will be visible.



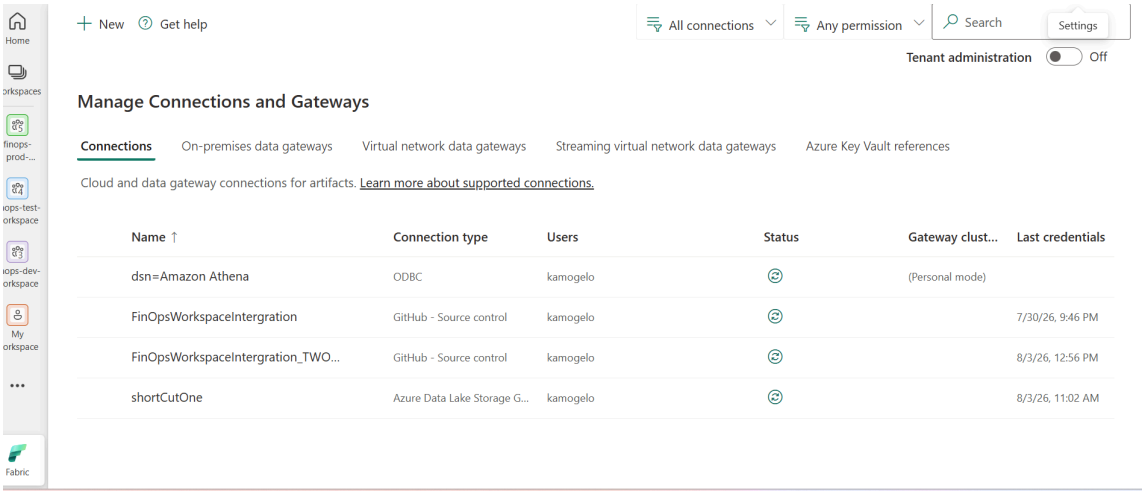
The container with the folders you provisioned.



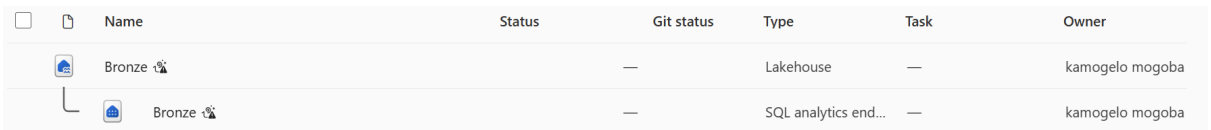
Once you reach this stage you will have to manually go into the costexports and schedule your files to storage account under costexport folder and daily schedule for 20:00.

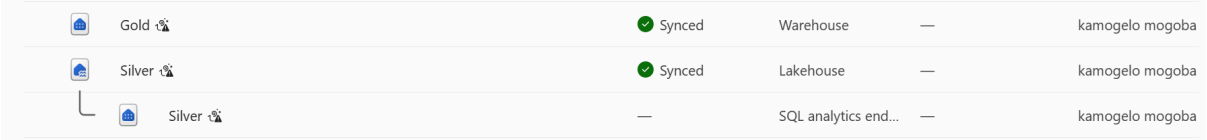


To connect everything, we used a OneLake shortcut provisioned via Terraform, followed by a quick manual step in Fabric to activate the connection and set up the required gateway under "Manage Connections."

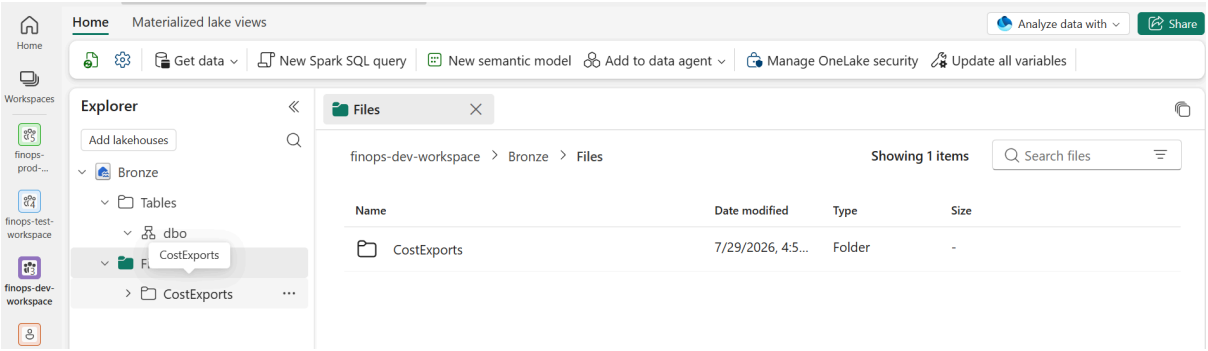


Using those same modular Terraform steps, we provisioned the core Microsoft Fabric data layers: the **Bronze Lakehouse**, **Silver Lakehouse**, and **Gold Warehouse**.



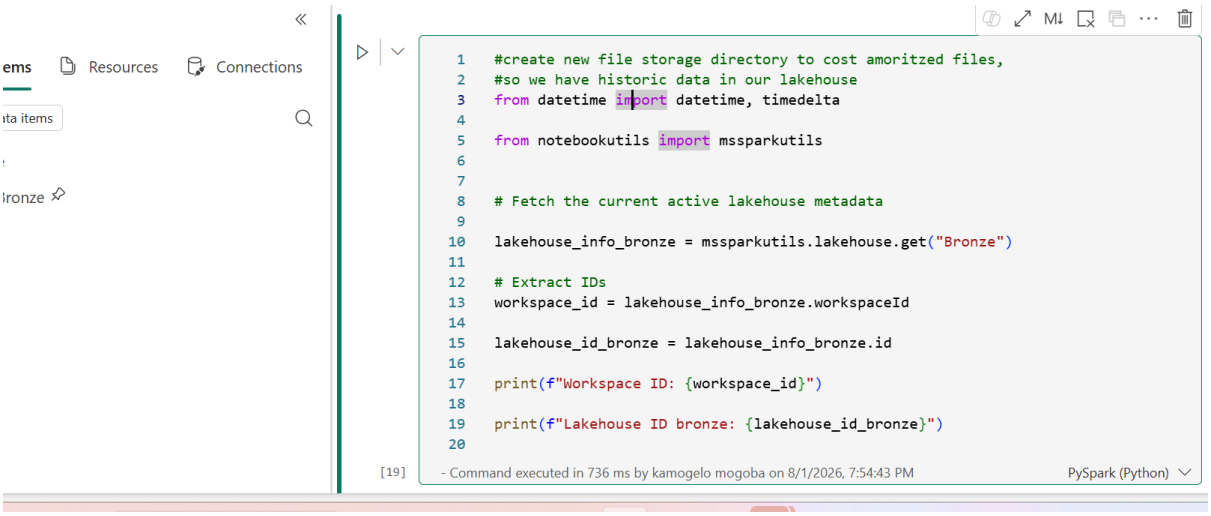


Once the Bronze layer is created and the OneLake shortcut is successfully connected, the raw cost export data immediately flows through and populates the Bronze folder.

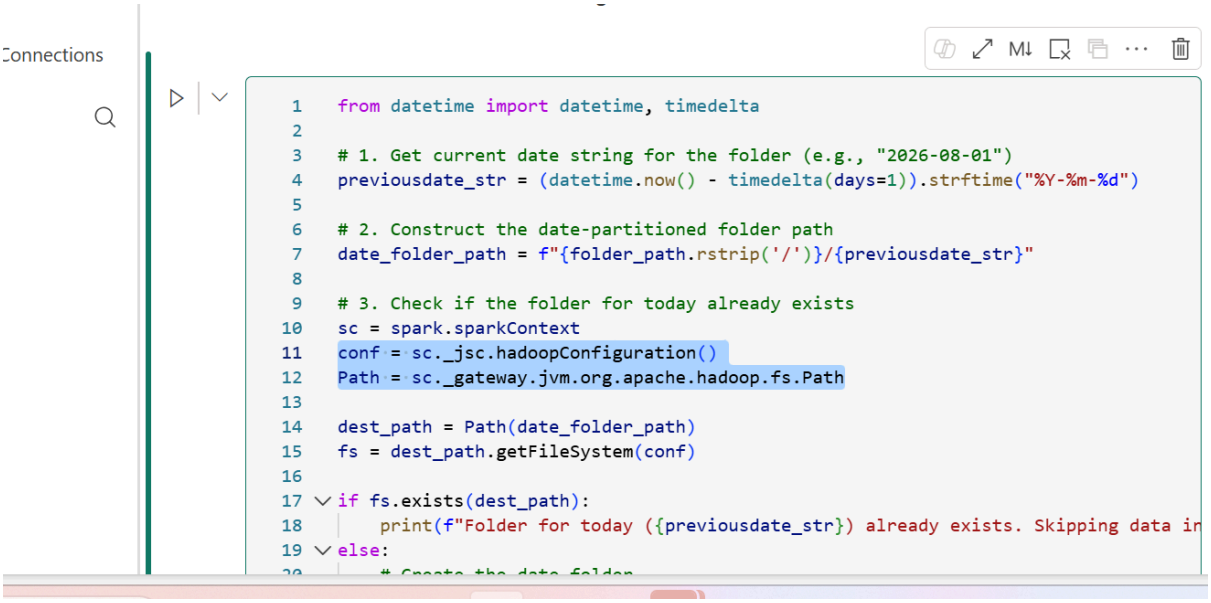


In the amortized cost folder, the data is stored in Parquet format, and to preserve historical records, we run a daily PySpark notebook at 09:00 using `mssparkutils` that fetches the data and copies it straight into the data storage folder within the Bronze Lakehouse.

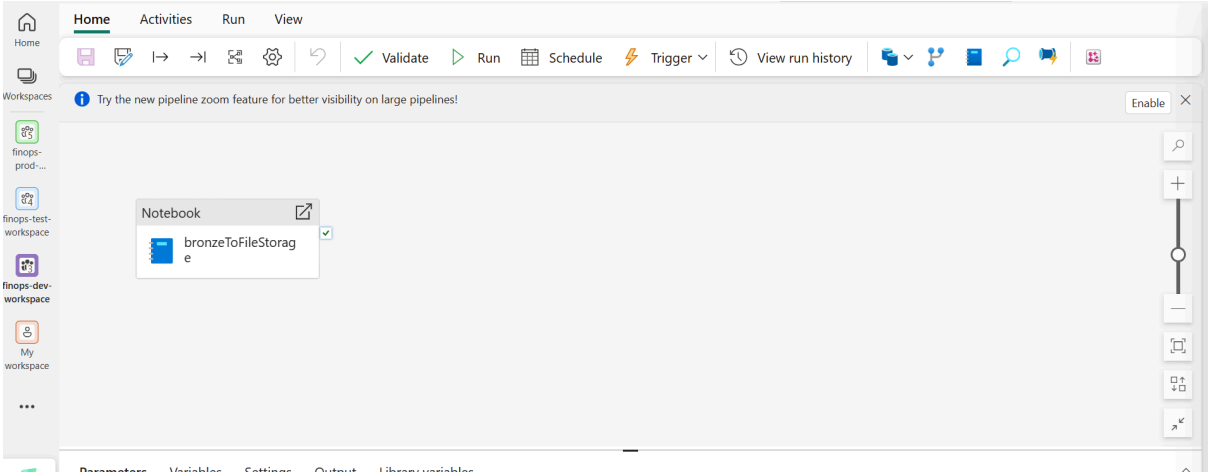
To avoid hardcoding absolute paths, we pulled workspace and lakehouse IDs dynamically straight from Azure and the Spark context, keeping everything flexible and clean.



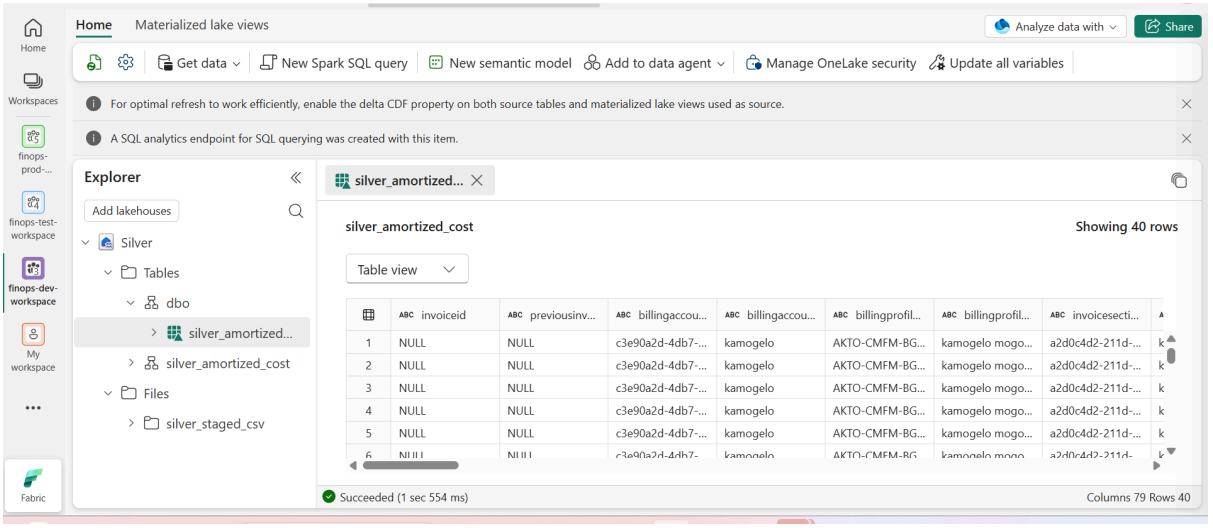
We also used the Spark context and Hadoop configurations to programmatically create file folders right inside the Spark notebook.



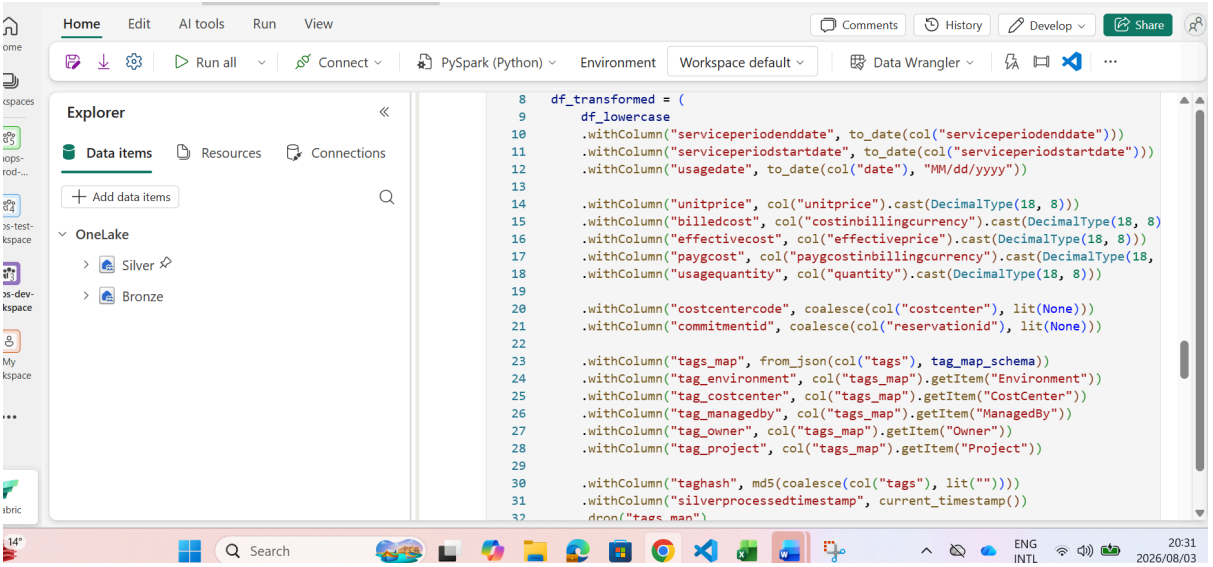
This notebook was the scheduled within a pipeline.



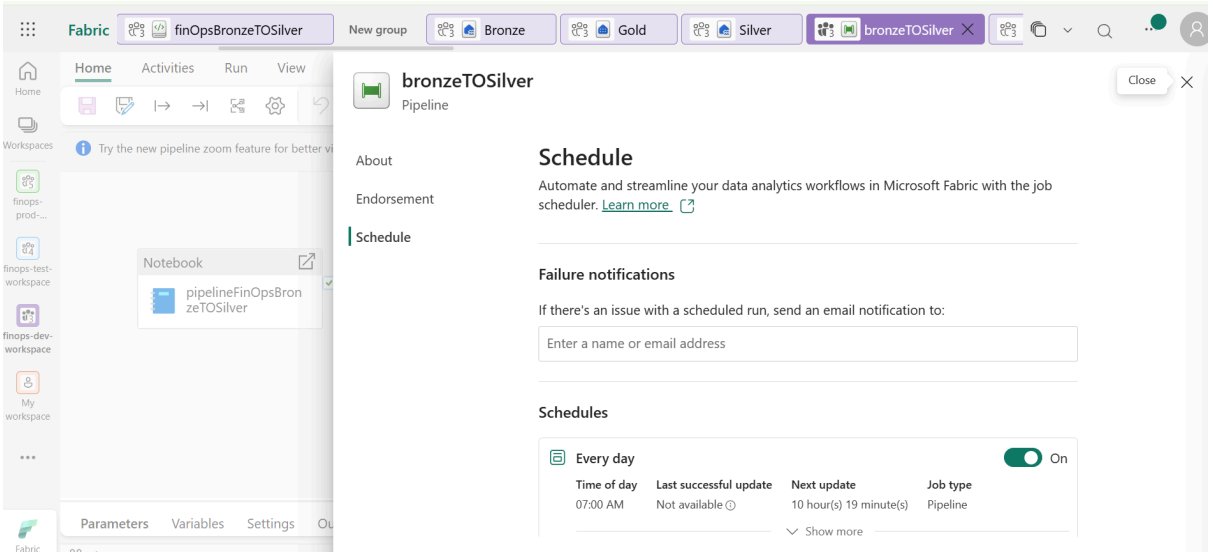
In the silver lakehouse should find a table from the bronze run.



Next, the `bronzetosilver` notebook reads the data from the mounted Bronze lakehouse source and applies PySpark transformations, including assigning data types and using the `coalesce` function to safely handle missing records with nulls.



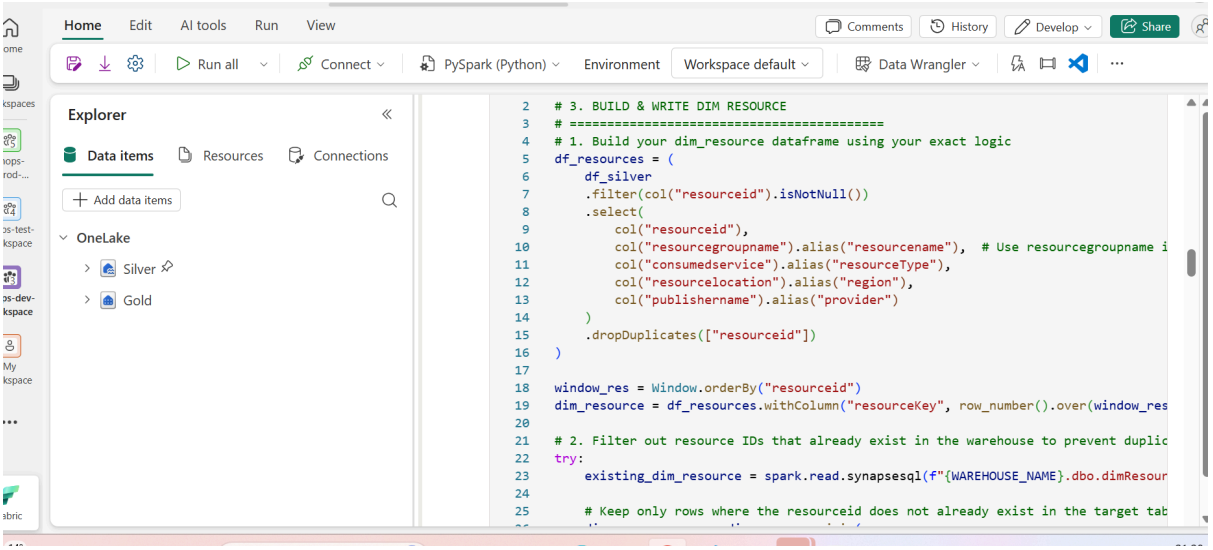
We then schedule this notebook in pipeline.



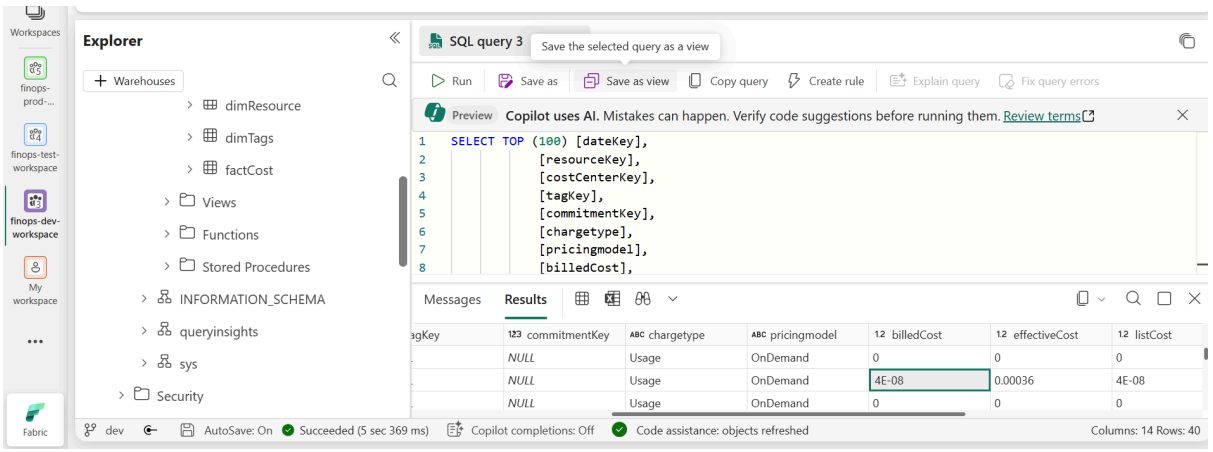
Once the silver table is up.

We then go to the gold warehouse were we create dim and fact tables.

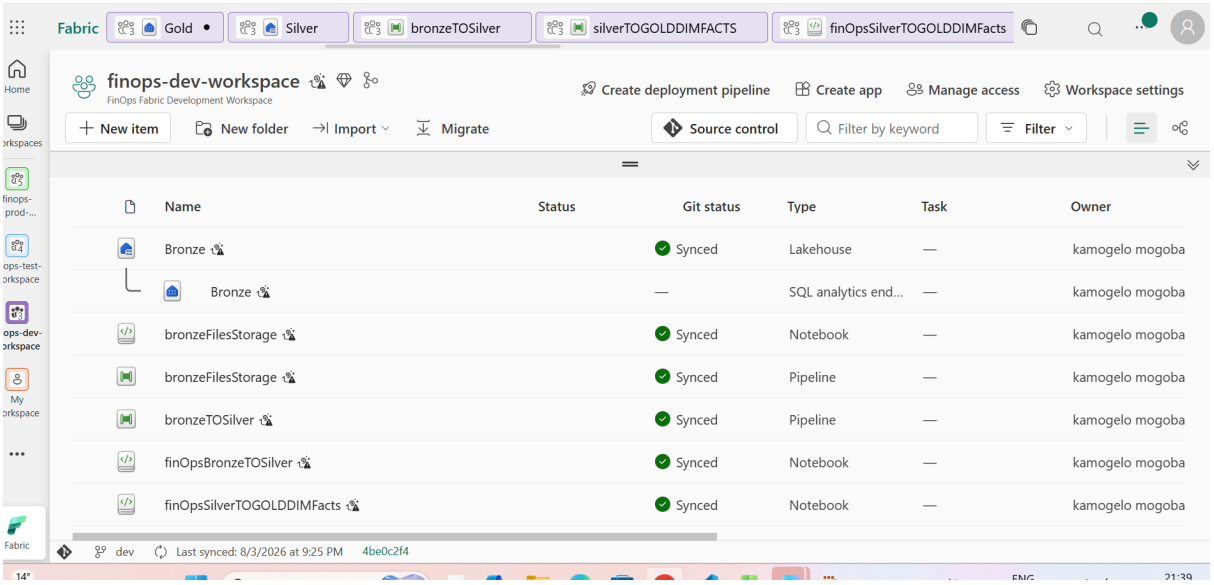
We used the finopssilvertogolddimfact notebook to create this dim resource table



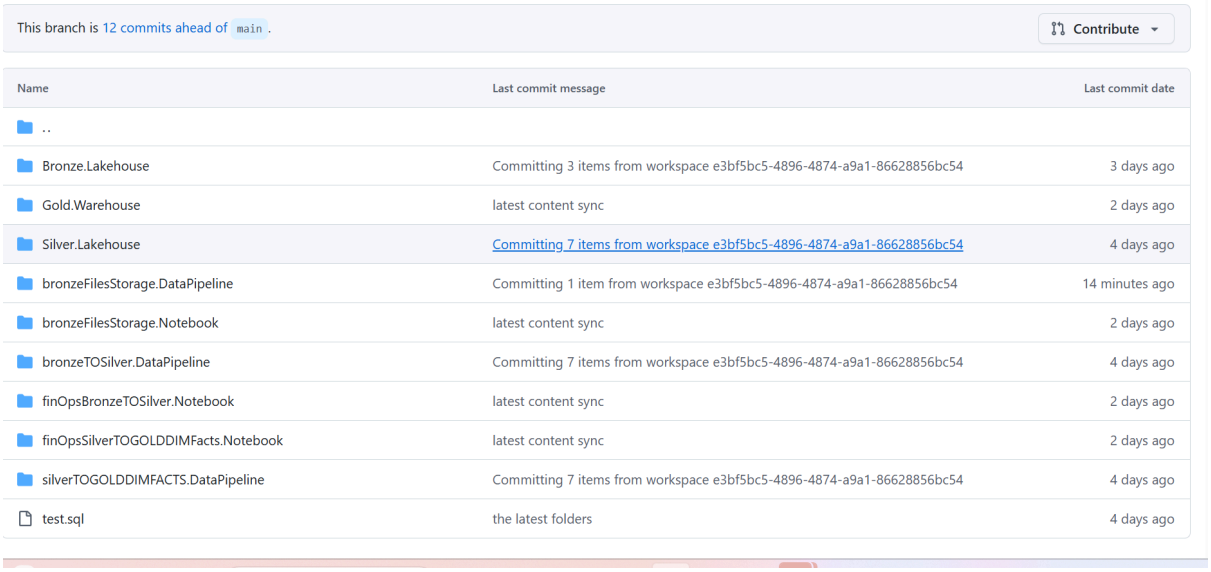
We then renamed silver table columns for our dimension tables, added `try-except` blocks to prevent duplicates, populated all five dimension tables and the fact table, and finally automated the whole workflow by creating and scheduling a pipeline.



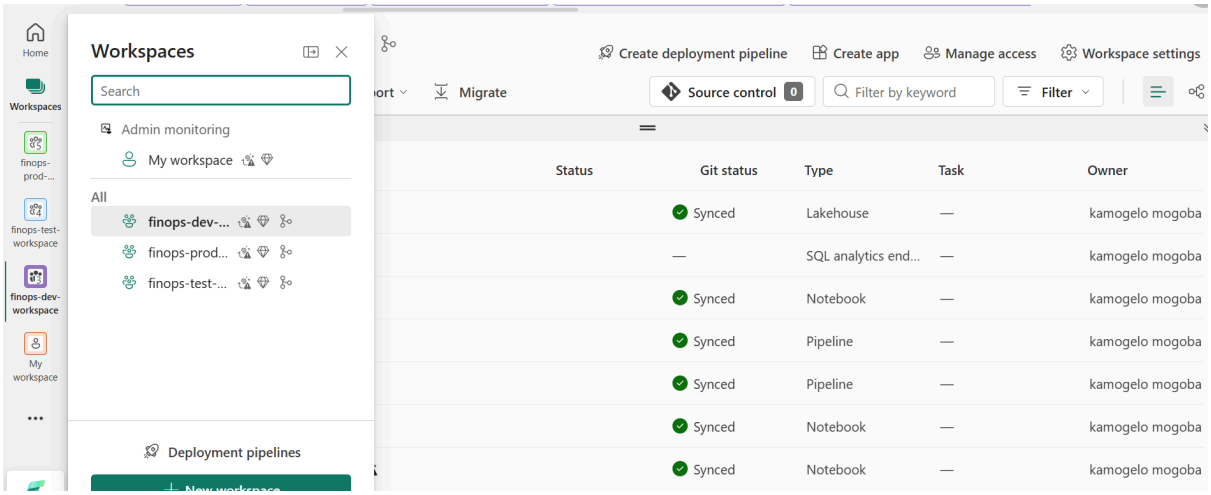
After building the `factCost` table and ensuring all code changes were versioned, we integrated the workspace with GitHub by pushing our work to the `dev` branch and syncing the latest updates.



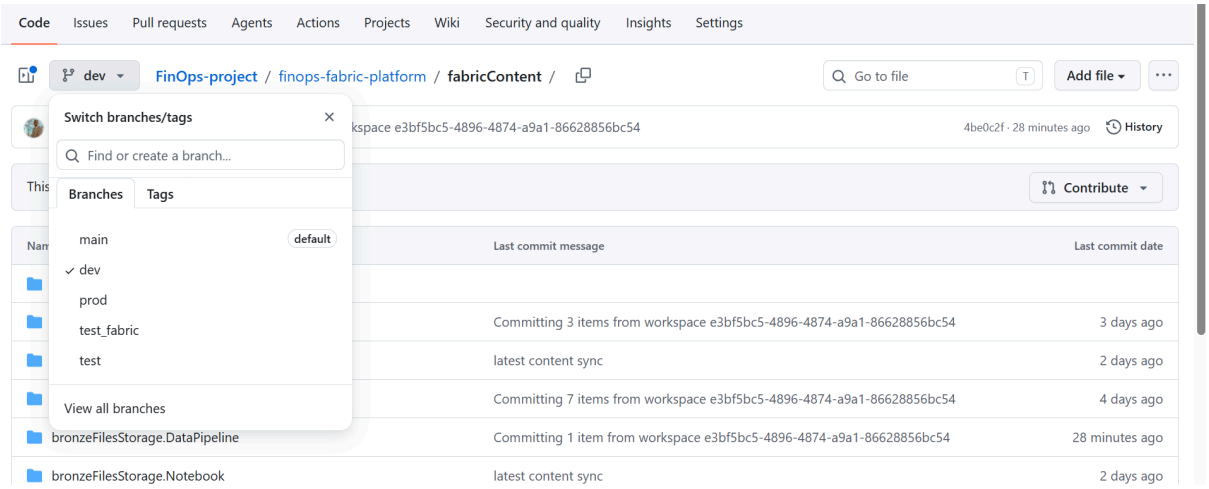
The git repo should have these content in the specified folder.



From there, we created `test` and `prod` branches in the repository, applied the same Git integration to their respective Microsoft Fabric workspaces, and successfully established a fully aligned, 3-tier Dev, Test, and Prod environment pipeline.



Git hub repo with the test, dev and prod branches



The Key takeaways

The Multi-Environment work flow

To keep code stable and prevent raw changes from breaking production, map your infrastructure and Git branches like this:

Environment	Fabric Workspace Name	GitHub Branch	Purpose
Development	finops-dev-workspace	dev	Where developers sync changes, test new notebooks/pipelines, and commit code.
Testing	finops-test-workspace	test	Automated integration testing. Code lands here via Pull Request from dev — test.

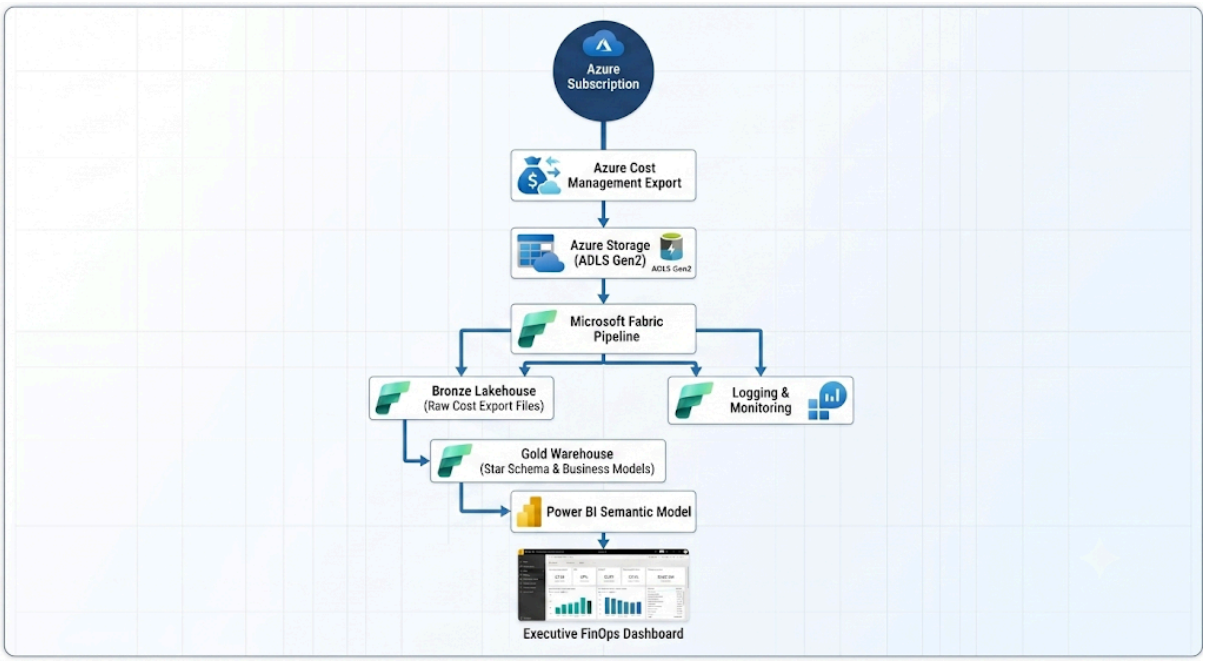
Production	finops-prod-workspace	main (or prod)	Stable user-facing environment. Code arrives here via Pull Request from test — main.

How the CI/CD Pipeline Flow Works

When using Fabric's native Git integration, the workflow changes from manual clicking in the UI to a structured code promotion process:

1. **Develop:** A developer makes changes in the dev workspace (or locally via VS Code) and commits/pushes updates to the GitHub **dev** branch.
2. **Sync to Workspace:** Fabric's workspace finops-dev-workspace automatically detects or updates changes from the dev branch.
3. **Promote via Pull Request:**
    - To move to Test, open a Pull Request in GitHub from dev —test.
    - Once merged, GitHub Actions or Fabric updates the **test** workspace.
4. **Production Release:** Open a Pull Request from test —- main. Once merged, the production pipeline deploys changes to **finops-prod-workspace**.

Lastly, the importance of terraform the reusability !!!


Azure Subscription
Azure Cost Management Export
Azure Storage (ADLS Gen2)
ADLS Gen2
Microsoft Fabric Pipeline
Bronze Lakehouse (Raw Cost Export Files)
Logging & Monitoring
Gold Warehouse (Star Schema & Business Models)
Power BI Semantic Model
Executive FinOps Dashboard