CHEAT SHEET MICROSOFT FABRIC / MONITORING & OBSERVABILITY / INTERMEDIATE

Monitor Fabric Pipelines and Notebooks

Monitor Microsoft Fabric pipelines and notebooks using Monitoring hub, Workspace Monitoring, Spark monitoring, KQL, alerts, and capacity telemetry so failures and performance regressions become observable.



Updated Oct 4, 2026 Online version →

PROBLEM

Fabric pipelines and notebooks can succeed, fail, slow down, retry, queue, or consume more capacity than expected.

Opening individual run pages is enough for one incident.

It is not enough for production operations.

You need to answer questions such as:

- Which pipelines failed today?
- Which activity failed first?
- Which notebooks are becoming slower over time?
- Is the slowdown Spark execution or pipeline orchestration?
- Are failures concentrated in one notebook or activity type?
- Which runs were scheduled vs on-demand?
- Are jobs queueing because of capacity pressure?
- Did a deployment change runtime behavior?
- Should the team receive an alert automatically?

The goal is to combine **operational monitoring**, **historical log analysis**, **Spark diagnostics**, and **capacity telemetry** without building a custom monitoring platform from scratch.

QUICK ANSWER

Use the monitoring layer that matches the question:

```
Need to inspect a current/recent run quickly?
→ Monitoring hub

Need pipeline activity details?
→ Pipeline run detail / activity run history

Need notebook Spark stages, tasks, executors, and logs?
→ Spark application monitoring / Spark UI

Need historical workspace-wide analysis?
→ Workspace Monitoring + Eventhouse + KQL

Need alerts?
→ Monitoring alerts / Activator / supported alerting workflow
```

```
Need capacity-consumption context?
→ Fabric Capacity Metrics app
```

For workspace-wide pipeline status:

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| summarize Runs = count() by JobStatus
| order by Runs desc
```

For failed pipeline activities:

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Failed"
| project
    Timestamp,
    ItemName,
    PipelineRunId,
    ActivityName,
    ActivityType,
    FailureType,
    DurationMs
| order by Timestamp desc
```

For notebook job history:

```kql
ItemJobEventLogs
| where ItemKind == "Notebook"
| where JobType in (
    "RunNotebook",
    "RunNotebookInteractive",
    "PipelineRunNotebook"
)
| project
    Timestamp,
    ItemName,
    JobType,
    JobStatus,
    DurationMs,
    JobStartTime,
    JobEndTime,
    JobInstanceId
| order by Timestamp desc
```

Use the UI for fast incident inspection.

Use KQL when you need trends, comparisons, automation, and recurring operational visibility.

Code / commands

1. Use Monitoring hub for fast operational triage

Open:

```
Fabric
→ Monitor
```

Use the Monitoring hub when you need to:

- find recent pipeline runs
- find notebook/Spark runs
- filter by status
- filter by item type
- inspect run duration
- open run details
- identify upstream/downstream execution relationships
- retry supported pipeline runs

This is usually the fastest place to begin during an incident.

Do not build a KQL query first if one failed run is already visible in the portal.

---

2. Open pipeline run details to find the failed activity

A pipeline can contain many activities, but only one may be the original failure.

Use:

```
Monitoring hub
→ Pipeline run
→ Run details
→ Activity runs
```

Capture:

```
pipeline name
run ID
activity name
activity type
status
start time
end time
duration
failure type
```

Follow the earliest real failure rather than downstream skipped activities.

---

3. Use notebook contextual monitoring during development

Inside a Fabric notebook, contextual Spark monitoring can show execution progress at notebook/cell level.

Use it to inspect:

- Spark jobs triggered by a cell
- stages
- tasks
- executors
- logs
- execution errors
- Spark advisor recommendations

This is useful when:

```
the notebook is slow
```

but you still need to determine whether the slow cell is caused by:

- scan
- shuffle
- skew
- executor failure
- memory pressure
- Spark configuration

4. Use Spark application monitoring for notebook performance

Open the Spark application detail page from:

```
Monitoring hub
```

or:

```
Notebook
→ Recent runs
```

Important areas include:

```
Jobs
Resources
Logs
Diagnostics
```

The Jobs view can expose:

- job duration
- stages
- tasks
- data read
- data written
- code snippet

The Resources view helps correlate the Spark job with executor usage.

The Logs view provides:

- driver logs
- Livy logs
- prelaunch logs

Use the Spark UI when you need lower-level stage/task detail.

5. Use Spark UI for stage-level diagnosis

The Spark UI is the right tool when the question becomes:

```
Why did this Spark application take 25 minutes?
```

Inspect:

```
Jobs
Stages
Storage
Environment
Executors
```

Examples:

```
One task much slower than peers
→ possible skew

Large shuffle read/write
→ join / aggregation redistribution

Executor failures
→ memory/resource/runtime issue

Large cached datasets
→ storage/cache pressure

Unexpected Spark config
→ Environment/configuration drift
```

Do not rely only on notebook wall-clock duration.

6. Enable Workspace Monitoring for historical analysis

Workspace Monitoring creates a monitoring Eventhouse and read-only monitoring KQL database for supported Fabric items.

Enable from:

```
Workspace settings
→ Monitoring
→ Add / enable monitoring Eventhouse
```

Use Workspace Monitoring when you need:

- historical run trends
- workspace-wide failures
- recurring performance analysis
- custom KQL
- dashboards
- alerting logic
- operational reporting

This is the key shift from:

```
inspect one run
```

to:

```
operate the platform
```

7. Understand the two main pipeline monitoring tables

Pipeline-level monitoring:

```
ItemJobEventLogs
```

Activity-level pipeline monitoring:

```
FabricDataPipelinesActivityRunsLogs
```

Use:

```
ItemJobEventLogs
```

for questions such as:

- how many pipelines failed?
- which pipeline is getting slower?
- which jobs are scheduled vs on-demand?
- who triggered the run?

Use:

```
FabricDataPipelinesActivityRunsLogs
```

for questions such as:

- which activity failed?
- which activity type is slow?
- where does pipeline time accumulate?
- how much CPU did an activity consume where available?

---

8. Inspect the monitoring schema before writing production KQL

Start with:

```kql
ItemJobEventLogs
| getschema
```

and:

```kql
FabricDataPipelinesActivityRunsLogs
| getschema
```

Use this because monitoring schemas can evolve.

Do not assume code copied from an old article or another tenant matches the current table exactly.

---

9. Count pipeline runs by status

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| summarize Runs = count() by JobStatus
| order by Runs desc
```

This gives a fast health summary.

Be careful when the log model includes multiple lifecycle events for the same job instance.

For completed-run reporting, filter to terminal statuses.

10. Count only terminal pipeline outcomes

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where JobStatus in ("Completed", "Failed")
| summarize Runs = count() by JobStatus
```

Use terminal states for success/failure reporting.

Do not count:

```
In progress
Not started
```

as completed executions.

11. Show recent failed pipelines

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where JobStatus == "Failed"
| project
    Timestamp,
    WorkspaceName,
    ItemName,
    JobInstanceId,
    JobInvokeType,
    JobStartTime,
    JobEndTime,
    DurationMs,
    ExecutingPrincipalType
| order by Timestamp desc
```

Use the `JobInstanceId` to correlate a specific run with the portal or downstream run-level investigation.

12. Show recent failed pipeline activities

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Failed"
| project
    Timestamp,
    ItemName,
    PipelineRunId,
    ActivityName,
    ActivityType,
    OperationStartTime,
    OperationEndTime,
    DurationMs,
    FailureType
| order by Timestamp desc
```

This is much more actionable than a pipeline-level failure count when the same pipeline contains many activities.

13. Find the most frequently failing activities

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Failed"
| summarize
    Failures = count()
    by ItemName, ActivityName, ActivityType
| order by Failures desc
```

Use this to prioritize structural reliability problems.

A repeatedly failing activity deserves more attention than a single isolated transient failure.

14. Summarize pipeline activity duration

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Succeeded"
| summarize
    Runs = count(),
    AvgDurationMs = avg(DurationMs),
    P95DurationMs = percentile(DurationMs, 95),
    MaxDurationMs = max(DurationMs)
    by ItemName, ActivityName, ActivityType
| order by P95DurationMs desc
```

Use percentile-based analysis for performance trends.

Average alone can hide intermittent slow runs.

Do not interpret P95 without enough representative history.

15. Find pipeline activity duration regressions

A simple recent-vs-previous comparison:

```kql
let RecentStart = ago(1d);
let BaselineStart = ago(8d);

let Recent =
    FabricDataPipelinesActivityRunsLogs
    | where Status == "Succeeded"
    | where Timestamp >= RecentStart
    | summarize
        RecentAvgMs = avg(DurationMs)
        by ItemName, ActivityName;

let Baseline =
    FabricDataPipelinesActivityRunsLogs
    | where Status == "Succeeded"
    | where Timestamp between (BaselineStart .. RecentStart)
    | summarize
        BaselineAvgMs = avg(DurationMs)
        by ItemName, ActivityName;

Recent
| join kind=inner Baseline
    on ItemName, ActivityName
| extend
```

```
    DurationRatio = RecentAvgMs / BaselineAvgMs
| order by DurationRatio desc
```

This is a triage comparison, not an automatic performance verdict.

Check:

- input volume
- source latency
- code changes
- capacity pressure

before calling a run a regression.

16. Find the slowest recent pipeline activities

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Succeeded"
| project
    Timestamp,
    ItemName,
    PipelineRunId,
    ActivityName,
    ActivityType,
    DurationMs
| order by DurationMs desc
| take 50
```

Use this for quick outlier discovery.

Then compare each activity with its own normal baseline.

17. Find pipeline activity failures by type

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Failed"
| summarize
    Failures = count()
    by FailureType, ActivityType
| order by Failures desc
```

Use this to distinguish patterns such as:

```
Notebook failures
Copy failures
Gateway-related failures
Other recurring failure categories
```

Do not treat `FailureType` as a full root-cause message.

Current workspace activity logs may not contain every detailed error string available in the run UI.

18. Show pipeline activity CPU usage where available

```kql
FabricDataPipelinesActivityRunsLogs
| where Status == "Succeeded"
| summarize
    TotalCpuCoreMs = sum(CpuCoreMs),
    AvgCpuCoreMs = avg(CpuCoreMs)
    by ItemName, ActivityName, ActivityType
| order by TotalCpuCoreMs desc
```

Use this as one operational signal.

Do not compare CPU metrics across fundamentally different activity types without context.

---

19. Find notebook runs from workspace monitoring

`ItemJobEventLogs` includes supported Notebook job types.

Example:

```kql
ItemJobEventLogs
| where ItemKind == "Notebook"
| where JobType in (
    "RunNotebook",
    "RunNotebookInteractive",
    "PipelineRunNotebook"
)
| project
    Timestamp,
    WorkspaceName,
    ItemName,
    JobType,
    JobStatus,
    JobInvokeType,
    JobStartTime,
    JobEndTime,
    DurationMs,
    JobInstanceId,
    ExecutingPrincipalType
| order by Timestamp desc
```

Use this for notebook-level historical visibility.

For Spark stage/task detail, move to Spark monitoring rather than expecting workspace job logs to contain low-level Spark metrics.

---

20. Find failed notebook runs

```kql
ItemJobEventLogs
| where ItemKind == "Notebook"
| where JobType in (
    "RunNotebook",
    "RunNotebookInteractive",
    "PipelineRunNotebook"
)
| where JobStatus == "Failed"
| project
    Timestamp,
    ItemName,
```

```
    JobType,
    JobInstanceId,
    JobStartTime,
    JobEndTime,
    DurationMs
| order by Timestamp desc
```

Use this to identify which notebook items fail repeatedly.

Then open the corresponding Spark application for actual Spark diagnostics.

21. Track notebook duration trends

```kql
ItemJobEventLogs
| where ItemKind == "Notebook"
| where JobStatus == "Completed"
| where JobType in (
    "RunNotebook",
    "PipelineRunNotebook"
)
| summarize
    Runs = count(),
    AvgDurationMs = avg(DurationMs),
    P95DurationMs = percentile(DurationMs, 95),
    MaxDurationMs = max(DurationMs)
    by ItemName
| order by P95DurationMs desc
```

This is useful for recurring notebook workloads.

Separate interactive development runs when they are not comparable to scheduled production runs.

22. Split notebook runs by invocation type

```kql
ItemJobEventLogs
| where ItemKind == "Notebook"
| where JobStatus == "Completed"
| summarize
    Runs = count(),
    AvgDurationMs = avg(DurationMs)
    by ItemName, JobType, JobInvokeType
| order by ItemName asc
```

This helps avoid mixing:

```
manual interactive development
```

with:

```
scheduled production execution
```

in one performance baseline.

23. Track scheduled vs on-demand pipeline behavior

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where JobStatus in ("Completed", "Failed")
| summarize
    Runs = count(),
    Failures = countif(JobStatus == "Failed"),
    AvgDurationMs = avg(DurationMs)
    by ItemName, JobInvokeType
| order by ItemName asc
```

This can reveal that a pipeline behaves differently when:

- scheduled
- run manually
- triggered under a different load window

24. Monitor one specific pipeline

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where ItemName == "Daily Sales"
| where JobStatus in ("Completed", "Failed")
| project
    Timestamp,
    JobStatus,
    JobInvokeType,
    JobStartTime,
    JobEndTime,
    DurationMs,
    JobInstanceId
| order by Timestamp desc
```

Use pipeline-specific queries when troubleshooting a known workload.

25. Monitor one pipeline's activities

```kql
FabricDataPipelinesActivityRunsLogs
| where ItemName == "Daily Sales"
| project
    Timestamp,
    PipelineRunId,
    ActivityName,
    ActivityType,
    Status,
    DurationMs,
    FailureType
| order by Timestamp desc
```

Filter further by `PipelineRunId` to reconstruct one run.

26. Reconstruct one pipeline run

```kql
let RunId = "<pipeline-run-id>";
```

```
FabricDataPipelinesActivityRunsLogs
| where PipelineRunId == RunId
| project
    ActivityName,
    ActivityType,
    Status,
    OperationStartTime,
    OperationEndTime,
    DurationMs,
    FailureType
| order by OperationStartTime asc
```

This gives a chronological view of activity execution.

Useful for identifying:

- where time accumulated
- which activity failed first
- whether activities overlapped

---

27. Identify long-running jobs using a workload-specific threshold

Example:

KQL

```
let LongRunMs = 60 * 60 * 1000;

ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where JobStatus == "Completed"
| where DurationMs > LongRunMs
| project
    Timestamp,
    ItemName,
    DurationMinutes = round(DurationMs / 60000.0, 1),
    JobStartTime,
    JobEndTime,
    JobInstanceId
| order by DurationMs desc
```

The 60-minute threshold here is only an example.

Define thresholds from:

- SLA
- expected schedule
- historical baseline
- business window

Do not invent one universal “slow pipeline” threshold.

---

28. Detect pipeline failures in a recent alert window

```kql
ItemJobEventLogs
| where ItemKind == "Data Pipeline"
| where JobStatus == "Failed"
| where Timestamp >= ago(10m)
| project
    Timestamp,
    WorkspaceName,
    ItemName,
    JobInstanceId,
    JobStartTime,
    JobEndTime
| order by Timestamp desc
```

This kind of query can feed an alerting workflow.

Keep the alert polling interval and query time window aligned so failures are neither missed nor repeatedly alerted unnecessarily.

---

29. Create alerts only for actionable conditions

Good alert examples:

```
production pipeline failed
critical notebook failed
pipeline exceeded its SLA
repeated failures within a short period
scheduled run did not complete in expected window
```

Poor alert examples:

```
every notebook started
every retry
every warning
every transient event that self-recovers
```

Alert fatigue reduces operational value.

---

30. Use job-instance IDs for correlation

`JobInstanceId` identifies a specific Fabric item job instance.

Use it to correlate:

```
monitoring log
portal run
alert
incident
execution log
```

For pipelines, the activity-level `PipelineRunId` is the important run correlation field inside:

```
FabricDataPipelinesActivityRunsLogs
```

Do not correlate production incidents only by item name and timestamp.

---

31. Add your own business execution ID

Fabric job IDs are operational identifiers.

Your pipeline may also need a business-level ID:

```
batch_id
business_date
source_file_id
logical_run_id
```

Example in a notebook:

```python
from pyspark.sql import functions as F

output = (
    output
    .withColumn(
        "pipeline_run_id",
        F.lit(pipeline_run_id),
    )
    .withColumn(
        "batch_id",
        F.lit(batch_id),
    )
    .withColumn(
        "ingested_at_utc",
        F.current_timestamp(),
    )
)
```

This lets you correlate platform execution with actual data written.

---

32. Persist operational metrics from notebooks

Useful metrics include:

```
rows_read
rows_written
rows_rejected
source_date_min
source_date_max
duplicate_key_count
null_key_count
target_table
batch_id
pipeline_run_id
notebook_name
status
```

Example:

```python
metrics = {
    "pipeline_run_id": pipeline_run_id,
    "batch_id": batch_id,
    "rows_written": output.count(),
    "target_table": target_table,
}
```

For production, write structured metrics rather than relying only on printed dictionaries.

Be deliberate about expensive actions such as repeated `count()` calls.

33. Log pipeline execution state in an append-friendly way

A simple logical table:

```
pipeline_execution_log
----------------------
pipeline_name
pipeline_run_id
activity_name
batch_id
status
started_at
completed_at
rows_read
rows_written
error_type
```

Prefer append-only event records where practical.

This reduces contention compared with many concurrent notebooks updating the same log row repeatedly.

34. Avoid making the monitoring table a bottleneck

If many notebooks write to one Delta log table concurrently, that table can become a hot write target.

Consider:

```
append-only event logging
partitioned operational events
one row per activity transition
centralized reconciliation
```

Do not create a monitoring solution that introduces more pipeline failures than it observes.

35. Use Fabric run-series monitoring for recurring Spark workloads

Fabric can group recurring Spark applications into run series.

Use run-series views to compare:

- duration trend
- data input/output trend
- anomalous runs

This is useful for recurring:

- notebooks
- Spark job definitions
- notebook activities launched by pipelines

Use this before building a custom trend view when the built-in comparison already answers the question.

36. Compare Spark application runs when a notebook regresses

Fabric Spark application comparison can compare multiple runs and highlight performance differences.

Use it when:

```
Notebook took 8 minutes last week
Notebook now takes 25 minutes
```

Compare:

- duration
- input/output
- Spark execution behavior
- anomalous stages

Then inspect code/config/data changes.

---

37. Use Spark monitoring APIs for automation when UI is not enough

Fabric exposes Spark monitoring APIs for programmatic retrieval of Spark application information.

Use APIs when you need to:

- collect notebook Spark run metadata automatically
- integrate monitoring into another operational system
- build a custom internal dashboard
- correlate Spark application IDs with external incident systems

Prefer built-in Monitoring hub and Workspace Monitoring first unless API automation adds real value.

---

38. Use Capacity Metrics when the problem may be capacity pressure

Pipeline or notebook slowness can come from:

```
capacity saturation
concurrent workloads
resource contention
throttling
```

Use the Microsoft Fabric Capacity Metrics app to inspect capacity-level consumption.

This is especially important when:

```
many unrelated workloads become slow at the same time
```

That pattern is different from one inefficient notebook.

---

39. Correlate platform capacity with workload timing

When a pipeline becomes slow at 09:00 every day, compare:

```
pipeline duration
notebook Spark duration
capacity utilization
other jobs running at the same time
```

Do not optimize notebook code in isolation if the actual bottleneck is shared-capacity pressure.

40. Track a small set of operational service metrics

A useful production baseline is:

```
Reliability
- total runs
- success rate
- failure count
- repeated failure count

Performance
- average duration
- P95 duration
- max duration
- queue/startup time where available

Data quality
- rows read/written
- rejected rows
- duplicate keys
- null required keys

Resource
- Spark stage/runtime metrics
- capacity utilization
- activity CPU where available
```

Do not collect metrics solely because they are available.

Each metric should support an operational decision.

What the result means

Monitoring surface	Best use
Monitoring hub	Fast current/recent operational view
Pipeline run detail	Activity-level pipeline troubleshooting
Notebook contextual monitoring	Cell/Spark execution while developing
Spark application detail	Job, resource, log, and diagnostic analysis
Spark UI	Deep stage/task/executor troubleshooting
Workspace Monitoring	Historical workspace-wide telemetry
`ItemJobEventLogs`	Item/job-level Fabric execution history
`FabricDataPipelinesActivityRunsLogs`	Pipeline activity-level history
Run series / run comparison	Recurring Spark performance trends
Capacity Metrics	Capacity pressure and workload consumption
Alerts / Activator	Notify when an operational condition occurs

The key distinction is:

```
Pipeline status
!=
activity root cause
!=
Spark root cause
!=
capacity root cause
```

Move down the monitoring stack only as far as the incident requires.

What to look for

Failure rate increases across many unrelated pipelines

Investigate:

- shared source dependency
- credentials
- gateway
- workspace/capacity issue
- platform/service incident

Do not debug every pipeline independently first.

One activity fails repeatedly

This is likely a structural problem.

Inspect:

- activity parameters
- data contract
- retry behavior
- identity
- source/sink dependency

Pipeline duration rises but Spark duration does not

Look at:

- queueing
- orchestration gaps
- external activities
- session startup
- source/sink waits

The notebook may not be the bottleneck.

Notebook duration rises with higher input volume

This may be expected scaling behavior.

Normalize comparisons against:

- input rows
- bytes processed
- file count
- partition count

before calling it a regression.

One Spark task dominates a stage

Investigate skew.

Workspace-level duration alone cannot diagnose this.

Several workloads slow down simultaneously

Check capacity pressure before rewriting individual code.

Failures happen only in scheduled runs

Compare:

- identity
- execution time window
- source availability
- schedule overlap
- invocation type
- capacity load

Scheduled and manual execution are not always equivalent operationally.

> **PRODUCTION NOTES**
>
> ### Workspace Monitoring is a production observability feature, but schema awareness matters
>
> The monitoring Eventhouse gives you durable KQL-queryable logs.
>
> Always inspect the current schema before standardizing dashboards or alert logic.
>
> Monitoring tables and coverage can evolve.
>
> ### Workspace Monitoring has capacity cost
>
> The monitoring Eventhouse and related monitoring components consume Fabric capacity.
>
> Observability is not free.
>
> Keep dashboards, queries, and retention aligned with actual operational value.
>
> ### Workspace Monitoring is read-only for collected logs
>
> Use the monitoring database for analysis.
>
> Do not design it as your application's mutable control database.
>
> ### Item job logs and activity logs have different granularity
>
> `ItemJobEventLogs` is the broad job-level view.
>
> `FabricDataPipelinesActivityRunsLogs` is the deeper pipeline activity view.
>
> Use the lowest granularity needed for the question.

Detailed error messages may still require the run UI or Spark logs

Workspace monitoring activity logs provide useful status/failure metadata, but they do not replace detailed activity output or Spark driver/executor logs for every root-cause investigation.

Notebook monitoring needs two layers

Use:

```
ItemJobEventLogs
```

for notebook run history.

Use:

```
Spark monitoring / Spark UI
```

for distributed execution details.

Trying to force stage/task analysis out of job-level workspace logs is the wrong layer.

Monitoring queries should filter terminal states correctly

Job lifecycle logs can include in-progress and terminal events.

For SLA/failure reporting, filter to terminal states or deduplicate by job instance where necessary.

Performance thresholds should be workload-specific

A 30-minute pipeline may be:

```
healthy for monthly processing
```

and:

```
unacceptable for a 5-minute SLA
```

Use historical baselines and service requirements.

Alerts need deduplication logic

A query/window design can generate repeated alerts for the same failure.

Include:

- run ID
- recent time window
- alert state
- polling interval

in the design.

COMMON MISTAKES

1. Monitoring only failures

Slow successful runs can be early warnings.

Track duration trends as well as status.

2. Using only pipeline-level status

A failed pipeline tells you very little about whether the root cause is:

- Copy
- Notebook
- SQL
- external API
- Delta write

Drill into activity level.

3. Using activity logs to diagnose Spark skew

That is the wrong telemetry layer.

Open Spark monitoring / Spark UI.

4. Treating average duration as enough

Averages hide intermittent regressions.

Track percentile and max behavior where useful.

5. Mixing manual and scheduled runs in one baseline

Different invocation patterns may process different volumes or run under different conditions.

Segment them when needed.

6. Hard-coding alert thresholds without workload history

Use SLA or historical behavior.

Do not copy a random “60-minute” threshold into every pipeline.

7. Logging every possible metric

Too much telemetry makes incidents harder to interpret.

Choose operationally meaningful metrics.

8. Building custom monitoring before using Fabric-native views

Monitoring hub, Spark monitoring, run series, Workspace Monitoring, and Capacity Metrics already cover much of the operational stack.

Extend only where required.

9. Ignoring business data metrics

A green pipeline can still write:

```
0 rows
duplicated rows
invalid keys
wrong date range
```

Monitor data correctness as well as technical execution.

10. Making the execution log table a concurrency hotspot

Observability writes need production design too.

Prefer append-oriented logging or a suitable centralized pattern.

NEXT STEP

Use the monitoring signal to choose the next investigation.

- **Pipeline failed** → open the activity run and identify the first failed activity.
- **Notebook activity failed** → open Spark application diagnostics and logs.
- **Pipeline is slower than normal** → compare activity duration before blaming Spark.
- **Notebook is slower than normal** → inspect run series, Spark stages, shuffle, skew, and input volume.
- **Many pipelines fail together** → investigate shared dependencies and capacity/platform state.
- **Capacity is saturated** → move to Capacity Metrics and workload scheduling/concurrency analysis.
- **Failure repeats** → create workspace-level KQL analysis and an actionable alert.
- **Pipeline succeeds but output is wrong** → add business-level execution and data-quality metrics.
- **Manual checks are becoming repetitive** → build a saved KQL Queryset/dashboard and alert only on conditions that require action.

Related deep dives

- Related cheat sheet: Debug Fabric Pipelines
- Related cheat sheet: Fabric Notebook Performance
- Related cheat sheet: Fabric Session Tags
- Related cheat sheet: Delta Lake Read and Write Patterns
- Related cheat sheet: Lakehouse vs Warehouse
- Related cheat sheet: Fabric Medallion Architecture
- Related cheat sheet: PySpark Performance Quick Reference
- Related cheat sheet: Read and Write Delta Tables
- Related cheat sheet: Data Quality Checks
- Related article: Monitoring Microsoft Fabric Pipelines and Notebooks