Conditional Tasks & Repairing Runs

Exploring workflows and task management techniques

Agenda Items

- Workflow-Level Conditionals
- Configuring Conditional Tasks in Databricks Workflows
- Loops in Workflows
- Repairing Failed Runs and Alerts
- Best Practices
- Hands-On Lab

Workflow-Level Conditionals

What are Workflow-Level Conditionals?

Conditional Workflow Basics

Workflow-level conditionals allow jobs to branch or skip steps based on task outcomes or dynamic values, enhancing flexibility.

Configurable in Databricks

These conditionals require no coding expertise and can be easily configured within the Databricks Jobs UI.

Looping and Dynamic Tasks

Incorporating loops and dynamic tasks allows for automated responses based on varying conditions in workflows.

Repairing Failed Runs

Workflow conditionals can also help in managing and retrying failed runs, ensuring smoother job executions.

Common Use Cases

Conditional Notifications

Run notifications only when the report is empty, ensuring that alerts are meaningful and necessary.

Data Archiving Process

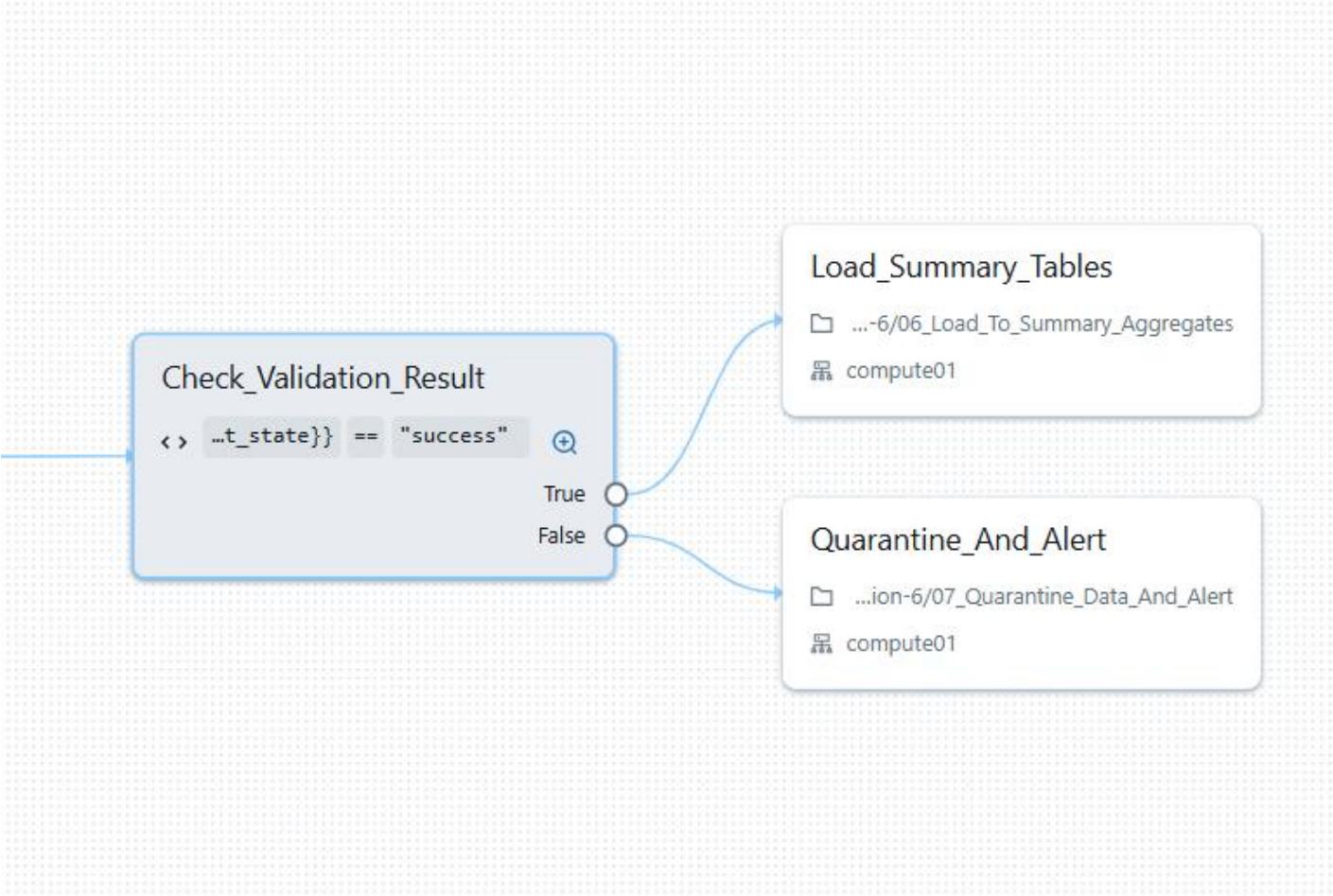
Archive data only if the quality check passes, maintaining high data integrity and relevance.

Task Branching Logic

Incorporate 'If Blocks' to branch logic in workflows, allowing for dynamic task execution based on conditions.

Configuring Conditional Tasks in Databricks Workflows

Adding Conditional Tasks



Jobs User Interface

The Jobs UI displays a visual representation of tasks and conditions, making it easy to manage workflows.

Adding If Block

Incorporating an 'If Block' allows for conditional task execution, enhancing workflow flexibility and control.

Branching Tasks

'Branching tasks' enable different pathways in the workflow based on conditions, improving task management.

Adding a Conditional Task (UI Walkthrough)

Validation or Metric Task

Begin by adding a validation or metric task that sets the criteria for your conditional workflow.

Adding an If Block

Next, insert an 'If Block' to create a conditional structure that directs workflow based on specified criteria.

Setting Conditions

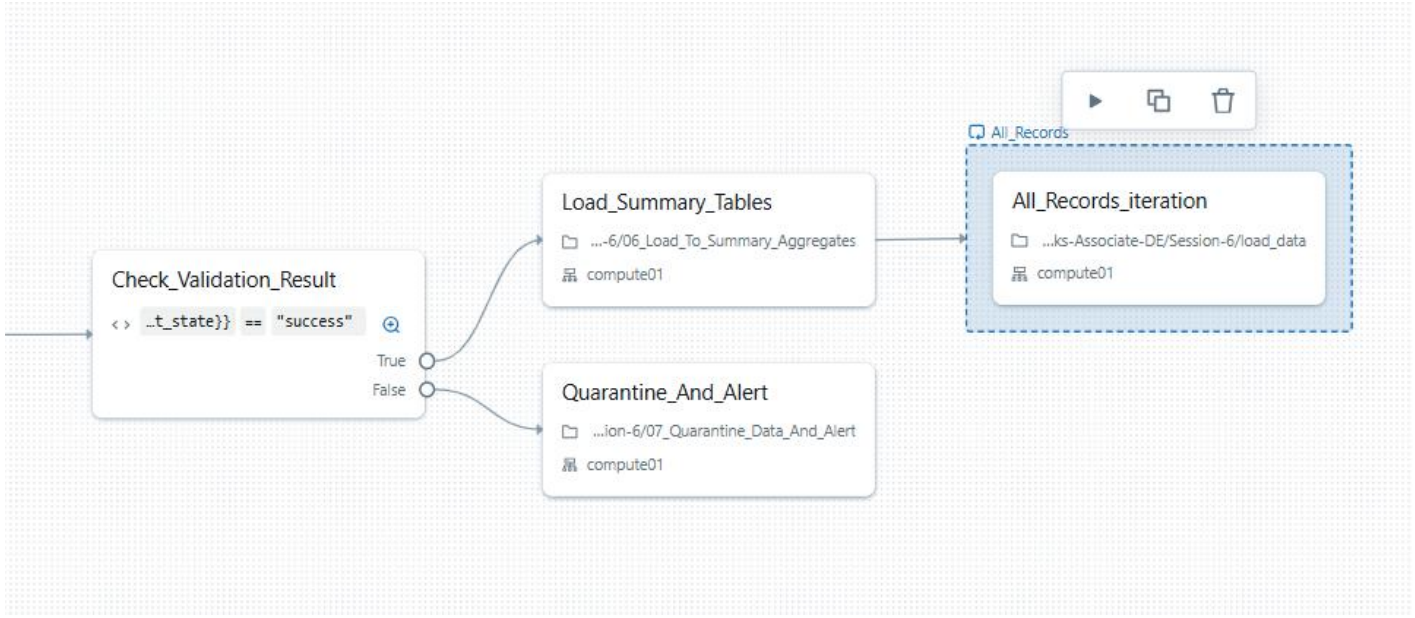
Define the conditions for the workflow using logical statements such as validation results or row counts.

True and False Branches

Add tasks to the 'True' and 'False' branches to handle outcomes based on the conditions you set.

Loops in Workflows

Using 'For Each' Blocks



Purpose of 'For Each'

'For Each' blocks allow you to streamline tasks by processing multiple items in a single workflow step, enhancing efficiency.

Dynamic List Definition

You can define the list for 'For Each' blocks dynamically or based on the output from previous tasks, adapting to various data sources.

Parallel Execution

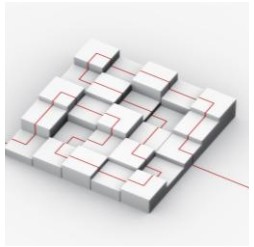
All runs within 'For Each' blocks can be executed in parallel, leading to improved performance and reduced processing time.

Foreach Example with REST API

```
regions = ["North", "South"]
for r in regions:
payload = {
"job_id": 123,
"notebook_params": {"region": r}
}
response =
dbutils.notebook.entry_point.getDbutils().notebook().get
Context().apiClient().performQuery(
"POST", "/api/2.1/jobs/run-now", payload
)
```

Iterating Over Regions

This example demonstrates how to iterate over a list of regions using a foreach loop in Python.

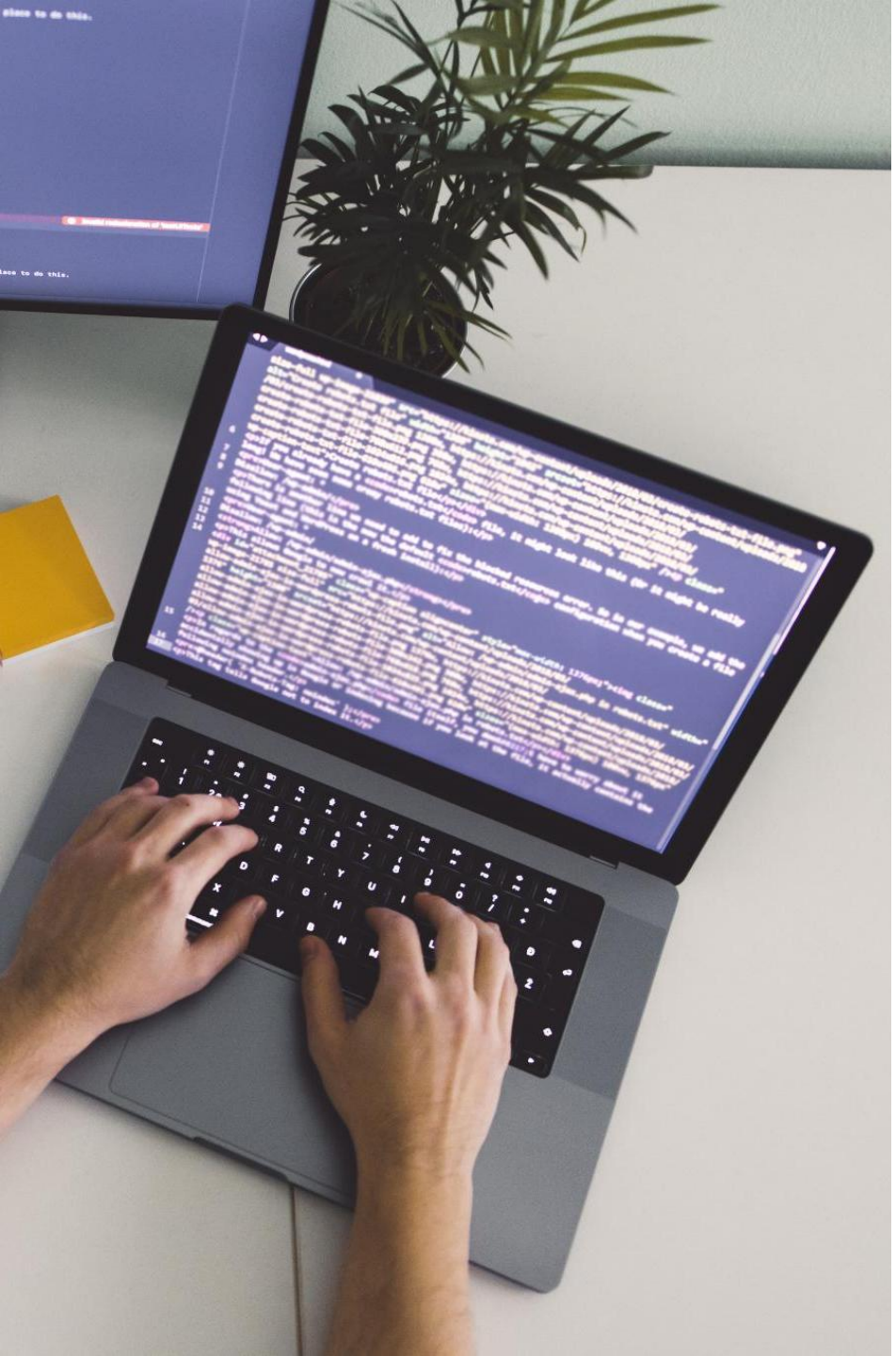


Payload Structure

The payload structure contains job_id and notebook parameters, showcasing how data is organized for the API call.

Performing API Query

The code performs a POST request to the REST API with the specified payload, executing a job.



Notebook Tasks

Notebook Tasks Overview

Notebook tasks allow users to run code in multiple programming languages such as PySpark, Python, and Scala.

Full Support for Languages

With notebook tasks, developers can utilize the full power of programming languages for data processing and analysis.

Comparison with SQL Tasks

Notebook tasks offer more flexibility compared to SQL tasks, which are limited to SQL syntax.

SQL Tasks

Executing SQL Queries

SQL tasks allow for executing queries directly on Lakehouse tables, providing efficient data management.

Parameter Support

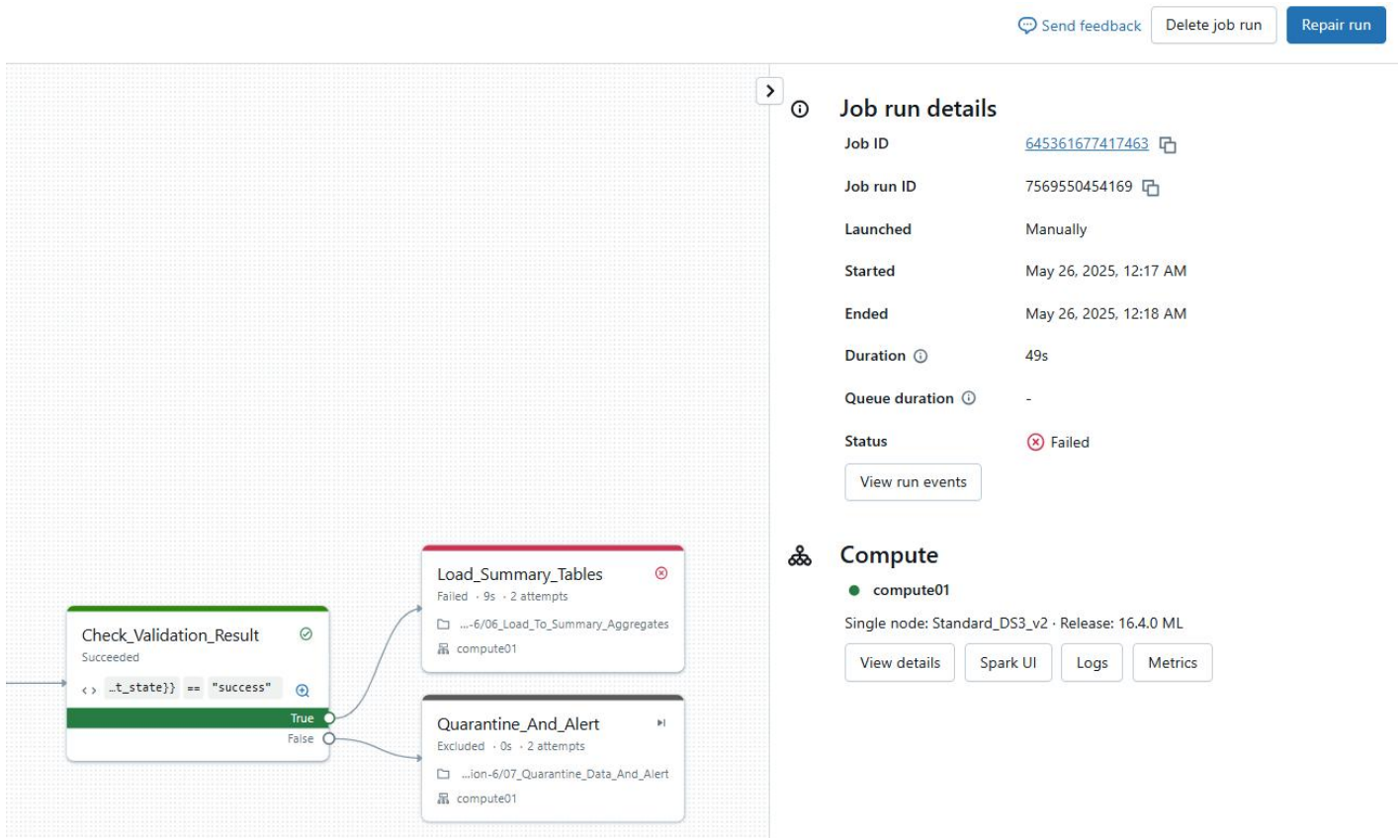
SQL tasks support parameters through dbutils.widgets or dbutils.jobs.task, enhancing flexibility and functionality.

Parameter Types and Access Methods

PARAMETER TYPE	ACCESS METHOD	SCOPE	USE CASE
Job Parameter	dbutils.widgets.get("param")	Global (Job-wide)	User input, global config
Task Parameter	dbutils.jobs.task.get("param")	Task-specific	Pass data between tasks

Repairing Failed Runs and Alert

Steps to Repair Failed Runs



Accessing Job Runs

To start the repair process, navigate to the Job Runs tab and select the failed job you wish to repair.

Repairing the Run

After selecting the failed run, click the 'Repair run' button to initiate the repair process.

Rerun Tasks

Finally, rerun only the failed and downstream tasks to ensure the job completes successfully.

Notifications & Alerts

Email Alerts Configuration

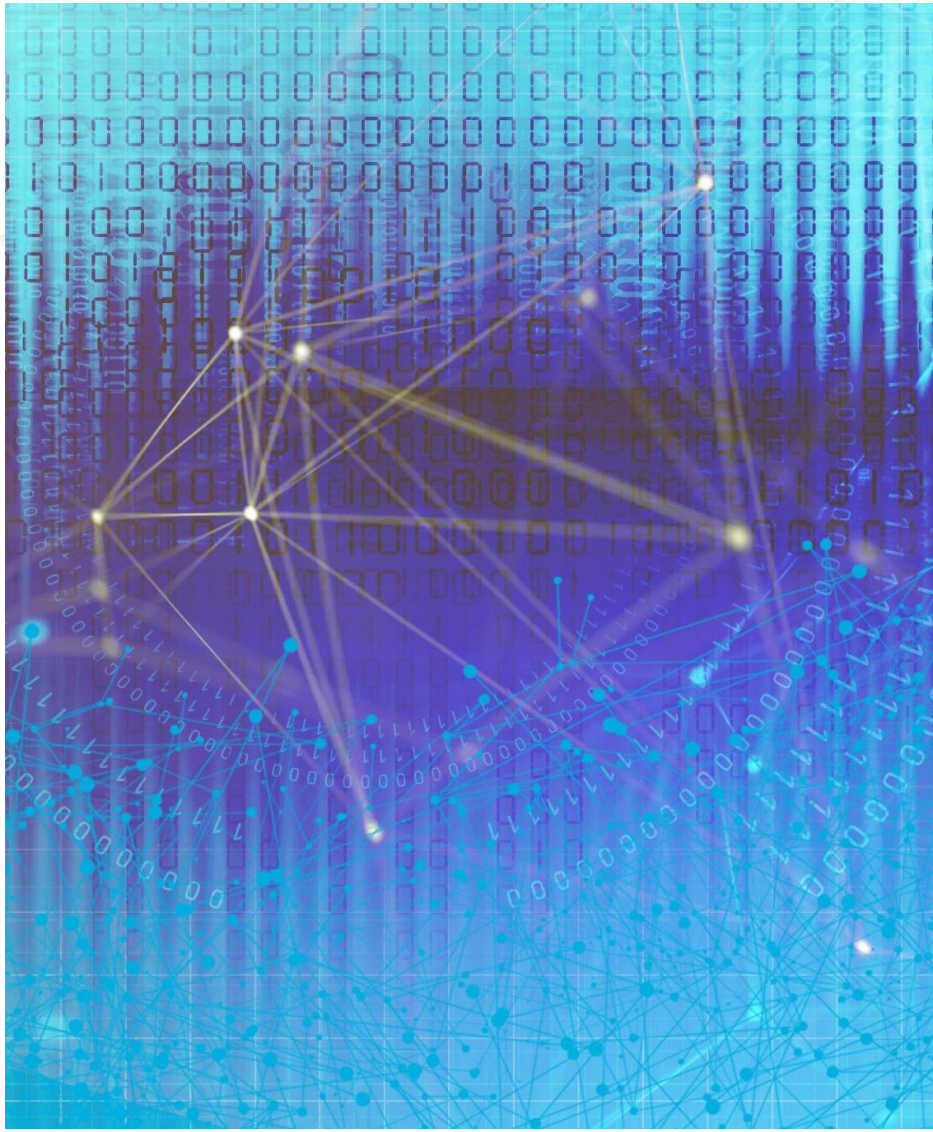
Set up email alerts for various scenarios like success, failure, and timeout to stay informed effectively.

Integration Options

Integrate notifications with webhooks or Azure Logic Apps to enhance functionality and response time.

Automated Operational Awareness

Leverage notifications to automate operational awareness, ensuring timely responses to critical events.



Best Practices

Best Practices

Use Compute Clusters

Utilizing Jobs Compute clusters optimizes workflows for efficiency and scalability across various tasks.

Parameterize Inputs

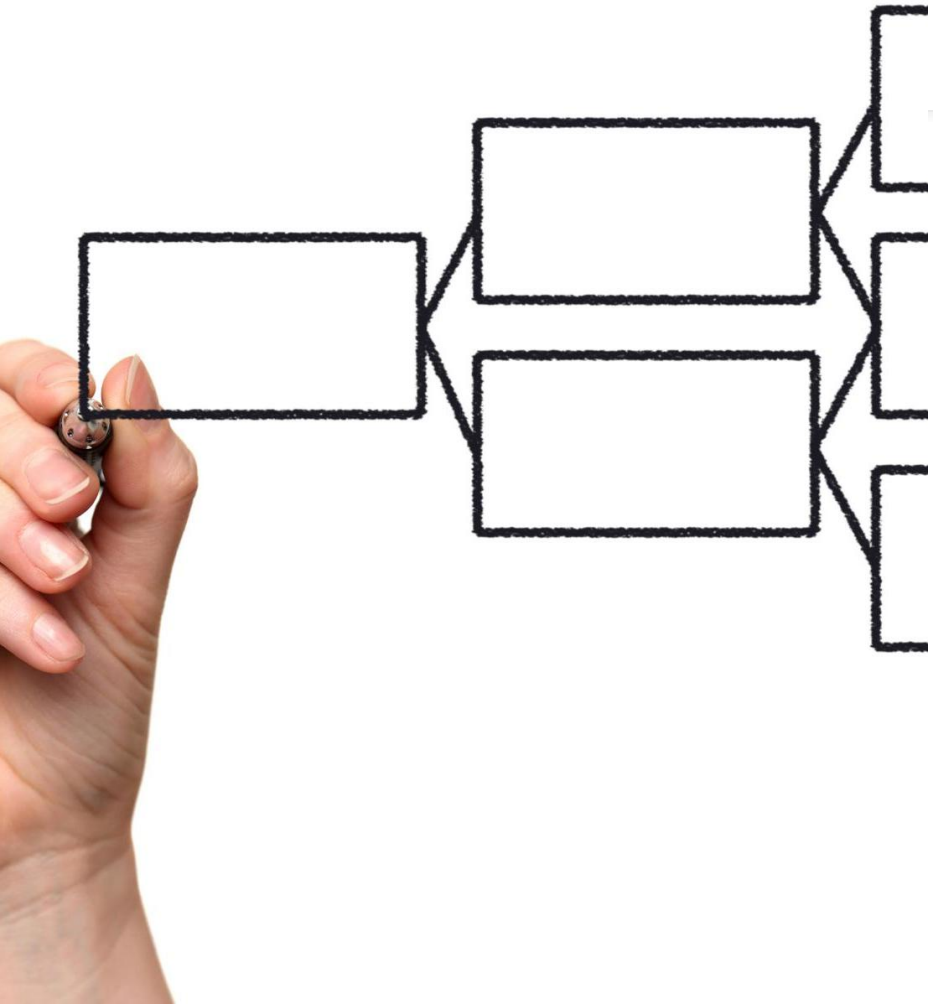
Parameterizing all inputs helps to avoid hardcoding, making workflows more flexible and adaptable to changes.

Modularize Logic

Breaking down logic into small, reusable tasks enhances the modularity and maintainability of workflows.

Monitor and Alert

Proactively monitoring runs and configuring alerts ensures better performance management and issue resolution.



Naming and Documentation

Descriptive Naming

Using descriptive names for tasks and branches enhances clarity and understanding in documentation and coding practices.

Handling Else Branch

Always handle the 'Else' (False) branch to ensure completeness and avoid potential errors in logic.

Documentation Practices

Documenting the logic in Job Descriptions provides clarity on role expectations and responsibilities, improving team communication.

Hands-On Lab