My structured approach to diagnosing historical failures and safely executing parameterized pipelines in Azure Data Factory

Dmitrii Fediuk: https://upwork.com/fl/mage2pro

1.

To diagnose past failures, Azure Data Factory provides the «**Rerun**» functionality on the «**Monitor**» tab.
This functionality allows to rerun the pipeline using the exact historical parameters and configuration (Snapshot) that were used during the failed run.
This is often the **highest-priority method** for verifying **transient issues** (e.g., network timeouts) or **reproducing** the failure.
I describe this step, along with log analysis, in **Section 4** below.
However, if it is necessary to run the pipeline with new parameters (e.g., for testing fixes or processing new data), then it is necessary to understand its logic and input data requirements.
I describe this in **Sections 7-10** below.

2.

Your likely oversight is narrowing the scope of the failure investigation to the problematic pipeline.
In fact, a significant portion of pipeline failures is caused by **external factors**: infrastructure and security, especially in legacy environments.
In particular:

- **Authentication and Authorization**: expired credentials or keys in Linked Services, incorrect configuration of Managed Identity, lack of access to Azure Key Vault or target data stores.
- **Network issues**: configuration of firewalls (Azure Storage, SQL DB), VNet, Private Endpoints, blocking access.
- **Integration Runtime** (**IR**): issues with availability, configuration, or performance of Self-Hosted IR.

Ignoring external factors is a critical mistake when diagnosing failures in Azure Data Factory.
Analysis of external factors is the next highest-priority diagnostic step after log analysis.
I describe it in **section 5** below.

3. Determining the Source Control configuration and operational mode

First and foremost, any such diagnosis must begin by determining the Source Control configuration and operational mode of Azure Data Factory. Simply put, it is necessary to understand **how code versions are managed** and how the **version** on the «**Author**» **canvas** relates to the **published** version, which is used in production runs.
This is critically important for the correct diagnosis of historical failures (**Section 3**) and the analysis of the current logic (**Section 5**).
For this, it is necessary to check the **Git configuration**:

3.1.

Navigate to the «**Manage**» tab on the left side panel.

3.2.

Go to «**Source control**» → «**Git configuration**» or check for Git controls (e.g., the branch selector) on the top bar of ADF Studio.

3.3.

Determine the ADF operating mode.

3.3.1. `Live Mode`

In this mode, changes on the «**Author**» canvas can be saved using the «**Save**» or «**Save all**» buttons. These saved changes are available for «**Debug**» runs.
However, these changes are not activated for triggered runs until they are published directly to the Data Factory service using the «**Publish all**» button. Only the published version is the version used in triggered runs.

3.3.2. `Git Mode`

In this mode, changes on the «**Author**» canvas are saved to the selected Git branch (e.g., feature branch) using «**Save**» or «**Save all**».
These changes are not applied to the production service (and are not used in triggered runs) until they are published from the Collaboration branch.

4. Analysis of logs

4.1.

Navigate to Azure Data Factory Studio.

4.2.
Open the «**Monitor**» tab on the left side panel.

4.3.
In the «**Pipeline runs**» section, review the list of previous runs of the target pipeline.
Use the filters by pipeline name («**Pipeline name**») and status (e.g., «**Failed**») to find the runs of interest.
Ensure that triggered runs are analyzed.
Triggered runs are production runs and always use the version of the pipeline that was published in the Azure Data Factory service at the time of the run.
This corresponds to the published version in `Live Mode` (**point 3.3.1** above) or the version published from the Collaboration branch in `Git Mode` (**point 3.3.2**).
«**Debug**» runs use the current version of the pipeline on the «**Author**» canvas.

- In `Git Mode` this is the version from the currently selected branch.
- In `Live Mode` this is the current version on the canvas, which may contain unpublished changes.

4.4.
Analyze the details of the failed run.

4.4.1.
Click on the pipeline name in the list to navigate to the «**Activity runs**» list.

4.4.2.
Identify the activity that failed.

4.4.3.
Examine the detailed error message.
To do this, click on the icon in the «**Error**» column for this activity.

4.4.4.
Examine the input parameters that were used for this run.
To do this, find the «**Parameters**» column (at the pipeline run level or at the top of the Activity runs screen) and review the values that were actually passed to the pipeline.

4.5.
Execute a rerun («**Rerun**») of the failed execution.
Objective: To verify whether the cause of the failure was transient (e.g., network timeout, resource unavailability) or it is persistent (e.g., a configuration, logic, or data error).
The «**Rerun**» functionality uses the same input parameters (**point 4.4.4**) and the same pipeline configuration (Snapshot, see **point 4.6**) that were used in the original run.

4.5.1.
Rerun from the start of the pipeline.
In the «**Pipeline runs**» list (**point 4.3**), hover the cursor over the failed run and click the «Rerun» button.

4.5.2.
Rerunning from the failed activity.
This is the preferred option for multi-step pipelines, allowing to skip successfully completed steps and start execution directly from the activity that failed.
In the «**Activity runs**» view (**point 4.4.1**), use the «**Rerun from failed activity**» option (on the top panel) or «**Rerun from activity**» (for a specific activity).

4.5.3.
Analyze the result of the rerun.
If the rerun is successful, it is likely that the cause of the original failure was transient.
If the rerun also fails with the same error, this indicates a persistent problem that requires further analysis (starting from **point 4.6**).

4.6.
Analyze the snapshot of the pipeline configuration that was used during the failed run.

Objective: To see the exact configuration and code that led to the failure, as the current version of the pipeline may differ.

4.6.

Analyze the snapshot of the pipeline configuration used during the failed run.
Objective: To see the exact configuration and code that led to the failure, as the current version of the pipeline may differ.

4.6.1. In the «**Activity runs**» view (**point 4.4.1**), find the option to view the pipeline code.
This is typically an icon with the «**Code**» tooltip (in the form of curly braces), located on the top bar next to the pipeline name.

4.6.2.

Examine the JSON definition of the pipeline and related resources (e.g., Datasets) as they existed at the time of that specific run.

5. Analysis of external factors

5.1.

Navigate to the «**Manage**» tab on the left side panel.

5.2. Check Linked Services:

5.2.1.

Go to the «**Linked services**» section.

5.2.2.

Identify all Linked Services used in the target pipeline (they can be found in the settings of Datasets or Activities).

5.2.3.

For each Linked Service, open its configuration and use the «**Test connection**» function.
This will make it possible to verify the basic connectivity to the resource from the corresponding Integration Runtime. **Important note**: The «**Test connection**» functionality **is not executed in the context of the user account**.
It is executed on the Integration Runtime (Azure IR or Self-Hosted IR) specified in the Linked Service.

It uses the credentials defined in the Linked Service configuration (e.g., Managed Identity, Service Principal, Account Key).
A successful «**Test connection**» **does not guarantee a successful pipeline execution** for the following reasons:

5.2.3.1. Network differences

There may be differences in the network configuration between interactive testing and automated execution.
This is especially relevant when using Self-Hosted IR, Virtual Networks (VNet) or Private Endpoints.
Firewall rules, Network Security Groups (NSG) or routing may differ, blocking access during execution.

5.2.3.2. Insufficient permissions

A successful test verifies only authentication and network connectivity.
It does not guarantee the permissions required to perform specific data operations (e.g., reading a table, executing a Stored Procedure) in the Pipeline (see **point 5.2.5**).

5.2.4.

Identify the authentication method used in the Linked Service (e.g., System-Assigned Managed Identity, Service Principal, Account Key, SQL Authentication).

5.2.5.

Verify the permissions of the ADF security principal.
Check permissions at two different levels, as successful authentication does not mean the presence of the necessary authorization for data access.

5.2.5.1. Azure RBAC (Role-Based Access Control) level

If Managed Identity or Service Principal is used, verify that this security principal is assigned the necessary Azure RBAC roles for data access (Data Plane) on the target resource.
E.g., the «**Storage Blob Data Contributor**» role for Azure Storage.
This verification is performed outside of ADF Studio, in the «**Access Control (IAM)**» settings of the target resource on the Azure portal.

5.2.5.2. Level of internal permissions of the target system (Data Store Permissions)

For systems such as Azure SQL Database and SQL Pools in Azure Synapse Analytics, standard Azure RBAC (mentioned in **point 5.2.5.1**) does not manage granular access to objects within the database (Data Plane).
Even if authentication is handled by Microsoft Entra ID (Azure AD), authorization is controlled by T-SQL permissions.
It is necessary to ensure that the security principal has the necessary permissions within the target system.
The security principal (e.g., Managed Identity) must be added as a database user, e.g., using `CREATE USER […] FROM EXTERNAL PROVIDER`.
This user must be assigned the appropriate database roles (e.g., `db_datareader`, `db_datawriter`) or granted direct permissions (GRANT).
If the pipeline executes Stored Procedures, the user also requires the `EXECUTE` permission.
This check is performed by connecting to the database using management tools (e.g., SQL Server Management Studio (SSMS) or Azure Data Studio).

5.3. Check Integration Runtimes:

5.3.1.

Navigate to the «**Integration runtimes**» section.

5.3.2.

Check the status of the Integration Runtimes used in Linked Services.
Ensure that they are in the «**Running**» state, especially if Self-Hosted Integration Runtimes are used.

6. Synthesis of diagnostic results and identification of the causes of historical failures

Objective: To accomplish your strategic task — to determine why the pipelines did not work in the past, based on the collected data.

6.1.

Correlate the detailed error messages (**Section 4**) with the results of the environment check (**Section 5**).
Detailed error messages from historical runs are the primary source for diagnosing past failures. It is necessary to consider the temporal context when interpreting the results of the current environment check.
The results of the «**Test connection**» or the IR status reflect the current

state of the system.
This state could have changed since the historical failure (e.g., updating expired credentials, changing network rules).
It is not possible to rely solely on the current state to diagnose past problems.

6.1.1.

If the historical error indicates connection problems (e.g., «**Authentication failed**», «**Connection timed out**»), and the current «**Test connection**» also fails, it is likely that a persistent infrastructure or security configuration problem exists.

6.1.2.

If the historical error indicates connection problems, but the current «**Test connection**» (**point 2.2.3**) is successful, this may indicate that the problem was transient (e.g., network timeout, resource unavailability) or that the environment configuration has been corrected since the failure.

6.2.

Analyze the input parameters used in the failed runs.
Determine if the failures could have been caused by incorrect or unexpected data passed to the pipeline (e.g., an incorrect date format, a non-existent path).

6.3.

Identify failure patterns.
Determine whether the failures are persistent (often indicating a configuration or access issue) or intermittent/random (often indicating performance issues, resource locking, or a dependency on specific data).

6.4

Formulate conclusions about the root cause of historical failures and classify it (e.g., authentication, network, pipeline logic, data quality).

7. Identification and analysis of pipeline parameters

7.1.

Navigate to the «**Author**» tab on the left side panel to analyze the current development version of the pipeline.

7.1.1.
If the factory is operating in `Git Mode`, ensure that the correct branch is selected for analysis in the branch selector (usually the Collaboration branch or the relevant feature branch).

7.1.2.
If the factory is operating in `Live Mode`, check for an indicator of unpublished changes (e.g., the activity of the «**Publish All**» button).

7.2.
Find and open the target pipeline.
Important note: It is necessary to consider that this version on the canvas may differ from the published version which was analyzed in section 4.

7.3.
Click on the empty area of the pipeline canvas to display its general settings.

7.4.
Navigate to the «**Parameters**» tab.
Compile a list of all parameters, recording their «**Name**», «**Type**» (e.g., `String`, `Int`, `Array`, `Object`, `SecureString`), and «**Default value**», if specified.
Knowing the data type is critical, especially for complex types (`Array`, `Object`).

7.5.
Examine how parameters are used inside the pipeline.
Objective: To trace the data flow from pipeline parameters to the end resources (Linked Services, Datasets, Activities) to understand the dynamic configuration.

7.5.1. Analysis of usage in Activities
7.5.1.1.
Select each activity on the canvas (e.g., Copy activity, Lookup, Data Flow).

7.5.1.2.
Check the settings tabs (e.g., «**Settings**», «**Source**», «**Sink**») for the use of dynamic content.

7.5.1.3.

Identify the expressions that directly reference pipeline parameters, using the syntax `@pipeline().parameters.<parameterName>`.

7.5.2. Analysis of usage in Datasets

Many Activities (e.g., Copy, Lookup, Data Flow) use Datasets to define the Source and the Sink.
The pipeline can pass its parameters to Dataset parameters.

7.5.2.1.

For each activity that uses a Dataset (see point 7.5.1), check the Source or Sink configuration.

7.5.2.2.

Identify the values passed to «**Dataset properties**» (or «**Parameters**» depending on the Activity type).
Determine whether these values are static or dynamic expressions referencing the pipeline parameters.

7.5.2.3.

Open the configuration of the corresponding Dataset (the «**Open**» button).

7.5.2.4.

In the Dataset, navigate to the «**Parameters**» tab to see the list of parameters that it expects.

7.5.2.5.

In the Dataset, navigate to the «**Connection**» tab.
Examine how the Dataset parameters are used for the dynamic configuration of properties (e.g., `File path`, `Table name`), using the syntax `@dataset().<datasetParameterName>`.

7.5.3. Analysis of usage in Linked Services

Datasets use Linked Services to connect to data stores.
A Dataset can pass its parameters to the parameters of a Linked Service.

7.5.3.1.

In the Dataset configuration (the «**Connection**» tab, **point 7.5.2.5**), examine the «**Linked service properties**» section.
Identify the values passed to the Linked Service parameters.
Determine if they reference the Dataset parameters.

7.5.3.2.

Open the configuration of the corresponding Linked Service (go to «**Manage**» → «**Linked services**» or use the edit option from the Dataset).

7.5.3.3.

In the Linked Service, examine the «**Parameters**» section. Examine how the Linked Service parameters are used in the connection string (`Connection string`), URL or other connection properties (e.g., `Database name`, `Secret name` for Azure Key Vault), using the syntax `@linkedService().<linkedServiceParameterName>`.

7.6.

Compare the current pipeline version with the snapshot of the historical run.
Objective: To identify discrepancies between the code that caused the failure and the code available for analysis and testing.

7.6.1.

Obtain the JSON representation of the current pipeline version.
To do this, use the «**Code**» button (in the form of curly braces) on the top bar of the «**Author**» tab..

7.6.2.

Compare this JSON representation with the JSON configuration from the snapshot of the historical run (point 4.5.2).

7.6.3.

Identify any differences in the logic, structure, expressions, or configuration of activities and parameters.

7.6.4.
If differences exist, the further analysis of the logic (**point 7.5**) and the test run (**Section 10** below) must account for these differences.

8. Identification of parameter sources
Objective: To determine how the pipeline receives parameters in the production environment (via Triggers or from a parent pipeline through the `Execute Pipeline` activity) in order to emulate these values during manual testing.

8.1.
8.1.1.
In the pipeline editor (the «**Author**» tab), click the «**Trigger**» button on the top bar and select «**New/Edit**».

8.1.2.
In the opened panel, examine the configuration of existing triggers associated with this pipeline.

8.2.
Examine the parameters passed by the trigger.

8.2.1.
While proceeding through the trigger configuration, pay attention to the step where the pipeline parameters are filled in.

8.2.2.
Identify the use of System Variables.
These are expressions that start with `@trigger()` or `@triggerBody()`.
E.g.:

- `@trigger().outputs.windowStartTime` (for Tumbling Window Trigger).
- `@triggerBody().fileName` (for Storage Event Trigger).

Production pipelines often receive metadata (e.g., timestamps or file names) automatically from triggers via System Variables.
During manual testing (Debug), it is necessary to emulate these values manually.

Unawareness of this dependency is a common cause of issues with manual runs.

8.3.

Analyze Parent Pipelines and the `Execute Pipeline` activity.
Objective: To determine whether the target pipeline is called by another pipeline and what parameters are passed.

8.3.1. Identify Parent Pipelines

8.3.1.1.

While in the target pipeline editor (the «**Author**» tab), open the pipeline properties pane (the «**Properties**» icon).

8.3.1.2.

Navigate to the «**Related**» tab.

8.3.1.3.

In the «**Pipelines**» section, examine the list of pipelines that reference the current one (they use the `Execute Pipeline` activity to call it).

8.3.1.4.

As an alternative or for re-verification, use the Global Search in Azure Data Factory Studio by the name of the target pipeline.

8.3.2. Examine the configuration of the `Execute Pipeline` activity in the Parent Pipeline

For each Parent Pipeline identified in **point 8.3.1**:

8.3.2.1.

Open the Parent Pipeline on the «**Author**» tab.

8.3.2.2.

Find the `Execute Pipeline` activity that calls the target pipeline.

8.3.2.3.

Select this activity and navigate to the «**Settings**» tab.

8.3.2.4.

In the «**Parameters**» section, examine what values or expressions (dynamic content) are passed to the parameters of the target pipeline.

8.3.3 Determine the sources of the values in the Parent Pipeline

Values can be static, derived from Parent Pipeline parameters (e.g., `@pipeline().parameters.<ParentParameterName>`), variables, or the Outputs of preceding activities (e.g., `@activity('<ActivityName>').output.<PropertyName>`).

8.4. Determine the values for testing

Use a combination of values from historical runs (**Section 4**), default values (**point 7.4**), emulated trigger values (**point 8.2.2**), and values passed from Parent Pipelines (**point 8.3.3**) to form a set of test input data.

9. Safety Check

Objective: To prevent accidental damage, modification, deletion, or duplication of production data when running the legacy pipeline.
Rationale: Running the pipeline without a full understanding of its logic and the configuration of its data Sinks poses critical risks to data integrity.

9.1. Activate the «Data Flow Debug» session (if applicable)

Objective: To ensure a Spark execution environment for `Data Flow` activities during debugging and interactive analysis.

9.1.1. Determine if the pipeline contains `Data Flow` activities.

9.1.2.

If `Data Flow activity` are present, an active debug session is required for their execution in «**Debug**» mode (**Section 10**) or for using «**Data Preview**» (**point 9.4.2.1**).

9.1.3.

On the top toolbar of ADF Studio, activate the «**Data Flow Debug**» switch.

9.1.4.

Select the `Integration Runtime` configuration and set the `Time to live` (TTL).

9.1.5.
Wait for the cluster to be ready (the status indicator will turn green).
The cluster startup (cold boot) may take several minutes.

9.2. Identify data modification operations and Sinks
Analyze all Activities in the pipeline that can potentially modify data in external systems:

9.2.1.
`Copy activity` (to examine the «**Sink**» tab).

9.2.2.
`Data Flow` (to examine all Sinks inside the data flow).

9.2.3.
Stored Procedure activity and Script activity (can perform DML/DDL operations: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`).

9.2.4.
Delete activity.

9.2.5.
Web, Webhook, Custom activities (if they call an API to modify data).

9.3. Analyze target systems
- Determine for each activity from the points of section 9.2 where exactly data will be written or deleted.
- Examine the configuration of the target Datasets and Linked Services.
- Determine whether they point to production (Production) systems.

9.4. Apply a safe run strategy
If the pipeline can affect production data, it is necessary to apply one or more safety measures.
The methods are listed in order of preference.

9.4.1. Strategy 1: Redirecting writes to a safe location (The most reliable method)

9.4.1.1.

Via parameters: If the Sink configuration (path, table name, connection string) is parameterized (see section 3), ensure that during the run (section 10) values pointing to a test environment (e.g., a `test` folder, a `dev` schema) are used.

9.4.1.2.

Through temporary configuration change: If parameters are not used, temporarily change the settings of Datasets or Linked Services to point to a test environment.
Important: Do not save and do not publish (Publish) these changes.

9.4.2. Strategy 2: Use of isolation and control mechanisms.

9.4.2.1. For `Data Flow` activities

Use «**Data Preview**» for data inspection. This requires an active «**Data Flow Debug**» session (**point 9.1**).
In the Data Flow editor, select the transformation immediately preceding the Sink.
Navigate to the «**Data Preview**» tab and click «**Refresh**» to obtain an interactive snapshot of the data.
This makes it possible to verify the transformation logic without writing data to the target system.
Critical note about Pipeline Debug Run:
There is a fundamental difference between interactive debugging («**Data Preview**») and running the pipeline in «**Debug**» mode (**Section 10**).
The statement that data are not written to the Sink is true only for «**Data Preview**» inside the Data Flow editor.
When running the entire pipeline in «Debug» mode, the Data Flow activity will perform writes to Sink transformations.
Therefore, to prevent writes when running the pipeline, it is necessary to use other strategies (e.g., **points 9.4.1** or **9.4.2.2**).

9.4.2.2 For Activities on the canvas

Use Breakpoints for Iterative Debugging.
Set a breakpoint on an activity to pause the pipeline before its execution.
To do this, click on the empty red circle in the upper-right corner of the activity (the «**Debug Until**» option).

The circle will change to a filled red circle.
When running in «**Debug**» mode, the pipeline will execute only up to the activity with the breakpoint. The breakpoint activity itself is not included in the test run.
This makes it possible to check the intermediate results and input data of the preceding activities before executing potentially dangerous operations.

9.4.2.3 Deactivating activities

If the analysis of data writes is not the objective of the test, temporarily deactivate (the «**Deactivate**» button on the top bar) the activities that perform modifications.

9.4.3. Strategy 3: Minimization of impact (Use with caution)

9.4.3.1. Limitation of data volume

To apply methods for limiting the number of processed rows, specific to the activity type.

9.4.3.1.1. For `Data Flow` activities

To open «**Debug Settings**» on the `Data Flow` canvas toolbar.
Set the «**Row limit**» to a small value (e.g., 100-1000).
This setting limits the number of rows read from the sources (`Sources`) only for the current debug session.
This requires an active «**Data Flow Debug**» session (**point 9.1**).

9.4.3.1.2. For `Copy activity` and `Lookup activity`

The «**Row limit**» setting from **point 9.4.3.1.1** does not apply.
It is necessary to temporarily modify the Source configuration.
E.g., if the source is an SQL database, modify the query (Query) by adding `TOP N` or `LIMIT` (e.g., `SELECT TOP 100 * FROM schema.table`).
Warning: Limiting the data volume limits reading from the source, but does not ensure complete safety if the pipeline performs operations that do not depend on the data volume (e.g., `TRUNCATE TABLE` via the «**Pre-copy script**» or deleting files via the `Delete activity`).

9.5. Run Cancellation Condition (Go/No-Go Decision)

If none of the methods in **point 9.4** can guarantee the safety of production data, do not execute the run (**Section 10**). In this case, clone and modify the pipeline for testing in an isolated environment.

10. Test Run

Objective: To run the pipeline with the correct parameters in a controlled mode and to verify its operational capability.

10.1. While on the «Author» tab in the editor of the target pipeline, select a run strategy and start the execution in «Debug» mode.

Rationale: The «**Debug**» mode is used for testing the current version of the pipeline on the canvas (including unpublished changes) and allows for observing the execution in real time without the need for publication.
Rationale:
The «**Debug**» mode is used for testing the current version of the pipeline on the canvas (including unpublished changes) and allows for observing the execution in real time without the need for publication.

10.1.1.

Choice of strategy: Iterative Debugging or Full run.
For complex or legacy pipelines, it is recommended to use iterative debugging for a step-by-step and safer analysis.

10.1.2.

For iterative debugging («**Debug Until**»): Ensure that breakpoints are set (**Section 9**).
Click the «**Debug**» button.
The pipeline will execute only up to the first activity with the breakpoint.

10.1.3.

For a full run: Ensure that breakpoints are not set (or will be ignored if it is required to execute the pipeline in its entirety despite them). Click the «**Debug**» button.

10.2.

In the panel that opens (usually named «**Pipeline Debug**» or «**Pipeline run parameters**»), enter the values for the parameters defined in section 8.

10.3. To observe the correct data input format:

10.3.1.

When emulating trigger timestamps, to use the ISO 8601 format (e.g., `2025-08-27T10:00:00Z`).

10.3.2.

For parameters of type `Array`, to enter the value in the JSON array format.
E.g.: `["item1", "item2"]`.

10.3.3.

For parameters of type `Object`, to enter the value in the JSON object format.
E.g.: `{"key1": "value1", "key2": "value2"}`. Rationale: Correct input of parameters, especially for complex types (`Array`, `Object`), is necessary for a successful run.
An incorrect JSON format will lead to validation errors even before the execution begins.

10.4.

Click «OK» to start the execution.

10.5.

Observe the execution progress on the «**Output»** tab under the pipeline canvas.
Check the input and output data for each Activity after its completion to verify that the parameters were interpreted correctly.