Platform & Orchestration

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

MANAGE THE ANALYTICS LIFECYCLE

Configure Workspaces & access → **Develop** Git & environments → **Promote** Validate & deploy → **Orchestrate** Dependencies & triggers

01 Implement and manage an analytics solution 30–35%

Workspaces, capacity and governance

- A **workspace** groups Fabric items and access. A **capacity** supplies shared compute. Assign engineering workspaces to a Fabric-enabled capacity; a Power BI Pro license alone does not supply Fabric compute.
- **Tenant settings** control capabilities such as Git integration and service-principal API access. Enabling a capability does not grant workspace, repository or data permissions.
- **Domains** and subdomains organize data ownership and discovery. Fabric admins create top-level domains; domain admins manage their assigned domain. Domain assignment does not grant workspace access.
- **Promoted** signals content ready for reuse; **Certified** signals an authorized organizational review. **Master data** identifies core organizational entities. Endorsement support depends on item type; Power BI dashboards are not endorsable.
- **Sensitivity labels** classify information and support protection policies. Endorsement communicates trust, lineage shows dependencies, and permissions authorize access; these controls serve different purposes.

Workspace roles and data access

- **Admin** manages the workspace and access; **Member** supports collaboration and access delegation; **Contributor** creates and edits items; **Viewer** reads items. Use groups and the narrowest sufficient role.
- Use **item sharing** when a user needs one lakehouse or warehouse. A workspace role exposes the broader workspace; sharing a gold-layer item avoids granting access to bronze and silver items.
- Lakehouse **ReadData** authorizes SQL endpoint data reads. **ReadAll** authorizes underlying OneLake data access through engines such as Spark. **Build** is a semantic-model permission for creating analytical content.
- For granular SQL access, combine item access with explicit **GRANT** permissions on required objects or columns. Viewer access can coexist with additional SQL permissions; a broad editing role is unnecessary for SQL-only access.
- SQL endpoint security governs SQL queries. Evaluate **OneLake file access**, Spark permissions and shortcut identities separately. Separate sensitive data into another item or workspace when broad engineering write access would expose it.

Platform & Orchestration

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

MANAGE THE ANALYTICS LIFECYCLE

Configure Workspaces & access → **Develop** Git & environments → **Promote** Validate & deploy → **Orchestrate** Dependencies & triggers

Row, column and masking controls

- **Row-level security** uses an inline table-valued predicate function and a **SECURITY POLICY** with a filter predicate. Match the signed-in identity to authorized rows, including guest identities where applicable.
- **Column-level security** grants access only to permitted columns. Audit inherited grants as well as explicit permissions, and test with the intended Microsoft Entra user or group.
- **Dynamic data masking** alters query presentation without changing stored values. Configure masks with column DDL. Masking reduces accidental exposure; combine it with access controls for sensitive data.
- **default()** uses a type-specific placeholder, including 1900-01-01 for dates. **partial(prefix, padding, suffix)** preserves chosen edge characters; **email()** uses a fixed email pattern; **random()** masks numeric values.
- **UNMASK** permits clear-text results while SELECT permits querying. Elevated warehouse permissions and workspace editing roles can expose unmasked values; test security as the actual consumer.

Git and isolated development

- Connect a workspace to a **repository, branch and folder**. Azure Repos integration also identifies the organization and project. Repository access and Fabric workspace permissions are separate requirements.
- Use a **dedicated workspace and feature branch** for each independent feature or developer. Review a pull request before merging into main, then update the workspace linked to main.
- **Commit** sends supported workspace definitions to Git. **Update workspace** brings repository changes into Fabric. A merge in the Git provider does not automatically update the live workspace.
- Git tracks supported **item definitions and metadata**, not a backup of table contents or credentials. Select multiple supported changes in one commit; grant only the repository scopes required for integration.

Deployment and database projects

- **Deployment pipelines** promote supported item definitions through development, test and production workspaces. Compare stages and use **selective deployment** when only specific items should change.
- Existing **paired items** are updated in place. Names alone do not define every subsequent pairing decision; preserve established pairing instead of deleting and recreating target items.

Platform & Orchestration

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

MANAGE THE ANALYTICS LIFECYCLE

Configure Workspaces & access → **Develop** Git & environments → **Promote** Validate & deploy → **Orchestrate** Dependencies & triggers

- **Autobinding** can connect a deployed report to its paired semantic model in the target stage. If the required dependency is absent, include it through **Select related** or deploy it first.
- Provision target data, connections, credentials, permissions and refresh schedules deliberately. Deployment is not a data copy; validate item support, folder behavior and environment-specific settings before release.
- A **SQL database project** versions warehouse schema and validates supported object references during build. Use project diagnostics for unresolved references; test data loading and runtime behavior separately.

Spark environments and sessions

- A Fabric **environment** packages Spark runtime, libraries and compute configuration. Publish reusable Python packages in an environment and select it as the workspace default for new notebooks.
- **High concurrency** shares a Spark application among compatible notebooks within a user boundary, with isolated notebook execution contexts. Align workspace, default lakehouse, compute settings and libraries.
- Use **standard sessions** for separate Spark applications when workloads need compute isolation. Separate variables within a shared application do not imply separate cluster resources.
- **Autoscale** adjusts pool resources; **dynamic allocation** adjusts Spark executors. Configure sensible minima and maxima for workload demand. More executors do not remove skew or an inefficient query plan.
- Use **managed private endpoints** for supported Spark access to private Azure sources. Pipeline and dataflow connectors may instead need an appropriate gateway; choose connectivity for the executing engine.

Pipeline dependencies and triggers

- A **master pipeline** invokes child pipelines or mixed activities under one schedule. Run independent imports in parallel, then gate silver and gold processing on successful completion of their prerequisites.
- **On success** gates dependent work; **On failure** starts error handling; **On completion** runs after success or failure. Explicit dependencies determine order, not the position of activities on the canvas.
- Set **retry count and interval** on the activity that experiences transient failures. Fault tolerance can skip supported bad records; it does not replace retries for a failed connection.
- Use a **schedule** for clock-based execution and a supported **event trigger** for file arrival. Set the intended time zone. **Activator** evaluates event conditions and can initiate actions such as alerts or pipeline execution.
- Refresh a **semantic model** only after its data load succeeds. Add an email action to the failure path and include run context. Design failure handling so that a notification does not conceal a failed load.

Platform & Orchestration / Ingestion & Transformation

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

FROM SOURCE TO ANALYTICAL DATA

Source Batch & events → **Bronze** Preserve fidelity → **Silver** Clean & standardize → **Gold** Dimensions & facts

Parameters, notebook DAGs and identity

- **Lookup** reads a control table; **ForEach** iterates its output array. Use **@activity('Lookup').output.value** for multiple rows and **@item()** for the current element; limit concurrency to protect sources.
- **@pipeline().parameters.name** evaluates a parameter with its native type. **@{...}** interpolates into a string. Parameterize table names, paths and load boundaries instead of duplicating pipelines.
- **notebookutils.notebook.runMultiple** orchestrates notebook activities as a **DAG**. Define dependencies explicitly; concurrency limits parallel work but does not establish business ordering. Load prerequisite dimensions before dependent facts.
- Use an authorized **service principal** for unattended supported Fabric API calls. Retrieve Key Vault secrets with **notebookutils.credentials.getSecret(vaultUrl, secretName)** and never print them. A workspace identity and a private endpoint solve different needs.

02 Ingest and transform data 30–35%

Medallion layers and engine selection

- **Bronze** preserves source fidelity; **silver** cleans, deduplicates and standardizes; **gold** serves business-ready dimensional data. Complete required upstream loads before publishing downstream results.
- A **lakehouse** combines files and Delta tables for Spark-led engineering. Its **SQL analytics endpoint** reads supported tables through T-SQL; perform table data writes through a supported lakehouse write engine.
- A **warehouse** supports relational analytics and T-SQL data modification. An **eventhouse** hosts KQL databases for event, telemetry and time-series analysis. Choose the store for access patterns and write requirements.
- Use **Dataflow Gen2** for low-code Power Query transformations, **notebooks** for Python/Spark and custom processing, and **T-SQL** for relational set operations. Pipelines orchestrate work and move large datasets.
- **Delta Lake** combines Parquet files with a transaction log. A folder of Parquet files alone is not a Delta table. Keep unstructured objects in **Files** and discoverable lakehouse Delta tables in **Tables**.

Ingestion & Transformation

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

FROM SOURCE TO ANALYTICAL DATA

Source Batch & events → **Bronze** Preserve fidelity → **Silver** Clean & standardize → **Gold** Dimensions & facts

Shortcuts, mirroring and shared data

- **OneLake shortcuts** reference existing data without an ingestion copy. Tables shortcuts expose supported table formats; Files shortcuts expose arbitrary files. Deleting the shortcut leaves its target intact.
- For **S3 and GCS** shortcuts, workspace caching reduces repeat cross-cloud reads. Files larger than **1 GB** are not cached; expiry or a newer source version causes remote reads. ADLS shortcuts do not use this cache.
- Internal shortcuts normally authorize the **calling user** at the target. Engine identity matters: **delegated identity** can use the item owner's access; use supported **user identity** behavior where each consumer's source permissions must apply.
- A shortcut in a **KQL database** is queried with **external_table()**. It enables access to lake data without an extra pipeline copy. Use supported **Dataverse integration** to link application data into Fabric.
- **Mirroring** replicates supported operational data into OneLake for analytics. A **Direct Lake** semantic model reads OneLake data, reducing reporting queries against the source database; freshness still depends on replication and model state.
- Eventhouse **OneLake availability** exposes KQL data in Delta format. Configure its scope and existing-table backfill explicitly; batching affects availability latency, so do not equate it with immediate visibility.

Batch ingestion and load patterns

- **Copy data** moves large files or tables with minimal transformation and can land data in Delta format. Use a built-in REST connector plus parameters and ForEach for multiple entity endpoints.
- Use **staged copy** when required by the source-to-warehouse connector path, including supported Snowflake transfers. Check source, staging and sink permissions; tune parallelism after measuring throughput.
- An **on-premises data gateway** connects supported pipeline or dataflow connectors to private sources. Validate gateway availability, source credentials, firewall rules and DNS separately.
- A **full load** replaces the complete target; an **incremental load** processes changes. Persist a reliable watermark or CDC position only after success, and make reruns idempotent to avoid duplicates or gaps.
- In **Dataflow Gen2**, destination **Replace** supports a full reload; **Append** adds rows. For incremental refresh, distinguish retained history from the recent window that is reread, and preserve source query folding where possible.
- Semantic-model incremental refresh uses Date/Time **RangeStart** and **RangeEnd** filters plus a refresh policy. Use a half-open interval to avoid overlap, and refresh a wide enough window for late corrections.

Ingestion & Transformation

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

FROM SOURCE TO ANALYTICAL DATA

Source Batch & events → **Bronze** Preserve fidelity → **Silver** Clean & standardize → **Gold** Dimensions & facts

Spark cleansing and Delta updates

- Use DataFrame **select**, **filter**, **withColumn**, joins and casts to shape data. Normalize case, handle nulls explicitly and validate malformed values before promoting data to silver.
- **dropDuplicates(keys)** removes duplicate keys but does not choose the latest business record. For deterministic selection, rank by a reliable timestamp and tie-breaker, then retain one row per key.
- **saveAsTable** writes and registers a table; **save(path)** writes to a storage location. **append** adds rows; **overwrite** replaces the target. Choose Delta format for lakehouse tables.
- **Delta MERGE** matches source and target by key. Use **whenMatchedUpdate**, **whenNotMatchedInsert**, and supported **whenNotMatchedBySourceUpdate** logic with explicit conditions for upserts and deactivation.
- Deduplicate the merge source and preserve business change timestamps. Treat absence as deletion or inactivity only when the source is an authoritative full snapshot; a change-only feed omits unchanged records.

Star schemas and changing dimensions

- Define a fact table's **grain** before its measures and keys. Dimensions describe entities; facts record events. Flatten category hierarchies into a dimension when a simple star schema is required.
- A **business key** identifies an entity in its source; a **surrogate key** identifies a dimension row. Prefer compact integer relationship keys over wide hash strings when appropriate for analytical performance.
- **SCD Type 0** preserves the original value; **Type 1** overwrites; **Type 2** adds a version with effective dates and a new surrogate key; **Type 3** retains limited prior attributes in columns.
- For **historical attributes**, resolve the dimension version effective at the fact's event time. For current attributes, resolve the business entity to its current dimension row. Use non-overlapping effective-date ranges.
- A normalized **attribute hash** can detect changes across many columns. Compare source and current-target hashes using consistent null, type and string handling; hash collisions and inconsistent serialization require consideration.
- Warehouse **PRIMARY KEY** and **UNIQUE** constraints require **NONCLUSTERED NOT ENFORCED** and are added with **ALTER TABLE**. ETL must enforce uniqueness and referential integrity; load parent dimensions before dependent tables.

Ingestion & Transformation

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

FROM SOURCE TO ANALYTICAL DATA

Source Batch & events → **Bronze** Preserve fidelity → **Silver** Clean & standardize → **Gold** Dimensions & facts

Relational shaping and aggregation

- **INNER JOIN** keeps matching rows; **LEFT OUTER JOIN** preserves every left row. A filter on right-side columns in WHERE can remove unmatched rows; place conditions carefully when preserving them matters.
- **WHERE** filters rows before aggregation; **HAVING** filters groups afterward. **ROLLUP** adds hierarchical subtotals and a grand total; **GROUPING SETS** defines exactly which grouping levels to return.
- **CAST(value AS type)** and **CONVERT(type, value)** change types. **TRY_CAST** or **TRY_CONVERT** returns null for supported failed conversions. Formatting a date as text changes its type; format presentation separately when dates must remain typed.
- SQL **window functions** use **PARTITION BY**, **ORDER BY** and a frame. **ROWS** counts row positions; a value-based frame has different semantics. Use deterministic ordering for ranking and cumulative calculations.

Eventstreams and change capture

- **Eventstreams** ingest, transform and route event data. Use a supported connector for Event Hubs or another source; a **custom endpoint** accepts events sent by your own application.
- Apply **Filter** before the destination to discard unwanted events. **Manage fields** shapes columns; **Expand** flattens supported nested data; **Union** combines streams; **Aggregate** computes measures over windows.
- Route events to an **eventhouse** for interactive KQL analysis and to a **lakehouse** for downstream Delta processing. A batch Copy activity and a continuous eventstream address different ingestion patterns.
- For an **Azure SQL CDC** source, enable required database and table change capture and grant connector permissions. Check source connectivity and CDC prerequisites before debugging downstream transformations.

Streaming time, windows and state

- **Tumbling windows** are fixed and non-overlapping. **Hopping windows** overlap when the hop is shorter than the duration. **Sliding windows** follow event-time changes; **session windows** group activity separated by an inactivity gap.
- **Spark Structured Streaming** processes incremental DataFrames with readStream and writeStream. Persist **checkpoints** for offsets and state, and retain source events long enough to recover from expected outages.
- **Event time** describes when an event occurred; **processing time** describes when the engine handled it. A **watermark** bounds supported state and late-data handling; it does not guarantee every late event is retained.
- For per-user rolling totals, partition by user and define backward-looking time boundaries. Arbitrary batch **Window.rangeBetween** logic is not universally supported on streaming DataFrames; use supported stateful processing with durable history.

Ingestion & Transformation / Monitoring & Performance

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

FROM SOURCE TO ANALYTICAL DATA

Source Batch & events → **Bronze** Preserve fidelity → **Silver** Clean & standardize → **Gold** Dimensions & facts

KQL filtering and shaping

- KQL pipes tabular results through operators. **where** filters, **project** selects columns, **extend** calculates columns, and **sort by** establishes output order. Combine conditions with **and** and **or**.
- **summarize** groups values with functions such as count(), sum(), avg() and stdev(). **bin(timestamp, 1h)** creates hourly buckets; **arg_min(timestamp, *)** returns the earliest row within each group.
- Use **dynamic** for JSON-like structures and arrays. Extract nested fields and cast them explicitly; **mv-expand** creates rows from array elements and can increase the result size substantially.
- An **update policy** transforms newly ingested source rows into a target table. Its output must match the target schema. Additional dynamic properties differ from changes to declared column types; validate both mappings and conversions.

KQL row and partition calculations

- **prev()** compares a row with a preceding row in a serialized row set. Sort by entity and timestamp, then prevent comparisons from crossing entity boundaries; serialization alone does not establish chronological order.
- **row_cumsum()** supports running totals with a restart condition. **row_rank_dense()** assigns equal ranks to ties without gaps. Sort by partition keys and ranking values, and restart when the logical partition changes.
- **row_window_session()** identifies session starts in ordered data using elapsed-time limits and optional restarts. For fixed calendar-hour grouping, use **bin()**; fixed buckets and inactivity-based sessions are different requirements.

03 Monitor and optimize an analytics solution 30–35%

Monitoring and refresh telemetry

- **Monitoring hub** brings together supported item runs. Filter by item, status and time; **Location** identifies the workspace. Open the relevant run before drilling into activity or application details.
- **Workspace monitoring** collects supported logs and metrics into an Eventhouse-backed monitoring database after collection is enabled. Query it with KQL; it does not backfill activity from before collection began.

Monitoring & Performance

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

DIAGNOSE AND IMPROVE

Observe Runs & metrics → **Locate** Activity & query → **Diagnose** Logs & execution → **Tune** Data & compute

- For recent ingestion, use **where ingestion_time() >= ago(540s)** when ingestion timestamps are available. Source event time and job start or end time measure different things.
- Use **refresh history** for semantic-model outcomes and **semantic link** in notebooks for programmatic refresh and analysis. Correlate refresh status with upstream load runs; visualize history when trends matter.

Pipeline, dataflow and notebook errors

- In a failed **pipeline activity**, inspect **input JSON** for the resolved dynamic query and parameters. Inspect **output JSON** for results, counts and error details; the overall pipeline status alone lacks this context.
- For a failed **notebook**, open its execution logs and cell-level traceback. Use the run snapshot to understand the executed definition, and distinguish notebook code errors from capacity or connectivity failures.
- For **Dataflow Gen2**, inspect refresh details, failed queries and destination errors. Check credentials, source schema, type conversions and folding before increasing capacity or widening the refresh window.
- Use child pipelines to decompose workflows that exceed supported **activity limits**. Parallelism cannot fix a structural validation error. Record watermarks and row counts so recovery can restart safely.

Spark diagnostics and tuning

- The **Spark UI** exposes jobs, stages, tasks, executors, shuffle and memory metrics. Use the **Spark History Server** for completed applications and its timeline to compare job and stage execution.
- Find **data skew** by comparing task durations and partition sizes. Filter and select early, avoid unnecessary shuffles, and consider a **broadcast join** only when one input is small enough for executor memory.
- Inspect the application's **runtime and environment** to identify Spark and Delta library versions. A Delta table's transaction version is a commit number, not its Delta software version.
- Tune partitions and file sizes before adding compute. Reuse a compatible session where appropriate, cache reused data selectively, and release idle sessions to reduce wasted capacity.

Delta files and maintenance

- **OPTIMIZE** compacts small files into more efficient files. Many tiny Parquet files increase listing, scheduling and read overhead; compaction addresses this without changing the logical table rows.
- **VACUUM** removes data files no longer referenced by the Delta table and older than its retention threshold. It does not compact current files or delete old business rows based on their event date.

Monitoring & Performance

Implementing Data Engineering Solutions Using Microsoft Fabric

KNOWLEDGE CHEATSHEET

DIAGNOSE AND IMPROVE

Observe Runs & metrics → **Locate** Activity & query → **Diagnose** Logs & execution → **Tune** Data & compute

- Choose retention to preserve required **time travel**, concurrent readers and streaming recovery. Schedule maintenance through a notebook or pipeline; do not manually delete files that belong to an active Delta table.
- **V-Order** optimizes Parquet layout for analytical reads at additional write cost. Evaluate it for curated read-heavy data; write-intensive staging can favor less write optimization. Compaction, layout and retention are separate controls.

Warehouse query diagnosis

- Use **sys.dm_exec_sessions** for connected session identities, **sys.dm_exec_requests** for current requests, and **sys.dm_exec_connections** for connection details. Join supported views when the investigation needs both user and request context.
- Use **Query activity** and **queryinsights** history for completed or failed queries. Filter by time, status, user and statement; group repeated failures to distinguish isolated errors from a recurring workload problem.
- Maintain useful **statistics** and inspect scans, joins and data movement. Compact keys, suitable types and a clear star schema reduce analytical work; unenforced primary keys do not validate dirty data.
- For **Direct Lake**, assess model memory, table size, file layout and capacity guardrails when visuals slow down or fail. Optimize the model and query workload before assuming that a larger capacity is the only remedy.

Streaming health and capacity

- For **eventstreams**, use **Runtime logs** for timestamped error details and **Data insights** for throughput and error metrics. Check source, transform and destination nodes to locate schema or processing failures.
- Optimize **KQL** by filtering early and projecting only needed columns before expensive joins. Choose join semantics for correctness; use supported broadcast or shuffle strategies when data sizes justify them.
- In an **eventhouse**, retention controls data lifetime while hot-cache policy favors frequently queried history. Monitor ingestion lag and failures. Use the **Capacity Metrics app** to distinguish workload inefficiency from contention or throttling.