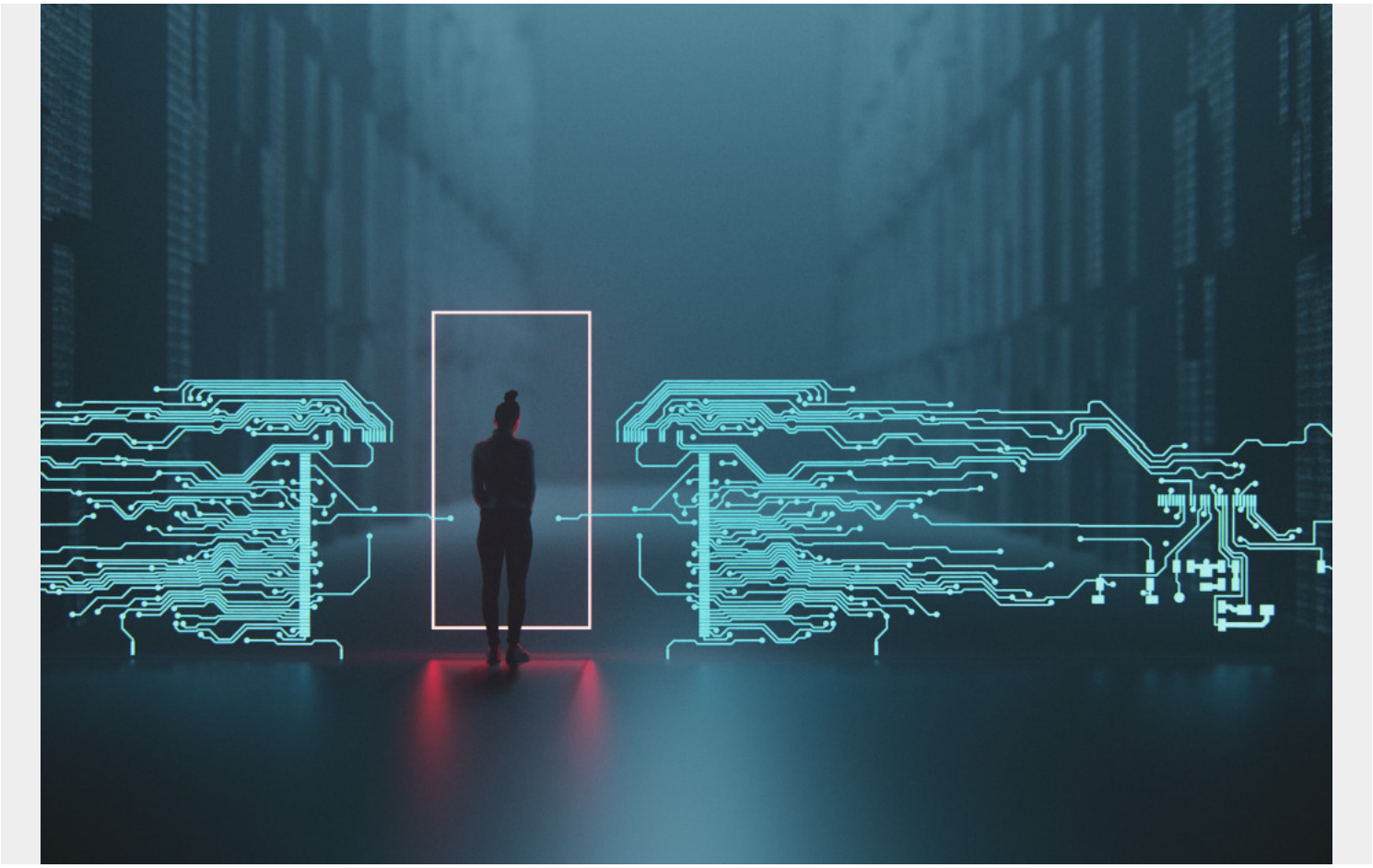
12 BEST PRACTICES FOR IMPLEMENTING APPLICATION WORKFLOW ORCHESTRATION



Companies across many industries have embraced application workflow orchestration as a way to drive digital modernization forward. From streamlining targeted advertising campaigns to automating predictive maintenance programs, application workflow orchestration platforms like Control-M are playing a critical role in helping businesses deliver better customer experiences.

> ***Application workflow orchestration*** makes sure data and application workflows are carried out in the correct sequence and at the correct time to ensure the successful delivery of a business service.

If you're ready to start your application workflow orchestration journey, here are 12 best practices to follow.

1. **Support an "as-code" approach**

Regardless of whether workflows are authored via some graphical interface or written directly in

code, version control is mandatory. Of course, to enable modern deployment pipelines, your platform should allow you to store and manage workflows in some text or code-like format.

2. **Think in microservices**

Avoid monoliths. This applies to workflows just as it does to applications. Identify functional components or services. Use an "API-like" approach for workflow components to make it easy to connect, re-use, and combine them, like this:

Service (Flow) A: Do something1, emit "something1 done" Emit "something2 running," Do something2, Emit "something2 done" Emit "Service A" done	Service B: Wait for "Service A" done Do BThing, emit BThing done	Service C: DO NOT run while something2 is running Wait for BThing done Etc.

3. **Don't reinvent the wheel**

If you have a common function, create a single workflow "class" that can be instantiated as frequently as required, yet maintained only once. Instead of creating multiple versions of a service, use variables or parameters that can accommodate the variety.

4. **Process lineage**

Data lineage is frequently cited as a major requirement in complex flows to support problem analysis. Process lineage is just as important and a mandatory requirement for effective data lineage. Without the ability to track the sequence of processing that brought a flow to a specific point, it is very difficult to analyze problems. The need for process lineage arises quickly when a problem occurs in a pub/sub or "launch-and-forget" approach used in triggering workflows.

5. **Make the work visible**

Process relationships should be visible. Have you ever encountered a situation where everything appears perfectly normal, but nothing is running? That's when visualization is particularly valuable. Having a clear line of sight between a watcher or sensor that is waiting for an event and the downstream process that wasn't triggered because the event did not occur can be extremely valuable.

6. **Codify SLAs**

The best way to define a non-event as an error is by defining an "expectation," commonly called a service level. At its most basic, an unmet service level agreement (SLA) is identified as an error. For example, we expect a file to arrive between 4 PM and 6 PM. It takes approximately 15 minutes to cleanse and enrich the file and another 30 minutes to process it. So, we can set the SLA to be 6:45 PM. If by then the processing is running late or hasn't started yet, and the flow hasn't completed, the error can be recognized at 6:45 PM.

A more sophisticated approach is to use trending data to predict an SLA error as early as possible. We know the cleanse step runs approximately 15 minutes because we collect the actual execution time for the last "n" occurrences. The same is true for the processing step. If the cleanse step hasn't finished by 6:15, or the processing step hasn't started by 6:15, we know we'll be late. We can generate alerts and notifications as soon as we know, so that we have the maximum time to react

and possibly rectify the problem.

A final enhancement is providing “slack time” to inform humans how much time remains for course correction. In the above scenario, if the cleanse step doesn’t start on time, at 6 PM, there are 45 minutes available to fix the problem before the SLA is breached.

7. **Categorize**

As you turn your workflow “microservices” and connecting tasks into process flows, make sure you tag objects with meaningful values that will help you identify relationships, ownerships, and other attributes that are important to your organization.

8. **Use coding conventions**

Imagine creating an API for credit card authorization and calling it “Validate.” While it makes sense to you, it may be too vague. Consider qualifiers that will carry more meaning such as “CreditCardValidation”. It is important to keep this in mind when you are naming workflows. It may be great to call a workflow “MyDataPipeLine” when you are experimenting on your own machine, but that gets pretty confusing even for yourself, never mind the dozens or hundreds of others once you start running in a multi-user environment.

9. **Think of others**

You may be in the relatively unique position of being the only person running your workflow. More likely, that won’t be the case. But even if it is, you don’t want to have to re-learn each workflow every time you need to modify or enhance it or analyze a problem. Include comments or descriptions on your workflows, or if it’s really complicated, add some documentation. And remember to revise them together with the workflow.

10. **Keep track**

Inquiring minds want to know…everything. Who built the workflow, who ran it, was it killed or paused, who did it and why? Did it run successfully, or did it fail? If so, when and why? How was it fixed? And so on. Basically, when it comes to workflows for important applications, you can never have too much information. Make sure your tool can collect everything you need.

11. **Prepare for the worst**

You know tasks will fail. Make sure you collect the data required to fix the problem and keep it around for a while. That way, you not only meet the “Keep track” requirement, but when problems occur, you can compare the new failure to previous failures or successes to help determine the problem.

12. **Harness intelligent self-healing**

Finally, look for flexibility in determining what is success and what is failure. It’s correct and proper to expect good code, but we have all seen code that issues catastrophic error messages even though the task completes with an exit code of zero. You should be able to define what is and isn’t an error as well as the automated recovery actions for each specific situation.