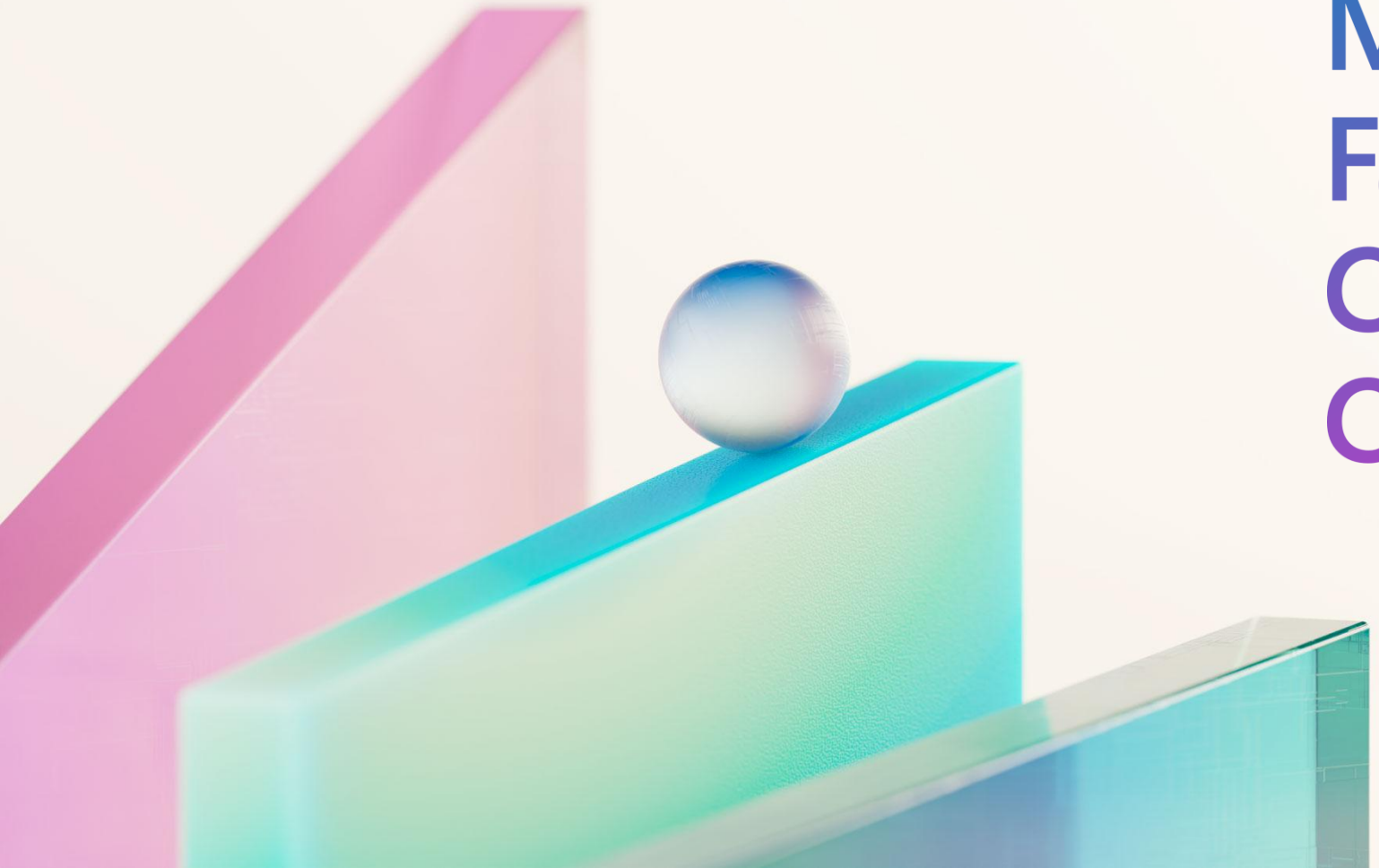
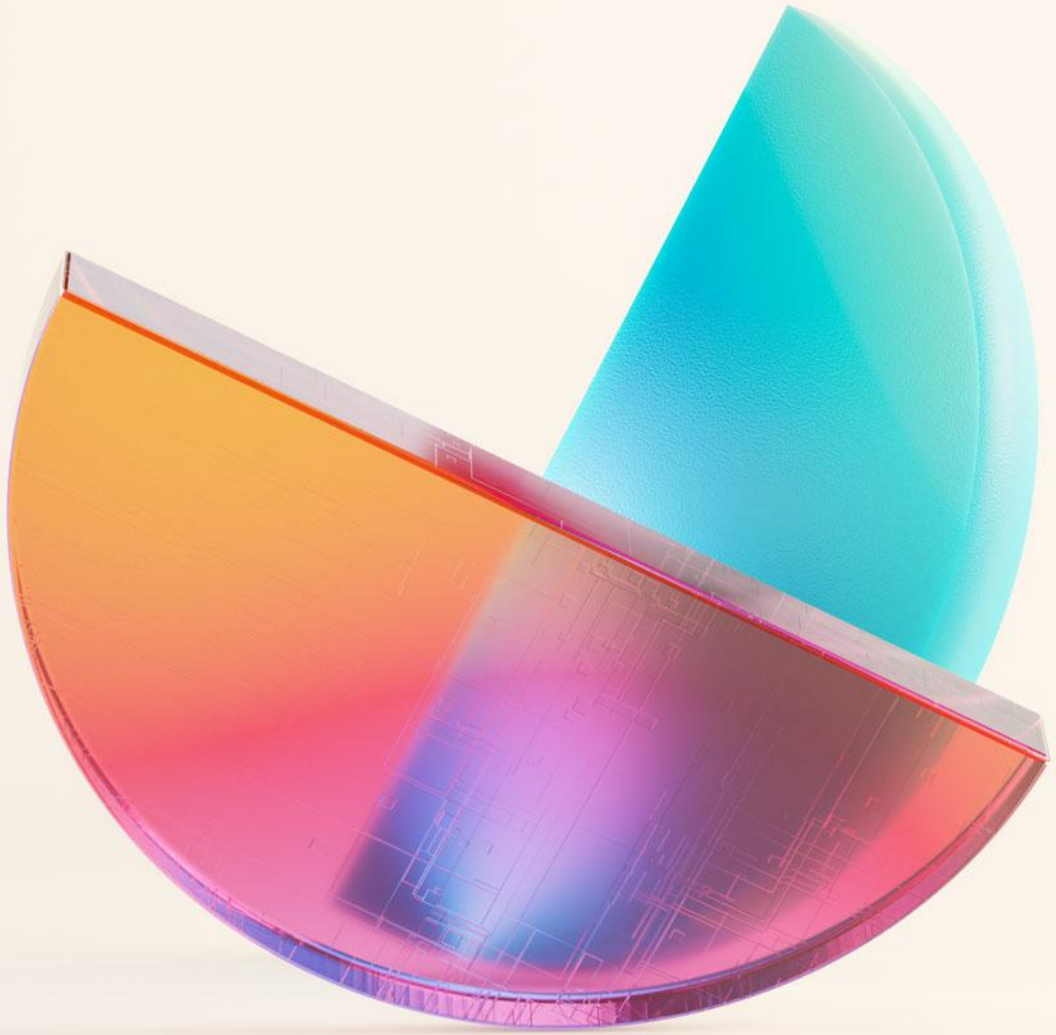
Microsoft



Microsoft Fabric Community Conference

Microsoft

Microsoft Fabric
Community Conference



Optimizing Fabric Spark and Best Practices for Production Ready Workloads

Speaker Bio

Santhosh Kumar Ravindran

Microsoft Corporation

thisissanthoshkumar

thisissanthoshkumar

Ashit Gosalia

Microsoft Corporation

ashit-gosalia

Narsimhan Bramadesam

London Stock Exchange Group

narsimhanbramadesam

Microsoft Fabric

The unified data platform for AI transformation

Data Factory

Analytics

Databases

Real-Time Intelligence

Power BI

Fabric Platform

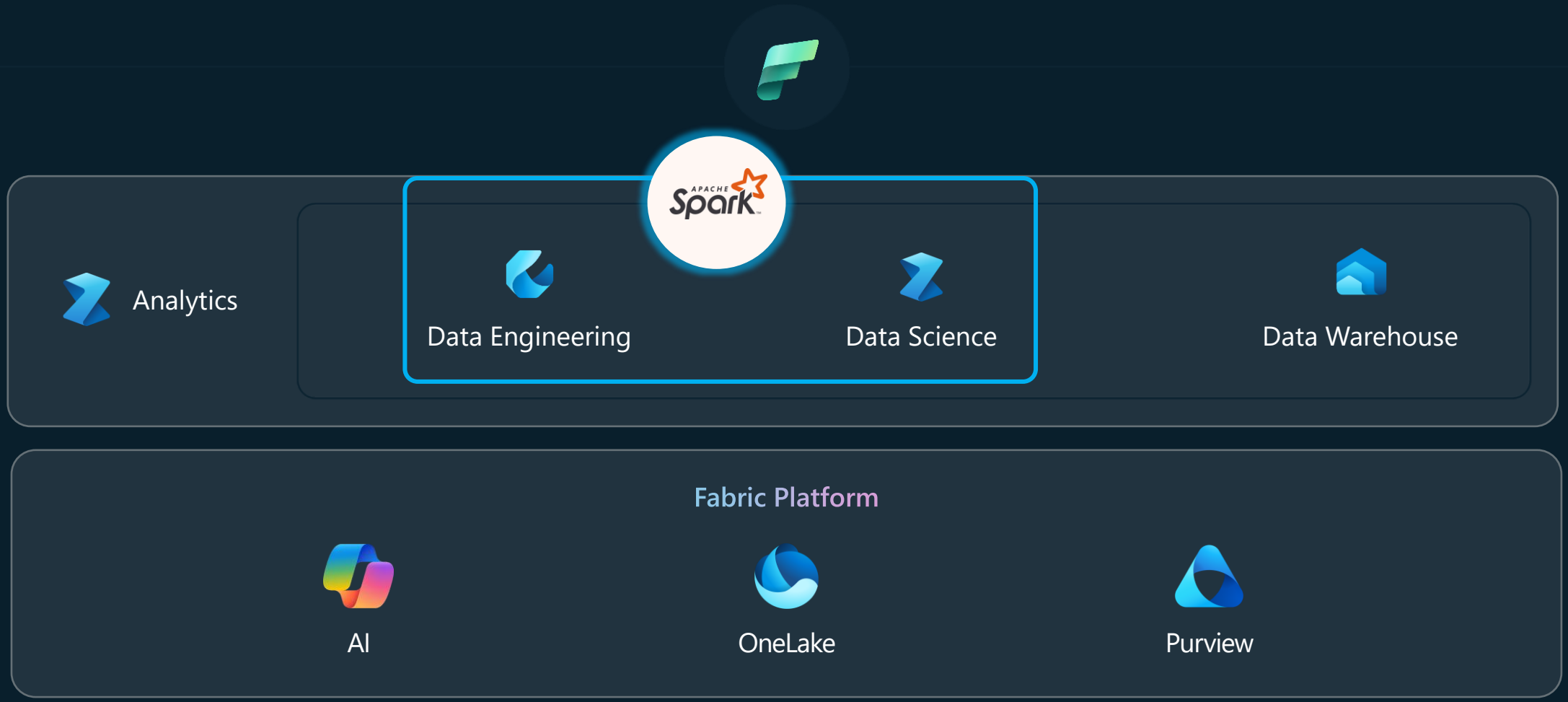
AI

OneLake

Purview

Microsoft Fabric

The unified data platform for AI transformation



Agenda

01

Introduction

02

Spark in Microsoft Fabric

03

Understanding your workload

04

Achieving the best Price Perf on Microsoft Fabric

05

LSEG's Fabric Journey: Scaling Data Analytics & Engineering at Enterprise Scale

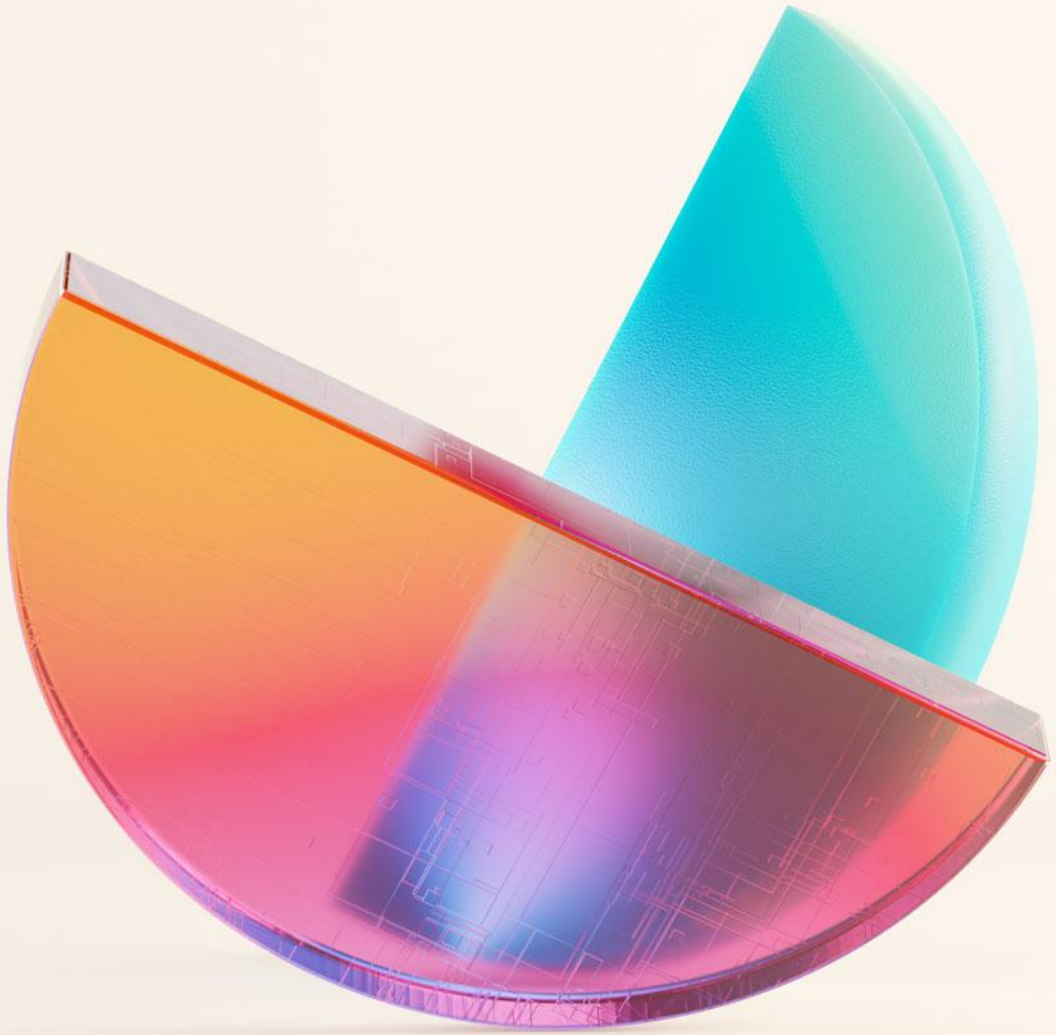
06

Q & A

07

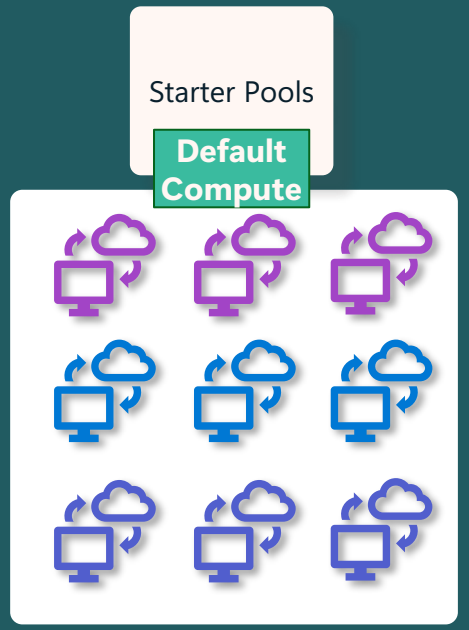
Microsoft

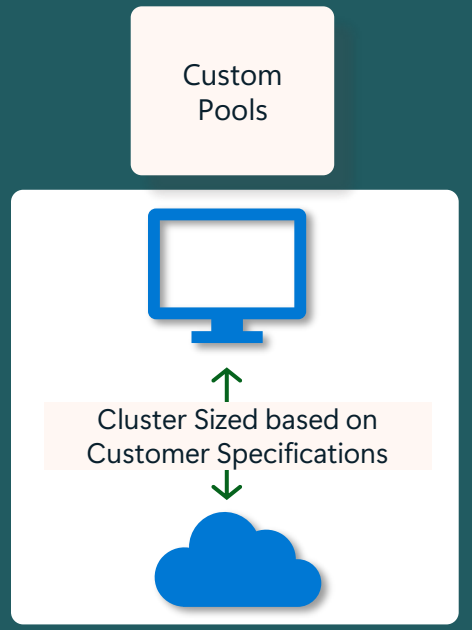
Microsoft Fabric
Community Conference



Spark Compute

Spark Compute is a powerful and flexible way of using Spark on Microsoft Fabric.
It enables you to perform various data engineering and data science tasks on your data stored in OneLake or other sources





Spark Compute Platform

Understanding Spark Session Startup Times in Microsoft Fabric

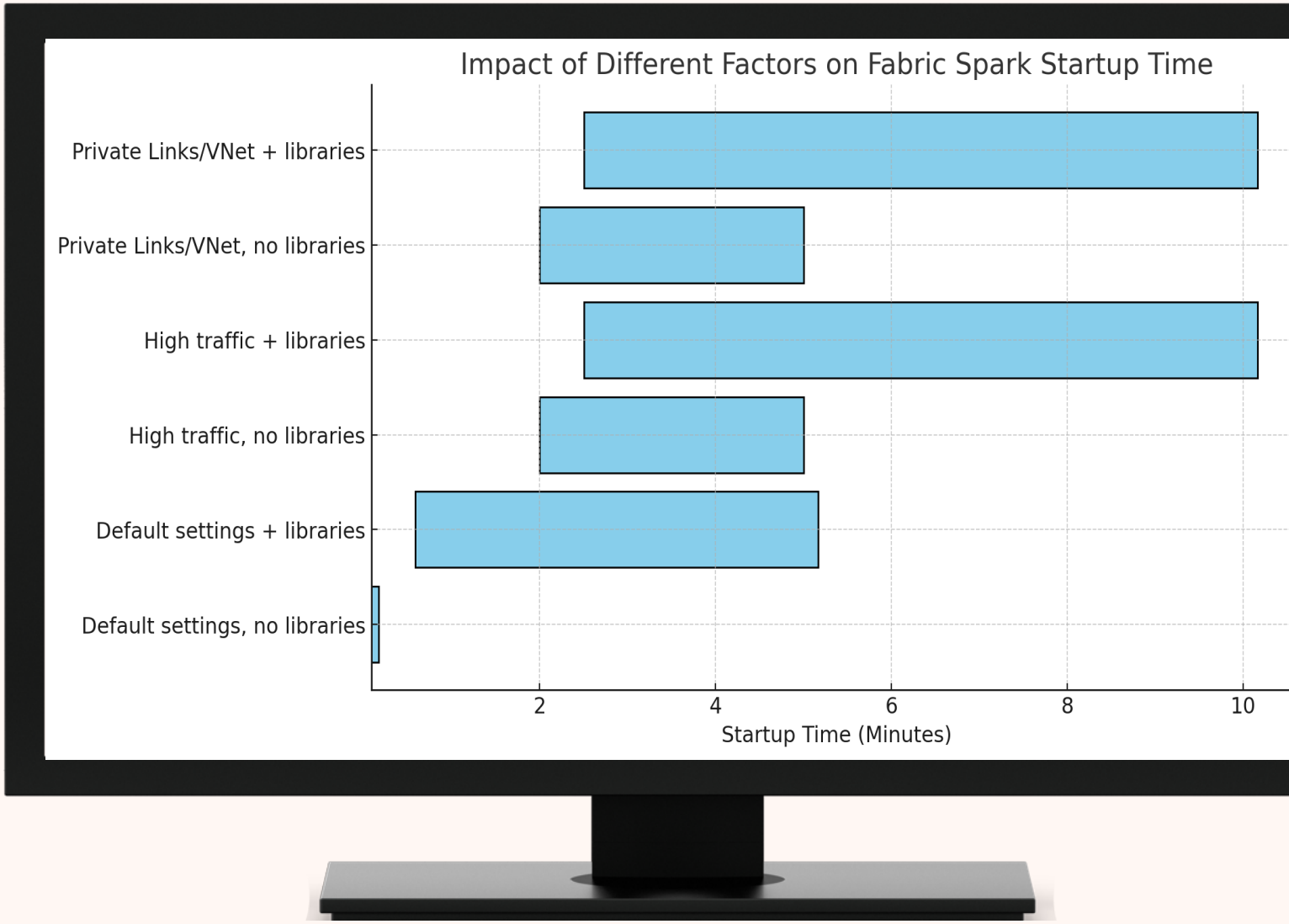
Factors Affecting Startup Time:

✅ **Fastest (5-10 sec):**

- Starter Pool Available
- No Custom Libraries or Spark Properties

⚠️ **Potential Delays (30 sec - 5 min+):**

1. **Custom Libraries & Spark Properties** → Extra setup time (30 sec - 5 min)
2. **Unpredictable High Traffic in Region** → New cluster provisioning (2 - 5 min)
3. **Advanced Networking (Private Links / Managed VNet)** → No Starter Pools (2 - 5 min)
4. **Combination of Factors** → Delays add up



What Spark performance challenges are you currently facing that limit your ability to scale workloads in Fabric?

📝 **We'd love to hear your biggest blockers — drop your thoughts in the open text survey!**

Spark Resource Profile Configs

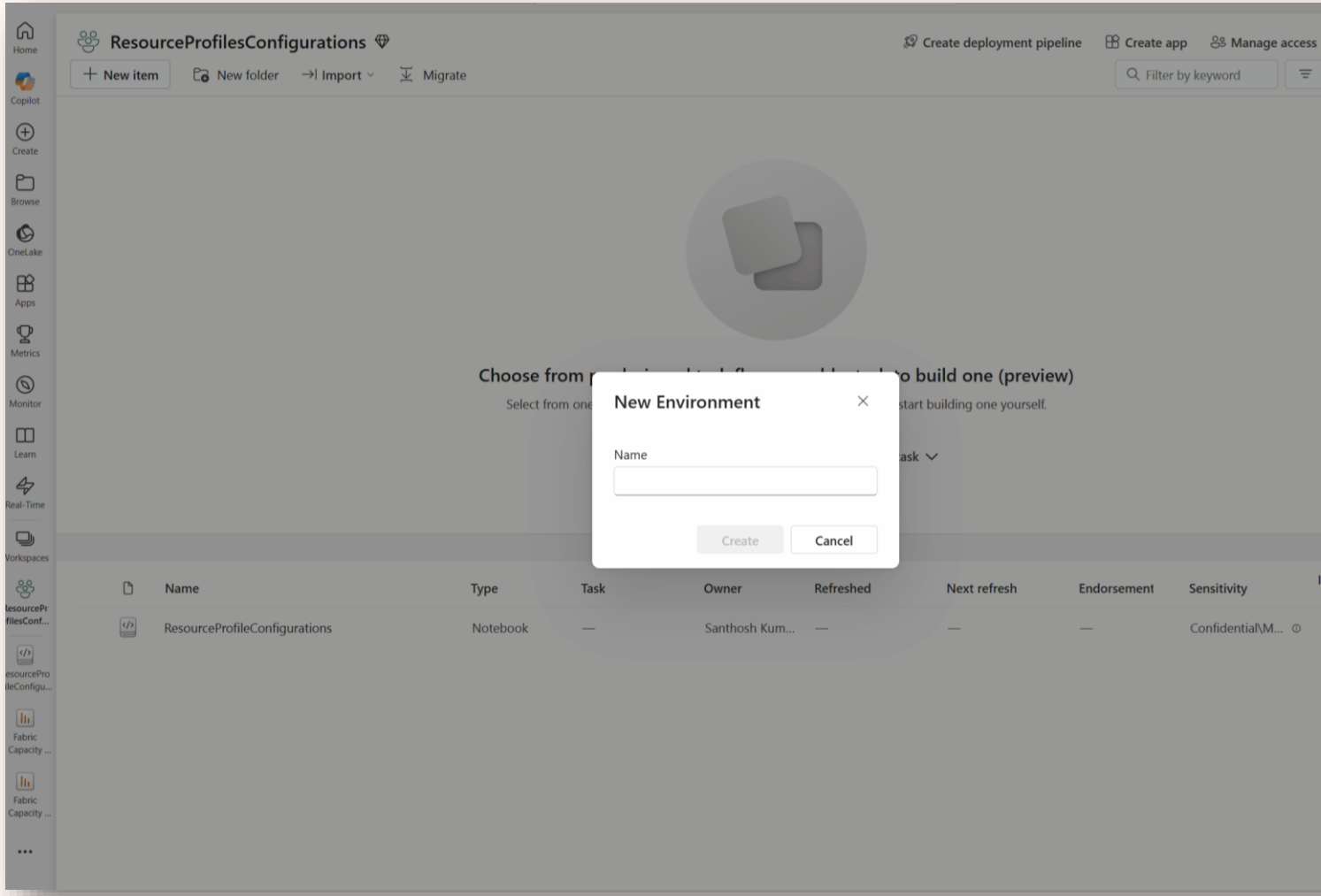
New Release

Out of the box configs for optimizing your Spark workload

Choose a **resource profile to suit your workload needs** (bronze, silver or gold)

Why this matters:

- **Performance by default** – no manual tuning required
- **Flexibility** – switch your profile based on workload
- **Fine tuned resource allocation** – ensure best performance for reads, writes and mixed workloads



Introducing Property Bags for Coordinated Customization of Sessions

Property Bag based approach which allows data engineers to customize their sessions based on different spark configurations

Fabric now provides **predefined resource profiles** to optimize Spark compute **based on workload type.**

New Spark Fabric Resource Profiles:

- **Easily configure workspace defaults for different workloads**

Profile	Use Case	Configurable Setting
ReadHeavyForSpark	Optimized for Spark workloads with frequent reads	spark.fabric.resourceProfile = readHeavyForSpark
ReadHeavyForPBI	Optimized for Power BI queries on Delta tables	spark.fabric.resourceProfile = readHeavyForPBI
WriteHeavy	Optimized for high-frequency data ingestion & writes	spark.fabric.resourceProfile = writeHeavy
Custom (ingestHeavy, writeToConformance or any profile based on your requirements**)**	Fully user-defined configuration	spark.fabric.resourceProfile = MyProfile

🧪 Example: Custom Resource Profile

Use Case:
Create a profile that is optimized for data conformance pipelines that require adaptive execution, controlled joins, and schema enforcement at scale.

✅ Where Can You Apply Resource Profiles?

🔹 Session-Level Configuration

Use %configure in notebooks to set a resource profile for a specific Spark session.

🔹 Environment-Level Configuration

Assign a profile to an environment to standardize behavior for pipelines or notebooks.

🔹 Workspace Default Configuration

Define a default profile for all jobs in the workspace to ensure consistency and governance.

```
[1]  spark.conf.set("spark.fabric.resourceProfile", "conformanceCheck")
```

✓ 2 min 45 sec - Session ready in 2 min 41 sec 805 ms. Command executed in 4 sec 248 ms by Santhosh Kumar Ravindran on 8:18:20

```
[2]  spark.conf.set(
       "spark.fabric.resourceProfile.conformanceCheck",
       """{
         "spark.sql.shuffle.partitions": "200",
         "spark.sql.adaptive.enabled": "true",
         "spark.sql.adaptive.shuffle.targetPostShuffleInputSize": "64MB",
         "spark.sql.broadcastTimeout": "600",

         "spark.sql.caseSensitive": "false",
         "spark.sql.parquet.filterPushdown": "true",
         "spark.sql.parquet.mergeSchema": "false",
         "spark.sql.files.ignoreCorruptFiles": "false",
         "spark.sql.sources.partitionOverwriteMode": "dynamic",

         "spark.executor.memoryOverhead": "512",
         "spark.sql.execution.arrow.pyspark.enabled": "true",
         "spark.sql.autoBroadcastJoinThreshold": "-1",

         "spark.sql.queryExecutionListeners": "org.apache.spark.sql.util.QueryExecutionListe
         "spark.databricks.queryWatchdog.enabled": "true",

         "spark.sql.join.preferSortMergeJoin": "true",
         "spark.databricks.io.cache.enabled": "true"
       }"""
     )
```

✓ <1 sec - Command executed in 1 sec 90 ms by Santhosh Kumar Ravindran on 8:33:19 AM, 4/01/25

New Fabric Workspaces – Optimized for Write-Heavy Workloads

🚀 **Newly created Fabric workspaces** now default to **writeHeavy** profile for best data ingestion performance.

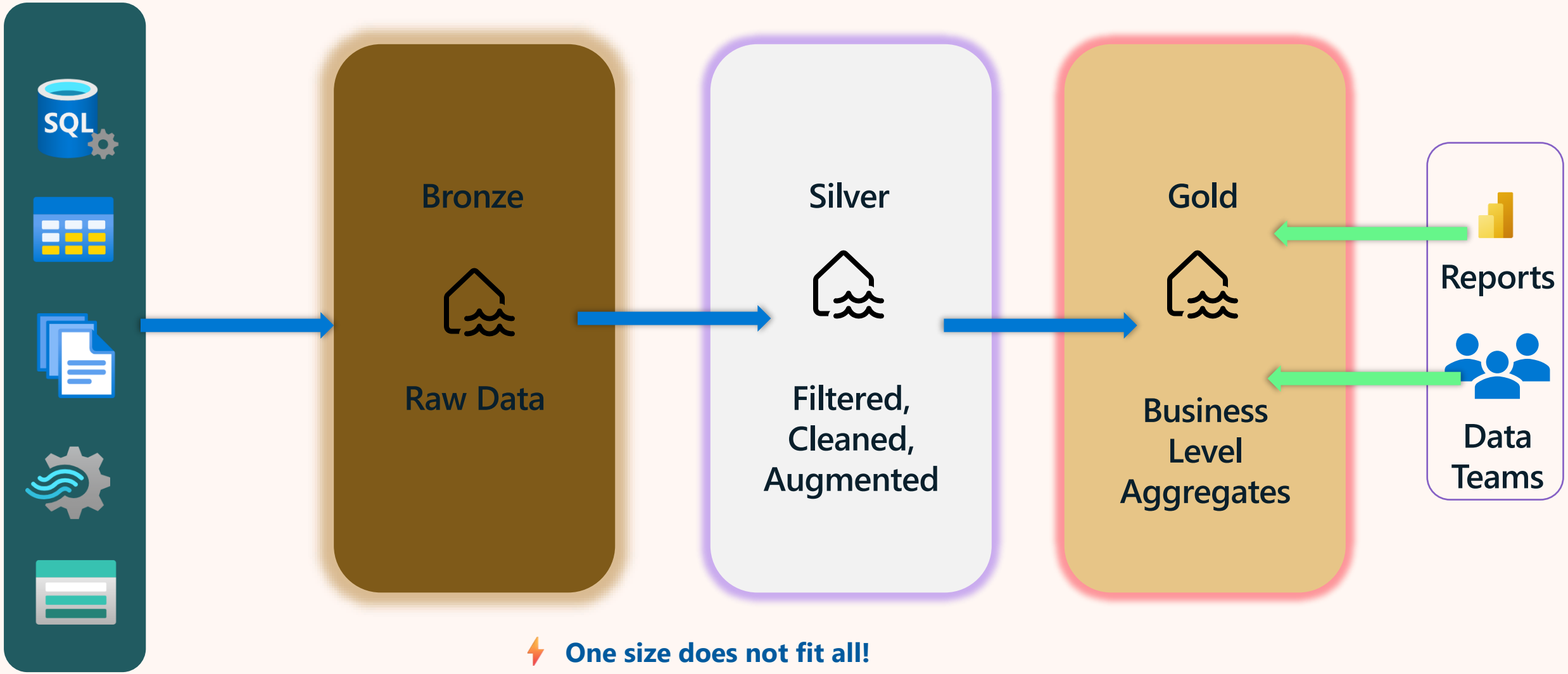
✓ **Performance by Default** – Faster ingestion without extra tuning

✓ Optimized for Data Engineering Jobs – Best for **large-scale ETL & pipeline processing**

📌 **Takeaway:** New workspaces come **pre-tuned for write-heavy scenarios**, but users can adjust configurations anytime to fit their workloads.

Configuration	Value	Effect
Spark.sql.parquet.vorder.default	false	Keeps writes sequential for better ingestion throughput
Spark.databricks.delta.optimizeWrite.enabled	false	Disables row-level optimizations to favor bulk writes
Spark.databricks.delta.optimizeWrite.partitioned.enabled	true	Enables partitioned optimize write for structured ingestion
Spark.databricks.delta.optimizeWrite.binSize	128mb	Efficient bin size for streaming & bulk writes
Spark.databricks.delta.stats.collect	false	Reduces write overhead by skipping automatic stats collection

Tailoring Compute for Your Lakehouse



One size does not fit all!

The compute and optimization settings you choose should align with **how data is processed at different layers of the Lakehouse.**

Configurations for Bronze Layer

Bronze

Raw Data

🚀 The **Bronze Layer is ingestion-heavy**, requiring **high throughput writes** and **efficient storage management**.

📌 **Why?**
✓ **Ensures high-speed ingestion** for streaming & batch data
✓ Optimized for **write performance over query speed**
✓ Recommended for Bronze Layer where raw data is stored as-is

```
spark.fabric.resourceProfile = writeHeavy
```

Configurations for Silver Layer

Silver

Filtered, Cleaned, Augmented

⚡ The **Silver Layer requires a balance between reads & writes** for data transformations and cleansing based on the dependency on data serving layers

spark.fabric.resourceProfile = Conformance

Configuration	Value	Effect
Spark.sql.parquet.vorder.default	true	Columnar storage format optimized for mixed workloads
Spark.databricks.delta.optimizeWrite.enabled	true	Enables row-level optimizations for transactional updates
Spark.databricks.delta.optimizeWrite.partitioned.enabled	true	Allows partitioned writes for optimized performance
Spark.databricks.delta.optimizeWrite.binSize	256mb	Provides balance between write performance & queryability
Spark.databricks.delta.stats.collect	true	Collects table statistics for better read optimization

Medallion Magic: Tailoring Compute for Your Lakehouse

Gold

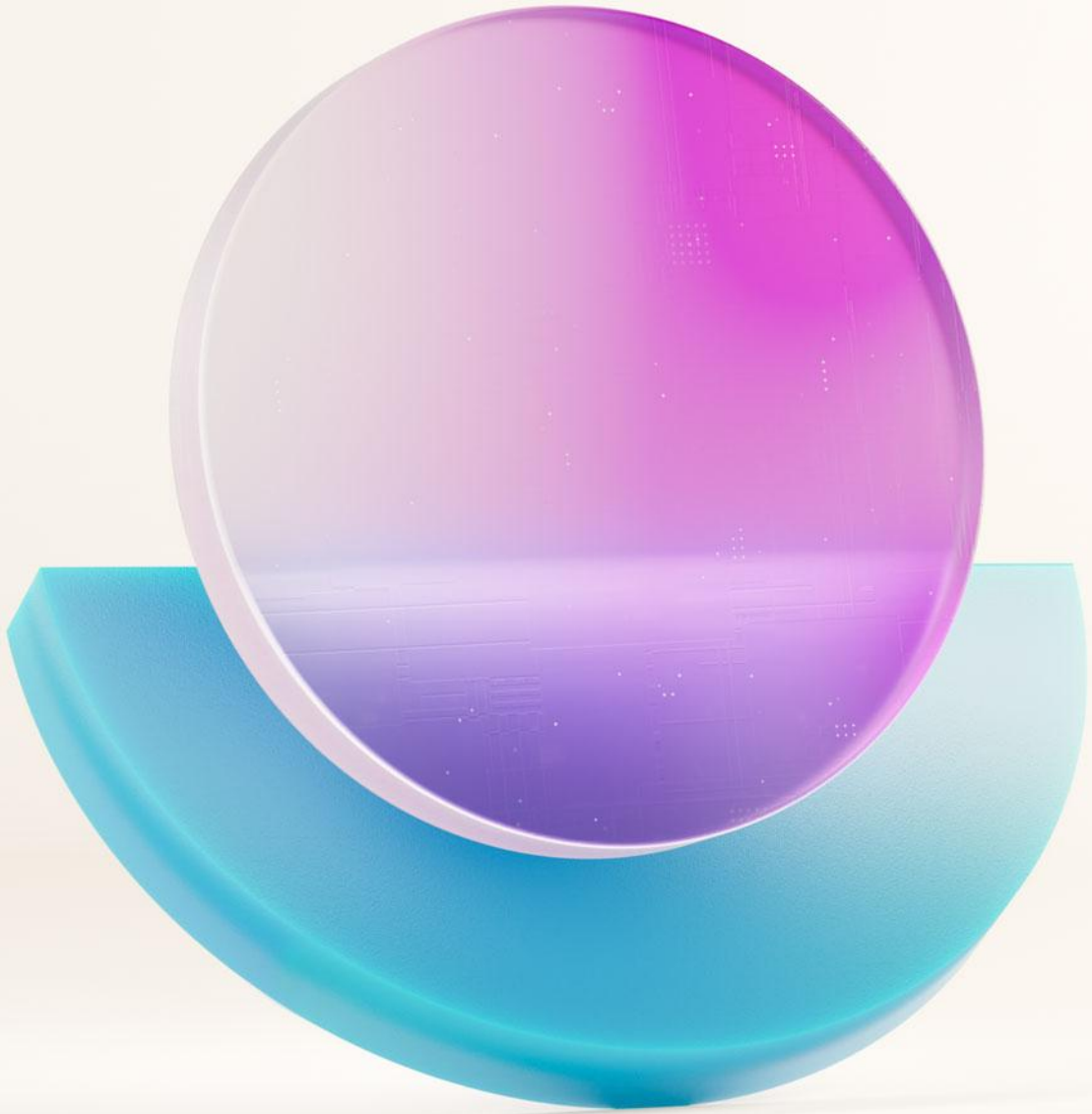
Business Level Aggregates

📊 The **Gold Layer powers analytics & reporting**, requiring fast queries with optimized storage.

✓ Optimized for **fast query execution** & **Power BI** workloads
✓ Improves aggregation performance on large datasets
✓ Recommended for **Gold Layer** where data is consumed for business intelligence & reporting

spark.fabric.resourceProfile = readHeavyForPBI

Microsoft Fabric
Community Conference

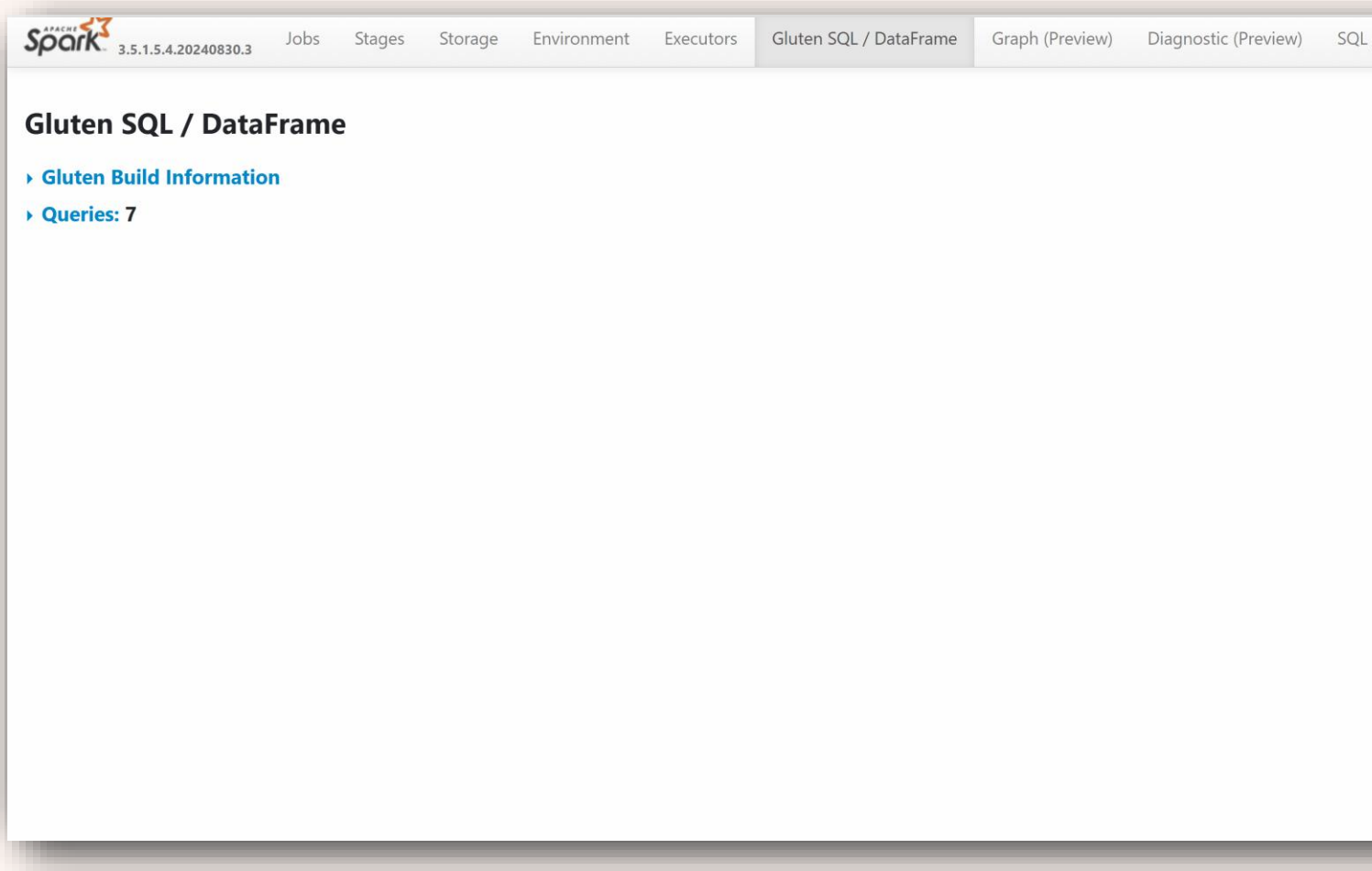


Achieving the best Price Performance

Native Execution Engine

About ~4x performance improvement over OSS Spark with No Additional Cost

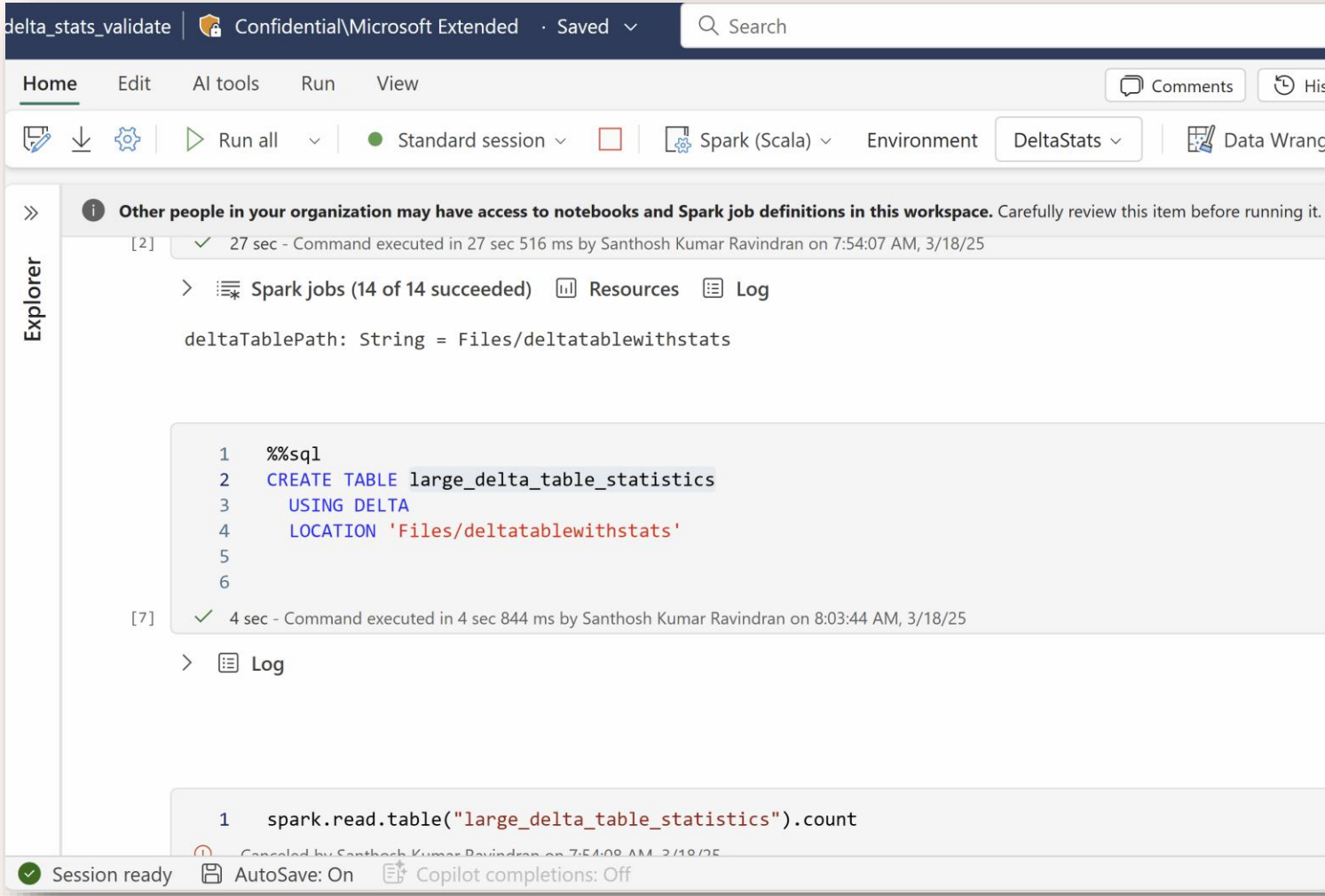
- Native, vectorized engine **written in C++ delivering** up to 4x improvements
- Based on **OSS projects - Velox & Gluten**
- No need to change code any - **Fully compatible with Apache Spark APIs**
- Can be easily turned on in the **new 'Acceleration' tab** of the Environment item
- Connect to sources behind secure networks through Managed Private Endpoints and with the support for Private Links
- SET spark.native.enabled=FALSE;



Automated Table Statistics

Sneak Peek

- **~45% performance gains** with automatic table statistics
- **Automatically** calculated for every Delta table created with Spark → No manual intervention needed
- **Optimizes query planning & execution** → Faster reads, reduced compute overhead
- Extended Stats for columns include **distinct values** at the table level



Autoscale Billing for Fabric Spark

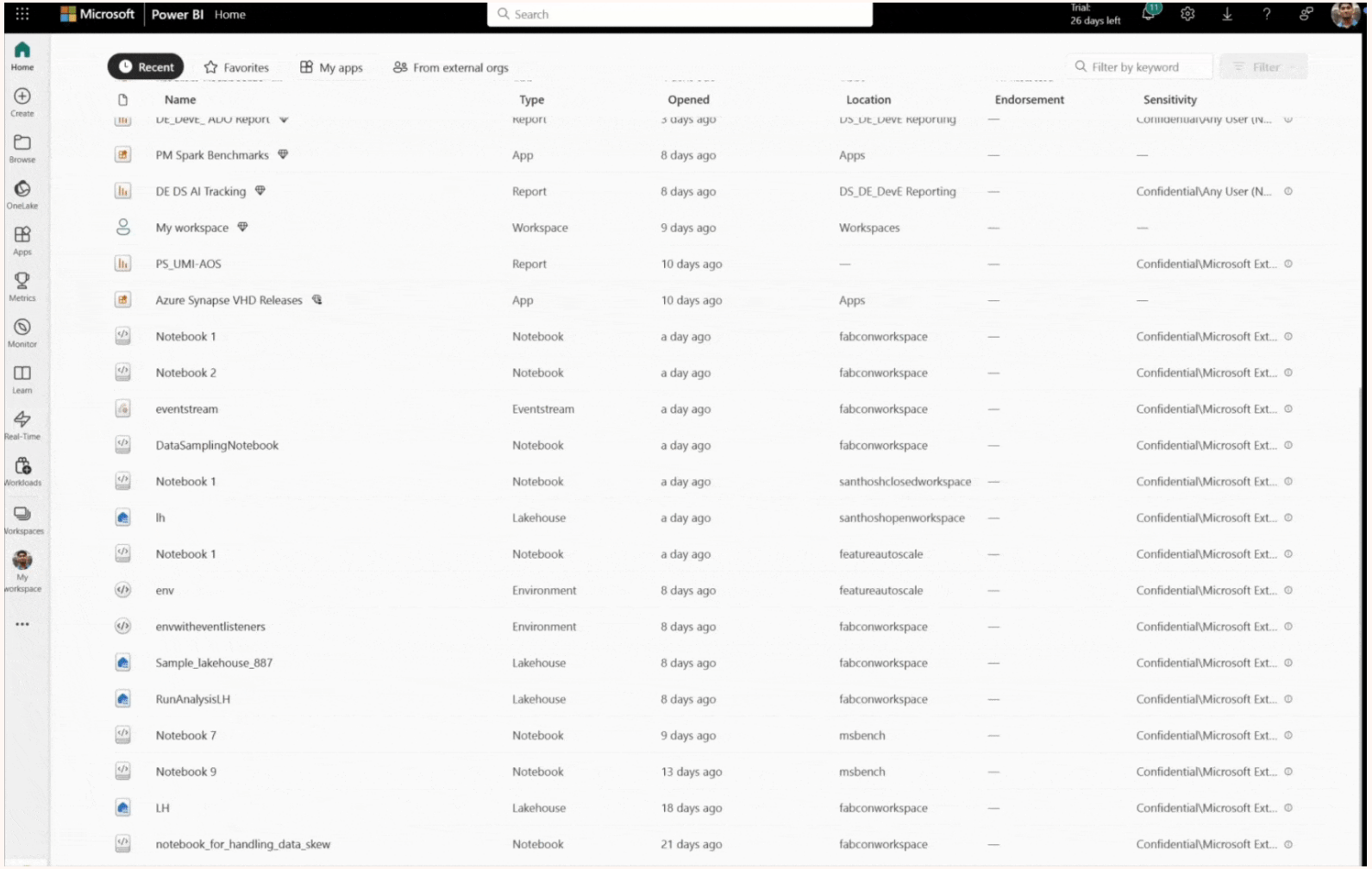
Public Preview

Flexible, Pay-as-You-Go Autoscale Billing for Spark

Run Spark independently of shared Fabric capacity where you are only billed for active Spark jobs.

Seamless Integration – Monitor usage for Autoscale billing in the **Fabric Capacity Metrics App** and request more capacity units via **Azure Quota Management.**

Optimized Resource Management – Spark jobs scale separately from base capacity, reducing contention for shared resources and ensuring smoother execution of dynamic, compute-intensive workloads.

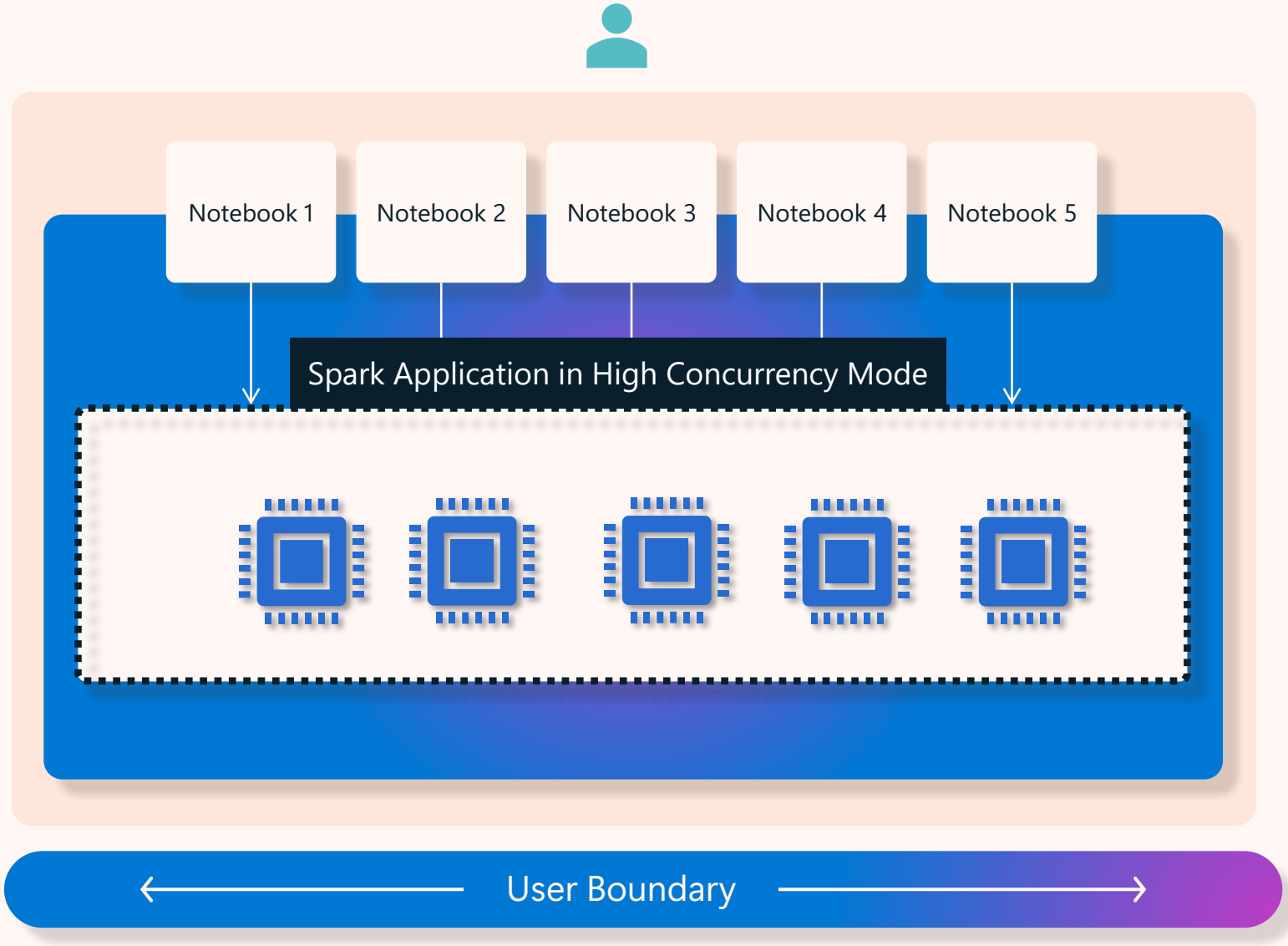


Session Sharing Within A Single Spark Application

- **Security** : Session sharing is always within a single user boundary.

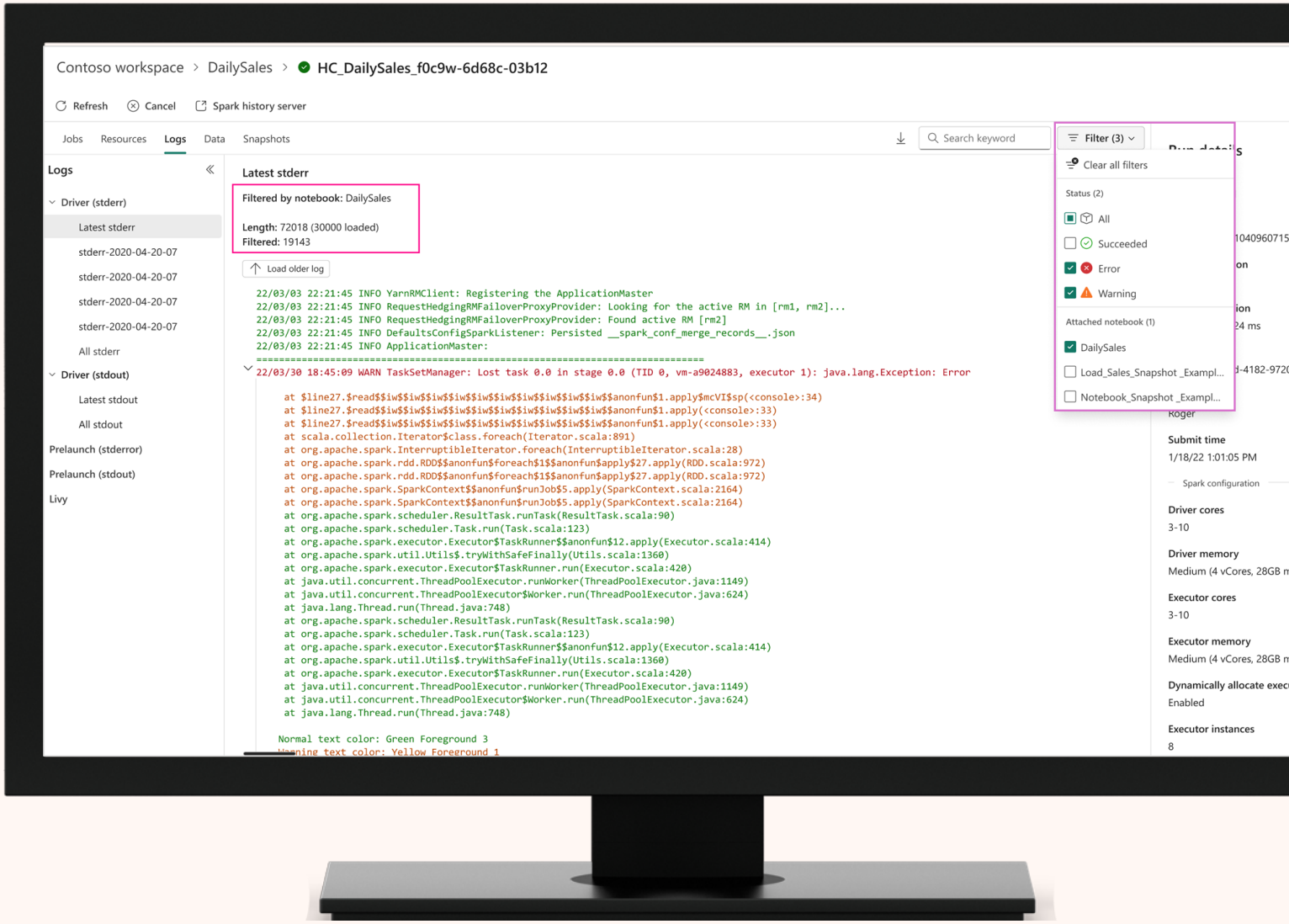
- **Multitask**: Users can seamlessly switch between notebooks and continue their work without experiencing any delays due to session creation or initialization.

- **Cost-effective**: Allows users to achieve better resource utilization and cost savings for their data engineering / science workloads.



High Concurrency Mode in Notebooks

- High Concurrency Mode enables multiple notebooks to share Spark sessions, reducing session startup times to as low as 5 seconds ‘
- In the case of Custom Pools this makes the session start **36X** faster for notebooks attaching to an existing session
- Notebooks get automatically assigned to available Spark High Concurrency sessions, eliminating the session start latency
- Updates:
  - Snapshot support for active running notebooks with High Concurrency mode
  - **Log separation** and **granular monitoring** experience for better debuggability

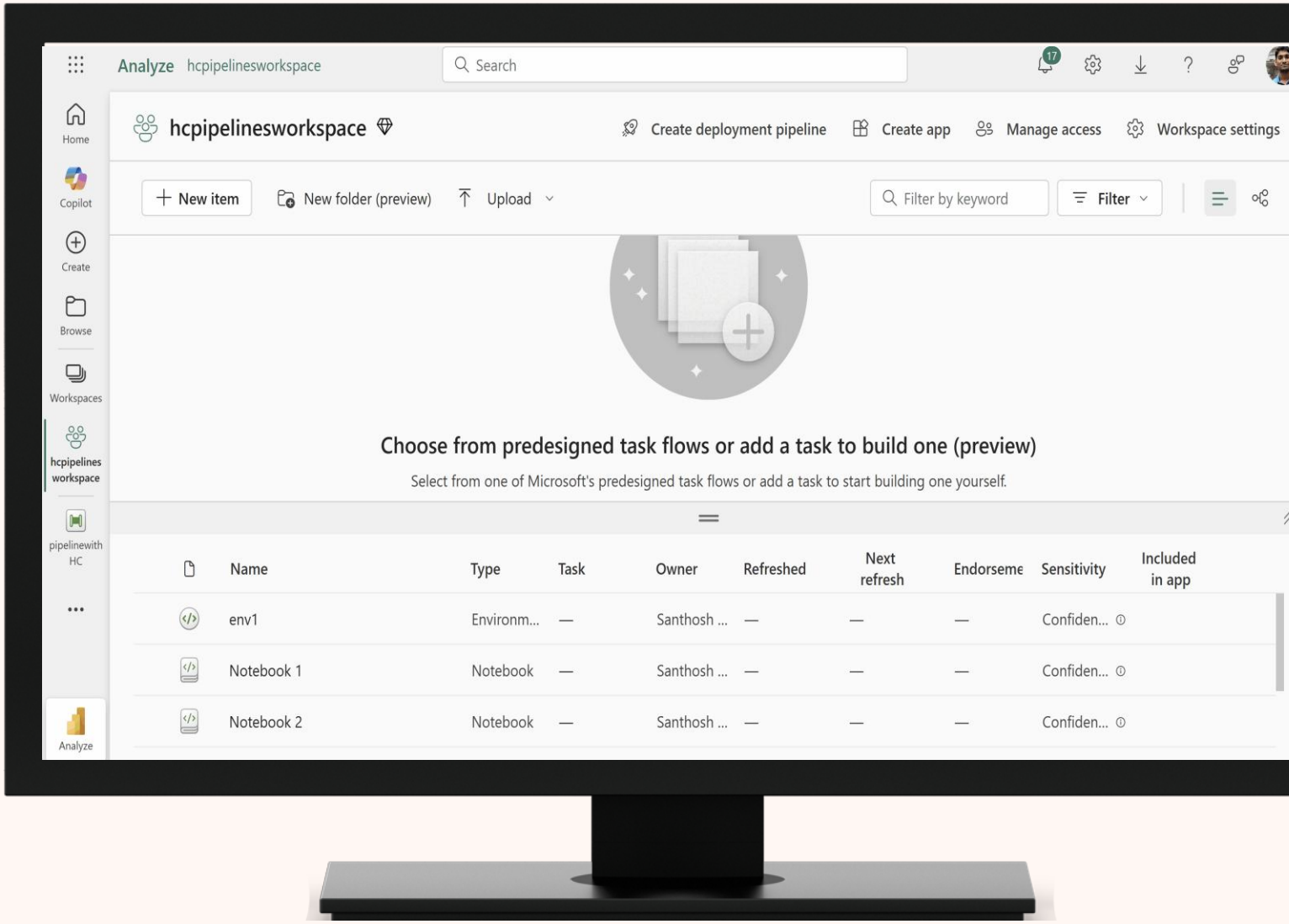


High Concurrency for Notebooks in Pipelines

GA

Get started with running Spark notebooks in pipelines in a matter of seconds through session sharing

- High concurrency mode enables users to share Spark sessions across multiple notebooks steps in pipelines
- Attaching a notebook to an existing session results in lightning-fast session start up times
- Use **session tag** to pack notebooks to an active spark session



Python Compute – When Spark is overkill

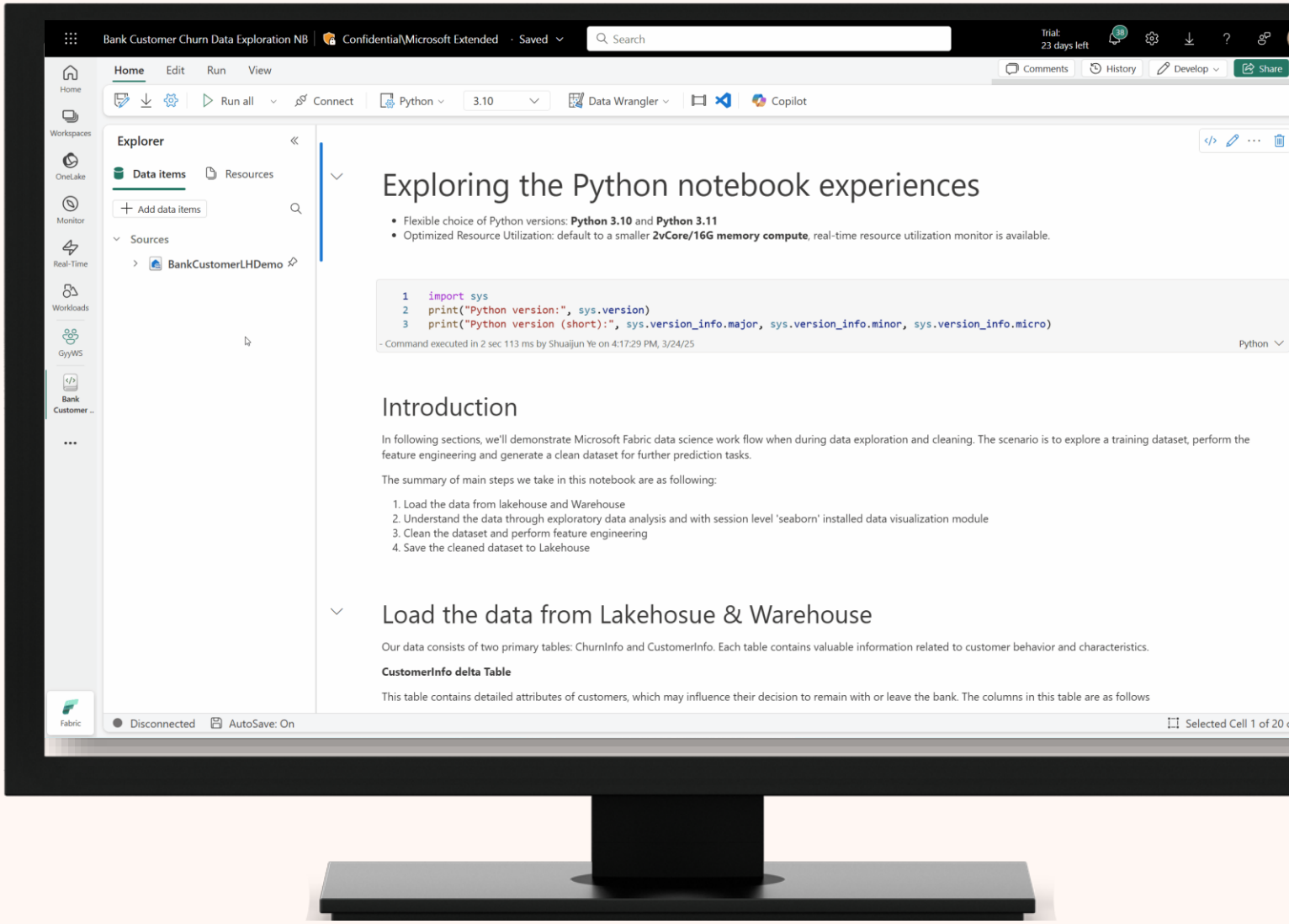
⚡ **When Spark is Overkill, Use Python Compute!**

✅ Comes with a **2-core VM** – perfect for lightweight data querying and orchestration sceanrios.

✅ Ideal for **API calls, deployment orchestration**.

✅ **Lower cost option** compared to spinning up a full Spark cluster.

Better resource utilization for smaller datasets, resulting in increase concurrent jobs and improved capacity utilization



Optimizing for Performance and Reliability

The **spark.task.cpus** configuration controls the number of CPU cores allocated to each Spark task.

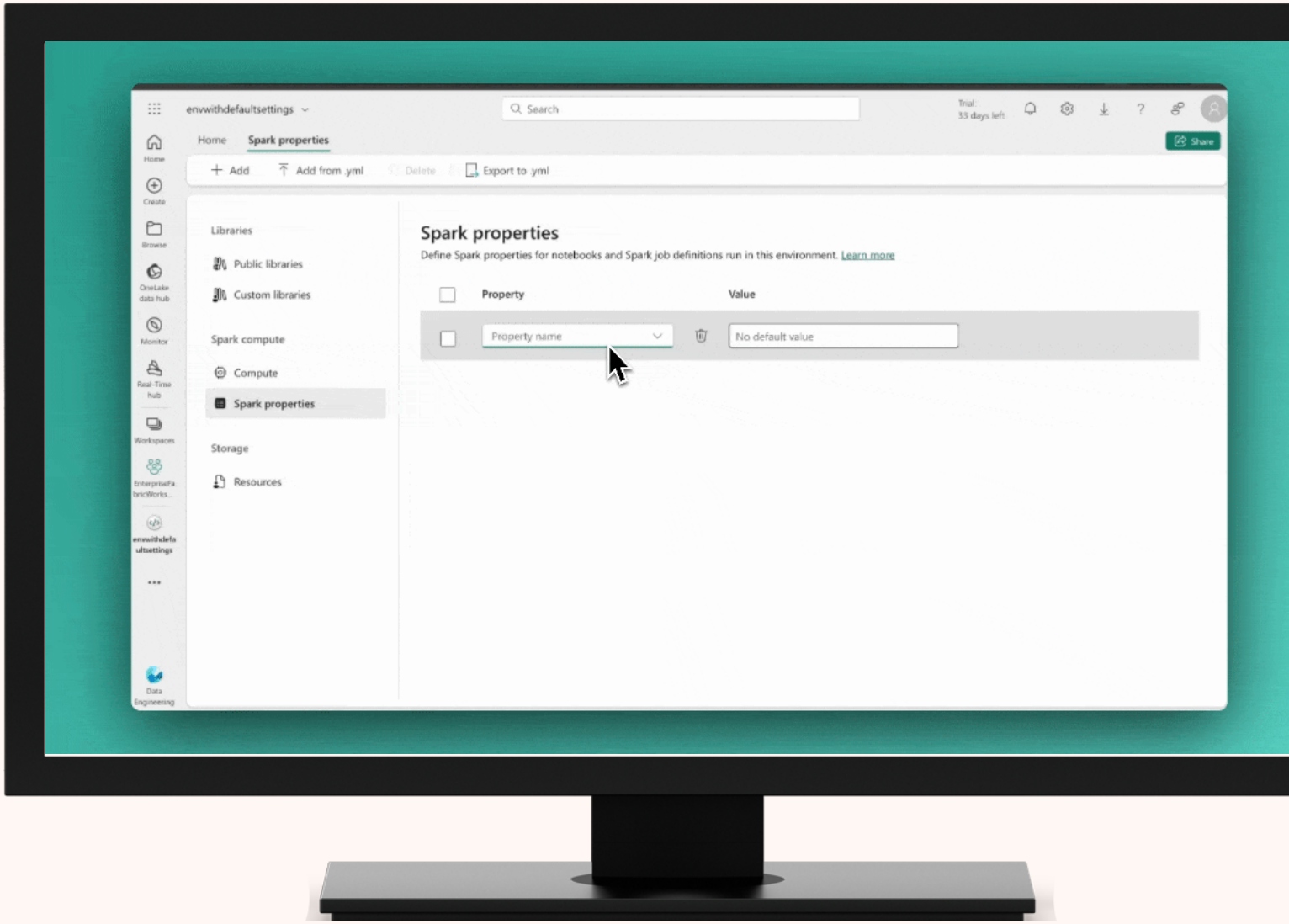
The default value is 1, meaning each task utilizes a single CPU core.

Smaller values (e.g., 0.5):

Can improve performance for jobs that are **CPU-bound** but don't require significant task parallelism or memory.

Larger values (e.g., 2 or higher):

Can enhance **job reliability** by providing more resources to fluctuations in workload.



Improve Time to Solution Using Livy Endpoint

Submit your Spark code via public Livy endpoint

Programmatically submit and run Spark code without the need to create any Notebook/Spark Job Definition

Support both session and batch job

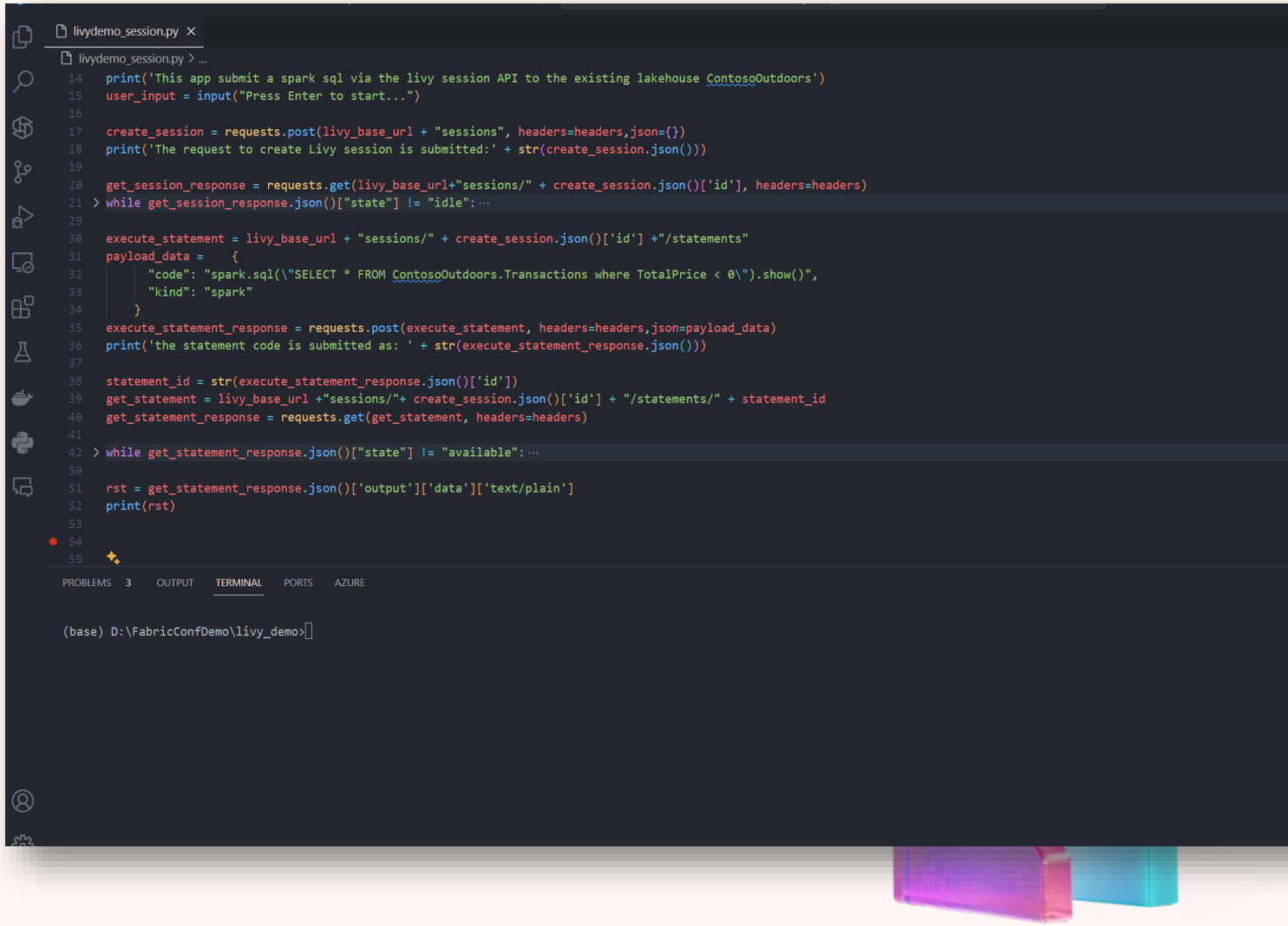
Deep integration with Lakehouse for data access in One Lake

Integration with Environment artifact to allow customizing runtime profile

Roadmap:

High Concurrency Support in Livy

SPN Support in Livy



Spark DW Connector

GA

Empowers Spark developers to work with Fabric DW data using Spark

Simplified PySpark/Scala API, hiding complexity of specifying configurations

Read pushes filter & column predicates to warehouse and Write Moves data in parallel and honors transactions

Allows reading of data from/to warehouse across workspaces

Honors security definitions of warehouse

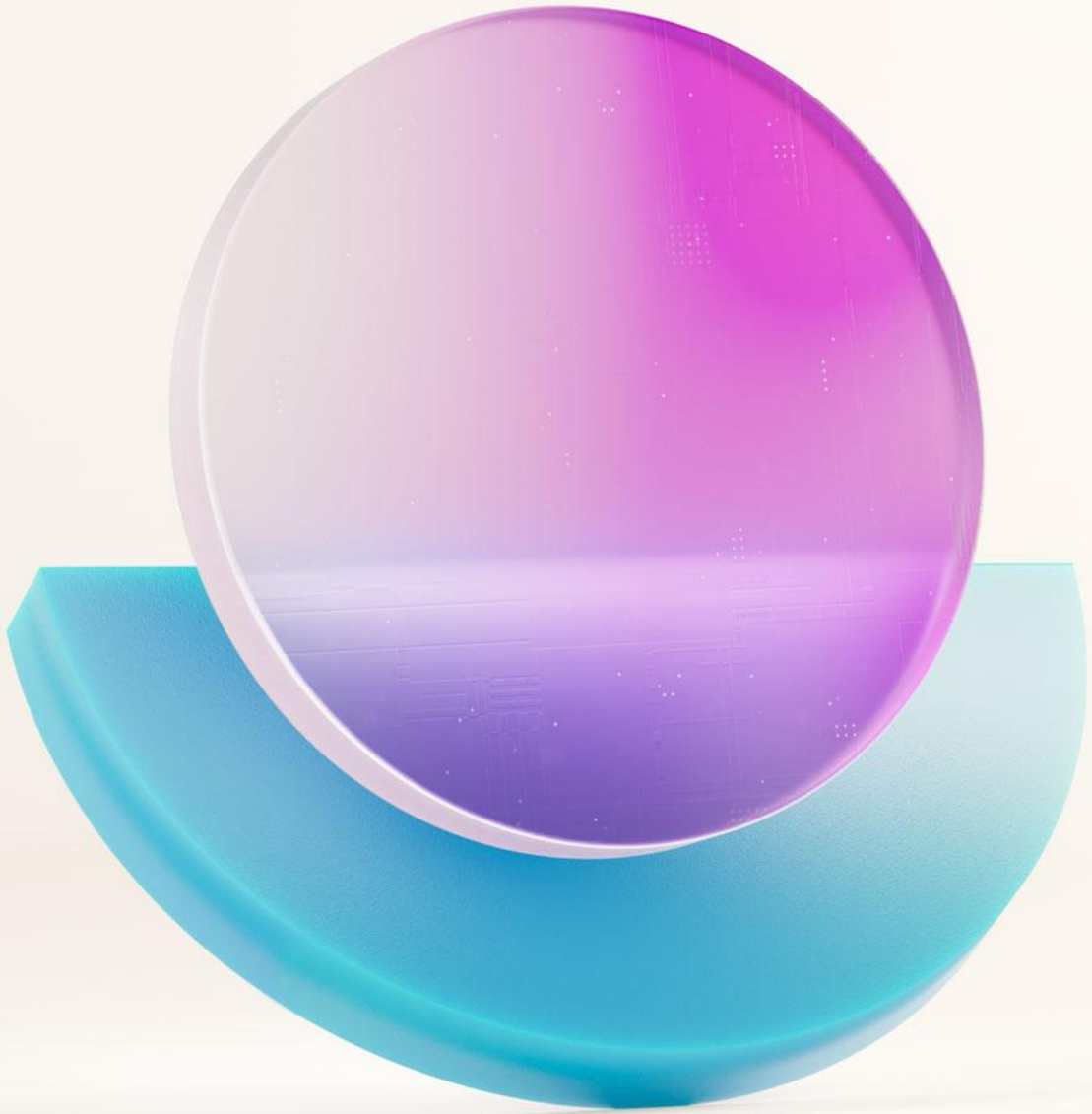
New Releases:

Write back support to the Data Warehouse and full PySpark support

Roadmap:

Azure SQL DB Connector for Spark

Microsoft Fabric
Community Conference



LSEG’s Fabric Journey: Scaling Data Analytics & Engineering at Enterprise Scale

About LSEG | London Stock Exchange Group

- LSEG is a **leading global financial markets infrastructure and data provider** that operates connected businesses to serve customers across the entire financial markets value chain.
- Together, our five business divisions offer customers seamless access to global financial markets, **across the trading lifecycle.**

LSEG DATA & ANALYTICS

FTSE RUSSELL
An LSEG Business

LSEG RISK INTELLIGENCE

LONDON STOCK EXCHANGE
An LSEG Business

LSEG POST TRADE

Data & Analytics | The Financial Industry's Most Comprehensive and Trusted Content

- One of the world's leading providers of financial information, we deliver **high-value data, analytics, indices, market surveillance, and solutions** to manage risk, workflow and data.

50 MILLION
Estimates & KPIs annually

2.7 MILLION
Fixed income instrument evaluations daily

15 MILLION
Private companies

62,000
Active public companies

140 BILLION
OTC Ticks per year

59 MILLION
AMR pages annually

700,000
Equity and Fixed income indices covered

13.4 MILLION
Fixed income securities

14 MILLION
Economic time series

7.5 MILLION
Active exchange traded & OTC derivatives

250,000
Company events

145 MILLION
Company financial data points per year

8 MILLION
Financial price updates per second

65
Languages

215 BILLION
Market messages per day

1.3 MILLION
Equity indices covered

42 MILLION
Individual instruments or indicators up to 70 years of history

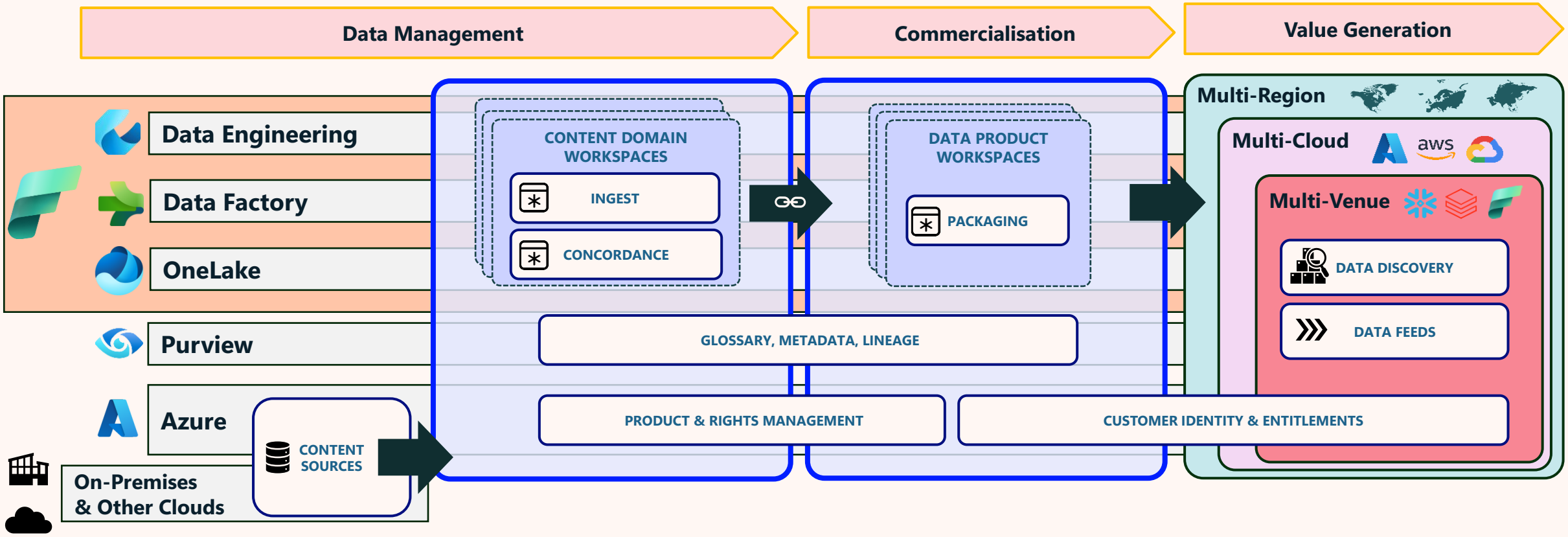
520,000
Equity quotes from 300+ exchanges

700 BILLION
Bond, Derivative and ETF trading supported via Tradeweb (Daily)

40,000
Company transcripts Annually

150,000
Deal transactions annually

Platform Architecture



Content is sourced, transformed and mastered across a range of proprietary and 3rd party solutions

Data Products are published aligned to LSEG's data model and commercially prepared with segments and rights defined

Data is commercially packaged into discoverable products that are governed and ready for distribution

Data is discoverable and distributed across a range of channels, clouds, venues and regions

Challenges and Optimizing Performance

Background: The initial release of our data platform, developed in collaboration with Microsoft, provides clients with access to LSEG's datasets.

During our first release and initial implementation of Fabric, we encountered challenges with environment setup, automated code deployment, and release management. Spark jobs faced performance bottlenecks and configuration issues.

Problem Statement: Enhance Spark job performance to meet KPIs and establish a scalable, consistent execution and monitoring.

Challenges

1. **Performance Bottlenecks**: Issues with capacity provisioning, inefficient resource usage, and Spark configurations (e.g., node size, executor settings).
2. **Technology Limitations**: Initial use of Spark and Delta for data pipeline orchestration, built on Fabric Spark notebooks and managed by Data Factory. Limited IaC capabilities. Limited debugging capabilities.
3. **Data Dependency Management**: Manual handling using static configurations.

Optimizing – Fabric Spark Job Performance

1. Baseline Measurement & Triage

- Baseline performance Metrics
- Inspect the logs
- Resource Utilization
- Code Review for inefficiency and optimizations

2. Evaluate Current setup and Data

- Local env. – Limited data to run spark under instrumentation
- Investigate file Sizes
- Investigate data Partitioning Impact on performance for the given dataset
- Look for data skews

3. Fine-Tune Spark Configuration

- Tune # of Executors
- Parallelism Settings
- Tune Executor memory
- Vorder Enabled / Disabled
- Optimized write enabled/disabled & other settings

4. Access Concurrency Factors

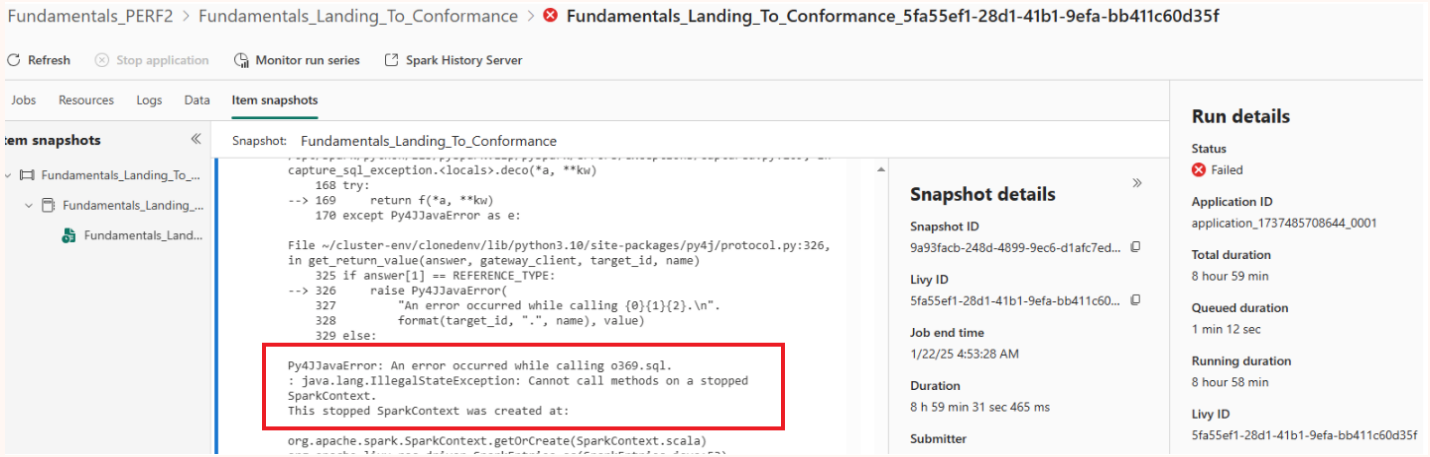
- Check for unnecessary DAG dependency
- Enough Capacity for concurrent Load Spike/jobs
- Experiment with Run Multiple as /High concurrency setting

Baseline Measurement & Triage

- Code review focus Areas:
  - ✓ Collection of Development-Level Statistics
  - ✓ Proper Error Handling and Monitoring (note: untested error handling led to an infinite loop)
  - ✓ Ensuring Code Coverage for Libraries and Notebooks
  - ✓ Avoiding repeated execution of terminal operations on non-cached Dataframes
  - ✓ Monitoring for Unbounded Cartesian Joins (pay attention to join conditions and keys)
- Inspect the Execution Plan and Runtime Statistics
  - ✓ Inspect the Execution Plan and Runtime Statistics
  - ✓ Analyze the Execution Metrics
  - ✓ Identify and Address Data Skew
  - ✓ Review the Execution Plan

Decoding Fabric Spark errors

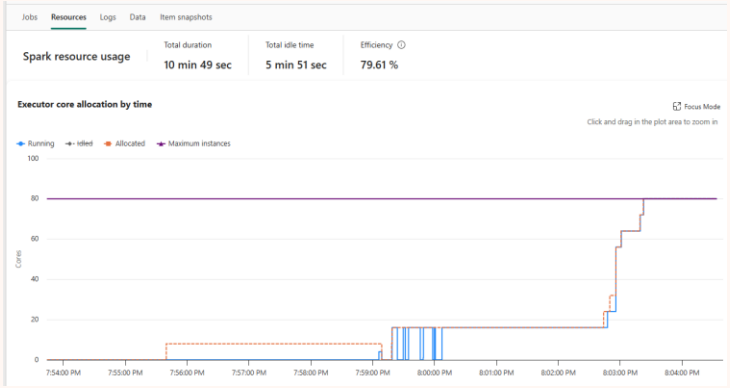
- **Livy errors**
  - "Failed to create Livy session for executing notebook".- Notebook execution failed at Notebook service with http status code - '200', please check the Run logs on Notebook, additional details - 'Error name - Exception, Error value - Failed to create Livy session for executing notebook.
- **Out of Memory error**
  - Large physical plan – error - Py4JJavaError: An error occurred while calling o367.sql.: java.lang.IllegalStateException: Cannot call methods on a stopped SparkContext.



Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Add Time	Remove Time
driver	vm-0a370513:44915	Active	0	0.0 B / 59.6 GiB	0.0 B	0	0	0	0	0	292.3 h (0.0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	2025-01-10 11:48:04	-

Decoding Fabric Spark errors

- **Out of Memory error**
  - The job ran for an extended period and eventually failed due to an out-of-memory error. Check spark.sql.shuffle.partitions and change the default.
- **Intermittent Memory Issues: Task fails due to timeout**
  - **Check for Data Skew –** our Initial implementation of Fabric used similar-sized nodes, try setting spark.task.cpus=2 in Spark to temporarily allocate more memory per task.
- **Checking for Resource utilization in Spark jobs**:
  - EX. The issue is indicated by a gradual ramp-up of resources(extensive small file listing), Typically, you should be able to quickly ramp up the required capacity.



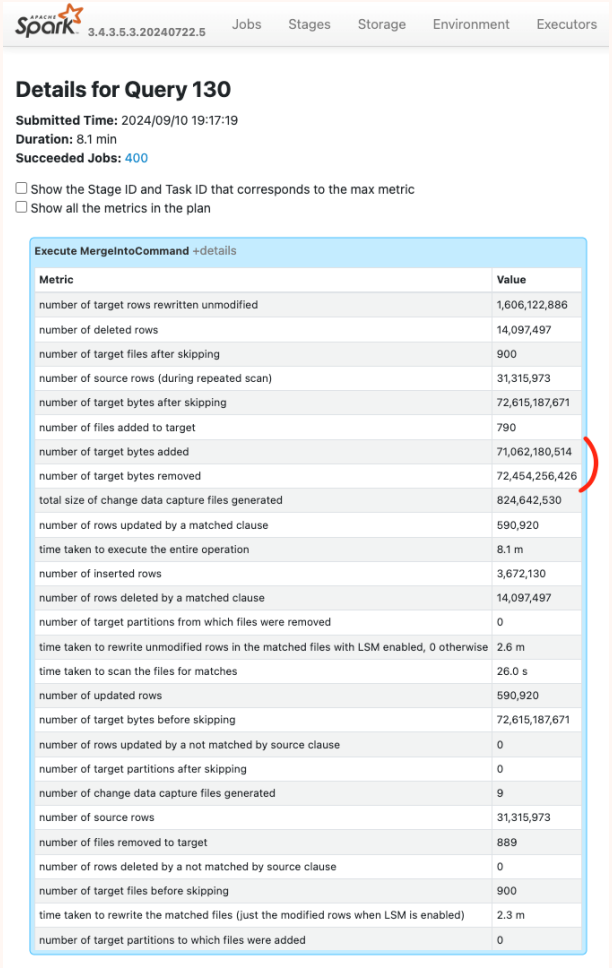
- **Timeouts due to lack of capacity** – If you're not a capacity admin and experiencing timeouts from long-running jobs, check capacity utilization if you have access. Alternatively, run a query in your lakehouse to get a clearer error message indicating insufficient capacity.

Performance Analysis and Benchmarking

Partitioning Data

Partitioning is crucial for optimizing performance in Spark. A poor partitioning strategy can lead to slow processing and inefficiencies. Here are key parameters to consider and steps to troubleshoot performance issues:

- ✓ Analyze your Merge Statistics to assess the amount of data rewritten in each update. By partitioning your data, you can make targeted merges faster and more efficient.
- ✓ The Spark History Server's merge query statistics clearly indicate that most rows are rewritten, even with minor changes.



Metric	Value
number of target rows rewritten unmodified	1,606,122,886
number of deleted rows	14,097,497
number of target files after skipping	900
number of source rows (during repeated scan)	31,315,973
number of target bytes after skipping	72,615,187,671
number of files added to target	790
number of target bytes added	71,062,180,514
number of target bytes removed	72,454,256,426
total size of change data capture files generated	824,642,530
number of rows updated by a matched clause	590,920
time taken to execute the entire operation	8.1 m
number of inserted rows	3,672,130
number of rows deleted by a matched clause	14,097,497
number of target partitions from which files were removed	0
time taken to rewrite unmodified rows in the matched files with LSM enabled, 0 otherwise	2.6 m
time taken to scan the files for matches	26.0 s
number of updated rows	590,920
number of target bytes before skipping	72,615,187,671
number of rows updated by a not matched by source clause	0
number of target partitions after skipping	0
number of change data capture files generated	9
number of source rows	31,315,973
number of files removed to target	889
number of rows deleted by a not matched by source clause	0
number of target files before skipping	900
time taken to rewrite the matched files (just the modified rows when LSM is enabled)	2.3 m
number of target partitions to which files were added	0

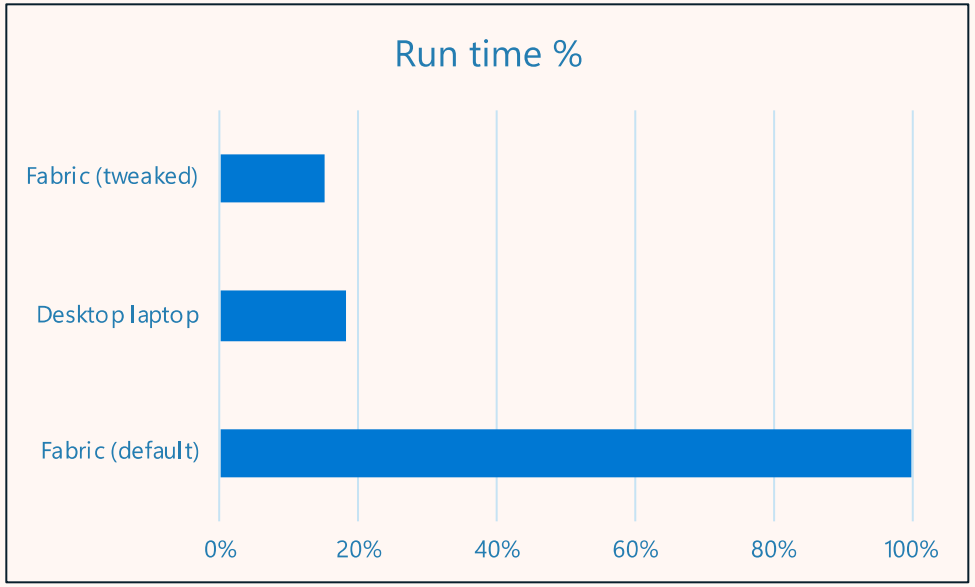
Performance Analysis and Benchmarking

Simulating Local Runs

- Observed unusual merge performance on small tables via Delta table commit stats using Spark.
- Established a benchmark: 50 iterations of merge, altering 5% of data on a 500MB table.
- Compare runs across diverse environments.
- The platform-specific settings and custom features caused performance degradation in this scenario. In this case – v-order and optimize write.
  - ✓ Small dataset with complex transformations: Performance boost by disabling optimized writes and V-order.
  - ✓ Large dataset with similar transformations: No significant gains from disabling optimized reads and writes.

Performance is job-dependent; fine-tuning may be required for each job.

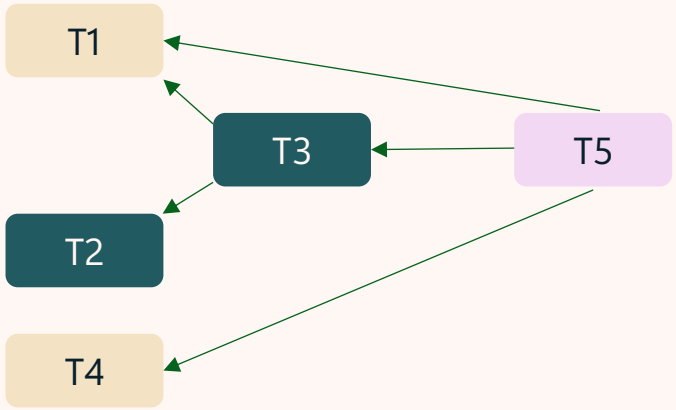
Run environment	Notes
Fabric (default)	All default settings. 3x 16vcpu
Desktop	All default settings. OSS Spark. 16vcpu
Fabric (tweaked)	Disabled v-order, disabled optimize write. 3x 16vcpu



Concurrency Factors

Dependency Management – DAGs

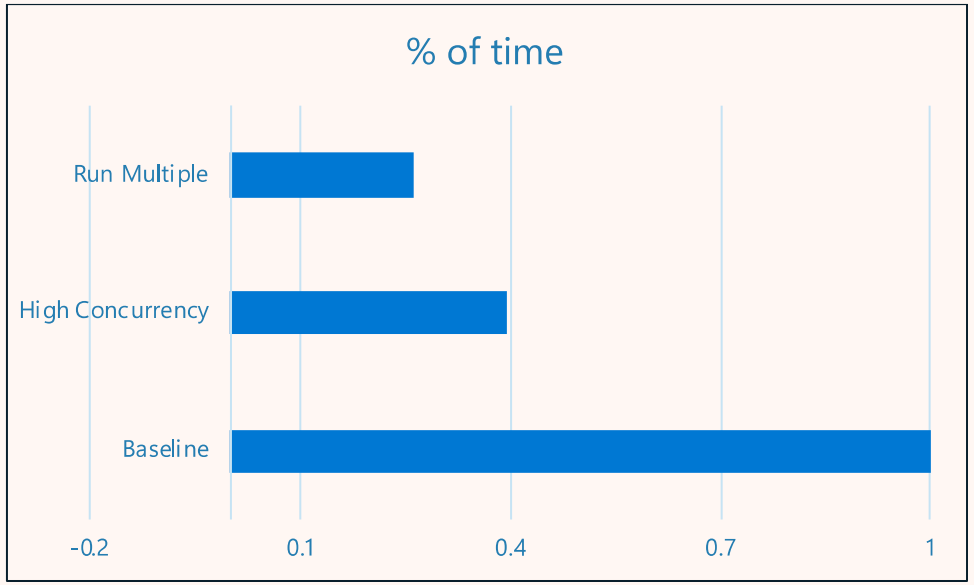
- Table transformations inherently create dependencies on other tables and datasets. Processing these without optimization can lead to underutilized capacity and longer pipeline processing times.



- Analyze dependencies to determine the optimal execution path for parallel pipeline runs.
- Utilize Fabric High Concurrency and NotebookUtils Run Multiple for orchestration, with Run Multiple being more effective for complex DAGs.

Statistics on Parallel Runs

- Optimal DAG critical path length – 4
- Total volume – 3.8TB
- Rows copied per update – 14,015,767,408
- Rows changed per update – 2,511,859



Summary

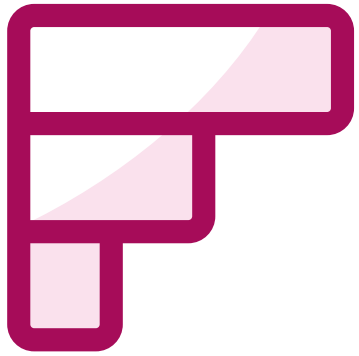
We formed a team to address key challenges and collaborated with Microsoft to ensure a production-ready solution.

- **Exploration (4-5 weeks):** Identified **short-term fixes, long-term enhancements, and pipeline design changes.**
  - **Implementation:** Short-term fixes deployed in **2 sprints**, running in **production for months.**

Key Enhancements

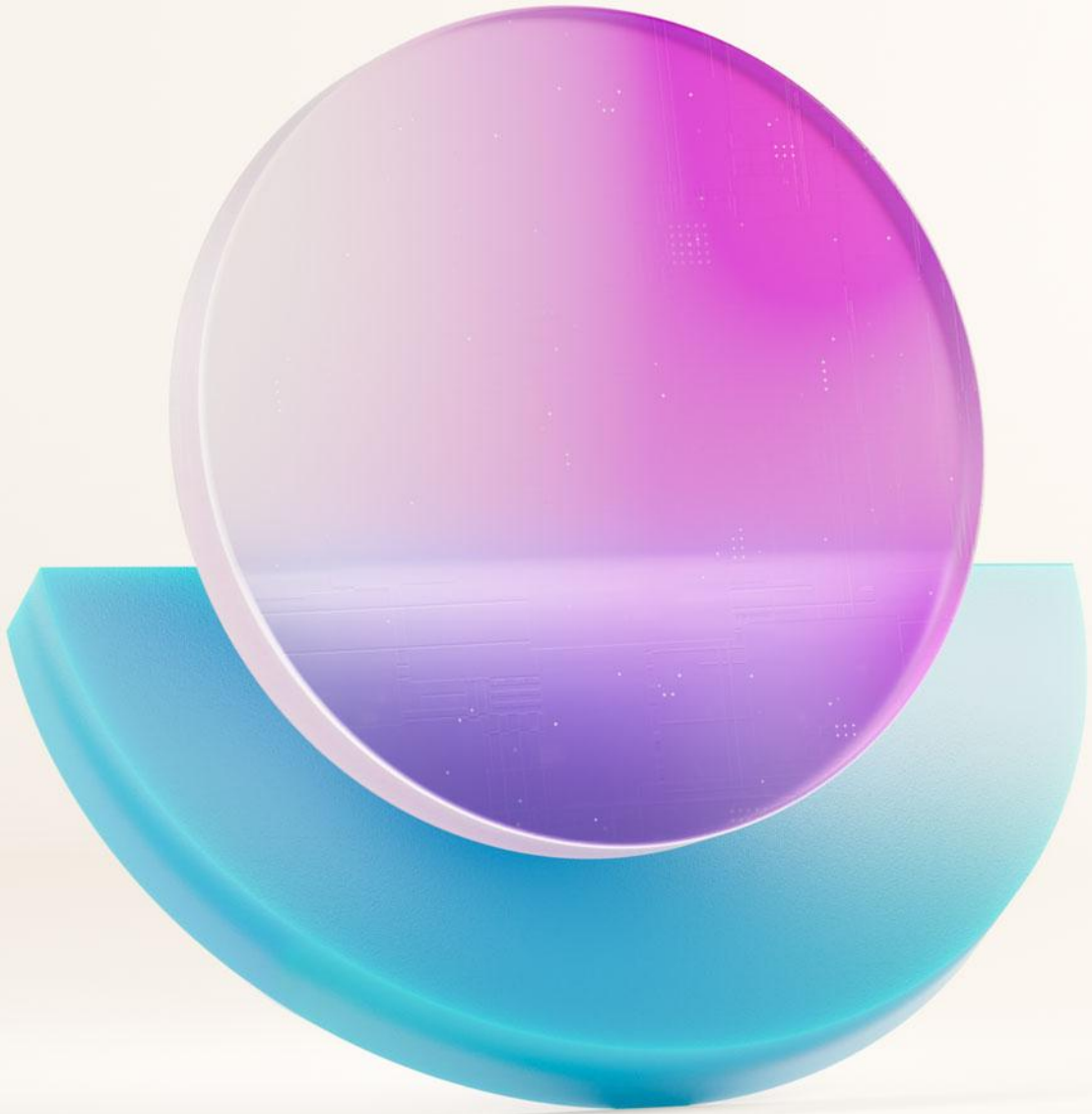
- ✓ **Feature & Config Optimization:** Disabled features, refined Spark configs, reduced complexity, partitioned time-series data.
  - ✓ **Data Modeling:** Ensured performance; easier in some areas.
  - ✓ **Capacity Management:** Addressed throttling gaps, admin rights, rigid settings, and burst capacity (autoscale/serverless billing testing in progress).
  - ✓ **Fabric Monitoring UI:** Working with Microsoft to resolve issues.
  - ✓ **Parallel Computation:** Evaluated **high concurrency & orchestration** for better performance.

This approach has **enhanced stability, scalability, and performance** in production.



Did the tools, techniques, and best practices shared today help you feel more confident in scaling your Spark workloads in Fabric?
(Let us know what resonated most — or what you'd like to learn more about!)

Microsoft Fabric
Community Conference



Fast-track your career in data and AI

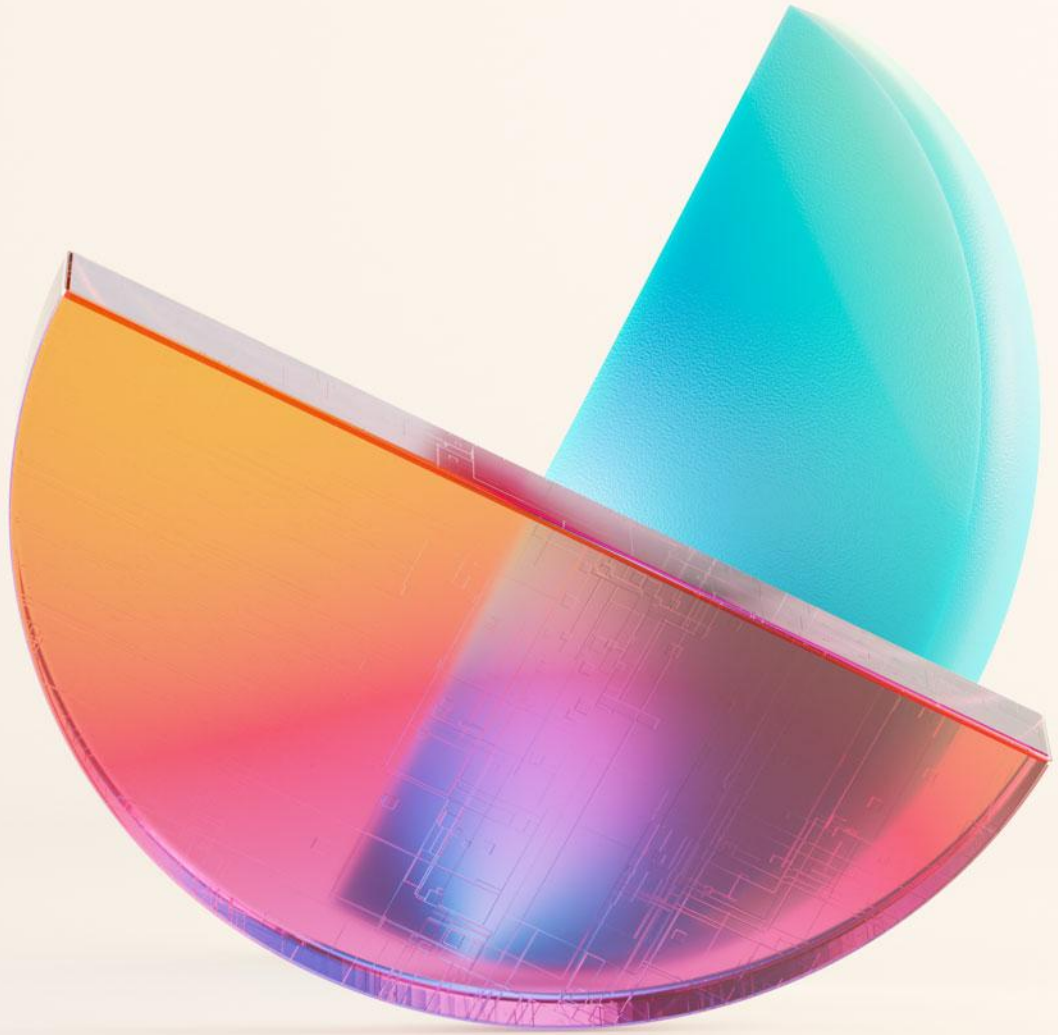
Become a Microsoft Certified Fabric Data Engineer Associate – take Exam DP-700 for free!

Visit the Fabric Community Lounge to learn more about this *limited-time offer*.

aka.ms/fabcon/dp700

Microsoft

Microsoft Fabric
Community Conference



Learn more about Microsoft Fabric

Power your AI transformation with a complete data platform

Microsoft

Get Involved in the Fabric Community

aka.ms/FabricCommunity
Connect with community members, ask questions, and learn more about Fabric

aka.ms/FabricUserGroups
Find a user group that matches your interests in your area or online

aka.ms/SuperUsers
Spread your Fabric knowledge, insights, and best practices with others

aka.ms/MVP
Technology experts that share their knowledge and passion with the community