User Guide

Author: Barbaros Özdemir
With AI-assisted drafting via Antigravity (Google DeepMind)

© IBM Corporation 2026

1 Introduction

Modern software projects have no shortage of orchestration tools. Most of them require you to learn a framework, write glue code, maintain YAML pipelines, or spin up infrastructure just to move data from A to B with a few decisions in between. This engine takes a different approach.

You design your workflow in draw.io, a tool many teams already use for diagrams, architecture sketches, and process documentation. You save the file. **The engine runs it.** That is the entire authoring loop. There is no DSL to learn. No orchestration framework to configure. No separate deployment artifact per workflow. A process diagram is the executable definition of the process, and the same file that your business analyst puts in front of a stakeholder is the file the engine runs in production. Changes to the process are changes to the diagram. The source of truth is always visible and always human-readable.

Under the hood the engine is a single Python file. **It runs on** a **laptop (or VM)** , inside a container on a **Kubernetes cluster**, or even on a **serverless platform** like IBM Code Engine. It connects to IBM Cloud Object Storage or MinIO for model loading, and to Cloudant or CouchDB for state persistence — or to nothing at all if you just want to run locally with files. It has **no mandatory cloud dependencies** and **no licencing cost**.

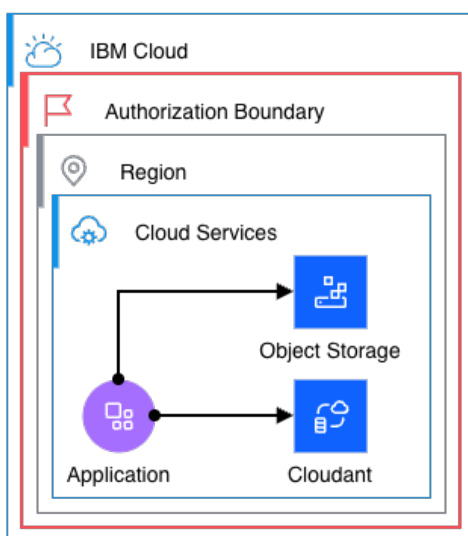
For developers this means you can drop a workflow engine into a project with one pip install and one command. You can model complex branching logic, parallel execution, REST callbacks, sub-processes, and timed waits without writing a single line of orchestration code. When a process changes, you update the diagram and redeploy, or in many cases, just drop the new file into the model directory.

This document will show you how to set up, configure, and run the workflow engine, and how to build and execute workflows with it.

© IBM Corporation 2026

2 Installation and Deployment

The engine can be deployed in three ways: as a command-line tool run directly on a local machine, as a long-running HTTP server on a VM or local machine, or as a container on Kubernetes or a serverless platform. All three modes use the same single Python file as their core. On diagram 1 below one of these deployments is shown. Note that models are stored in Object Storage and process state is saved in Cloudant NoSQL database.



Dia. 1: Architectural Overview of a Deployment on IBM Cloud, where the Application (i.e. this workflow engine) is deployed on the serverless platform “Code Engine”

2.1 Prerequisites

The following are required for all deployment modes:

Python 3.11 or higher must be installed on the machine or base image. All third-party dependencies are listed in requirements.txt and are installed via pip. A draw.io model file (.drawio) is required to run a workflow, the engine has no built-in demo model.

The following are optional depending on your setup:

Podman or Docker is needed only for containerised deployments. IBM Cloud Object Storage (or a compatible S3 store such as MinIO) is needed if you want to load models from object storage rather than the local filesystem. IBM Cloudant (or a compatible CouchDB instance) is needed if you want to persist workflow state to an external database rather than local JSON files. If neither is configured the engine runs entirely on the local filesystem.

© IBM Corporation 2026

2.2 Local setup

Clone or download the project files to a local directory. Install the Python dependencies:

```
pip install -r requirements.txt
```

Place at least one .drawio model file in the same directory as main.py.

To start the engine as an HTTP server on the default port 8080:

```
python main.py --server
```

To run a workflow directly from the command line without starting a server, pass the model name and an optional instance ID:

```
python main.py mymodel --id test-001
```

The engine reads CREDENTIALS.txt from the current directory on startup. Copy and edit the provided template to configure your object storage and authentication settings.

Note that you may need to use `python3` and `pip3` instead of `python` and `pip` when executing these command in your environment.

2.3 Container setup

Build the container image from the project directory:

```
podman build -t drawio-workflow-engine -f Containerfile .
```

Run the container:

```
podman run --rm -p 8080:8080 --name drawio-workflow-engine drawio-workflow-engine
```

If you prefer not to bake CREDENTIALS.txt into the image, remove the corresponding COPY line from the Containerfile and mount the file at runtime:

```
podman run --rm -p 8080:8080 -v "$(pwd)/CREDENTIALS.txt:/app/CREDENTIALS.txt:Z" drawio-workflow-engine
```

Credentials and configuration can also be passed as environment variables, which take precedence over values in CREDENTIALS.txt. This is the recommended approach for Kubernetes and serverless deployments. Refer to PODMAN.md for the full reference of supported environment variables, connectivity options, and a Docker Compose example for local development. If you want to emulate the cloud deployment on your local machine you can use;

- https://hub.docker.com/r/ibmcom/cloudant-developer
- https://hub.docker.com/r/minio/minio

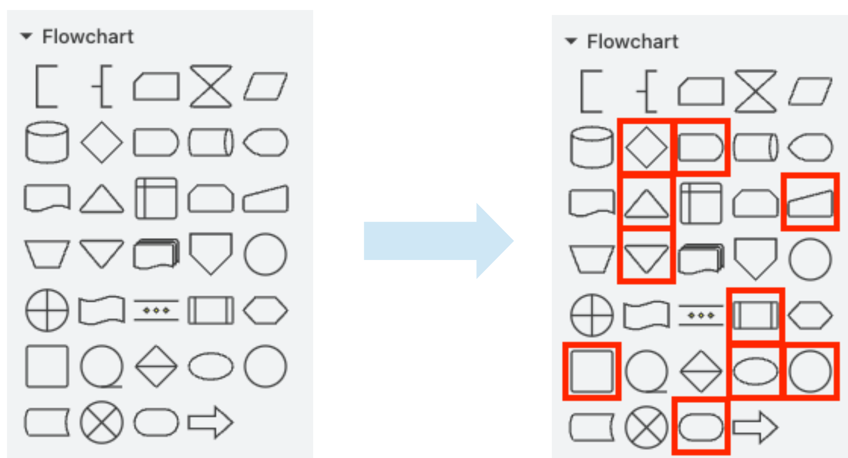
© IBM Corporation 2026

3 How it works

The workflow engine turns business process diagrams created in draw.io into executable, stateful workflows. There is no code to write for the process logic itself, the diagram is the program. The engine takes care of execution, state management, parallel branching, waiting for external events, and calling external services, while keeping a full audit trail of every step.

3.1 Modeling

Workflows are **designed visually** in draw.io using a defined sub-set of flowchart shapes and the line with **arrow** at its end (connector). The shapes used by this workflow engine are marked in the diagram 2 below.



Dia. 2: Sub-set of Flowchart shapes used by the workflow engine

There are 10 shapes in total and two of them are both “**Start**” element, so that there are actually 9 different elements in use. Each one of the 9 shape types has a specific role in the execution:

- **Start** (mxgraph.flowchart.start_1, mxgraph.flowchart.start_2) and **End** (mxgraph.flowchart.terminator) nodes mark the entry and exit points of the workflow.

- **Process** (label) nodes represent a unit of work. A process can merge a set of fixed values into the workflow context, call an external REST API and absorb the response into the context, or invoke another draw.io model as a sub-process (predefined process).

- **Predefined Process** (process) nodes invoke another draw.io model as a fully independent sub-workflow. The parent workflow passes its current context to the child, waits for the child to

© IBM Corporation 2026

complete, and receives the child's resulting context back before continuing.

- **Decision** (mxgraph.flowchart.decision) nodes route execution based on the current value of a context variable. Each outgoing edge carries a label that is matched against the context at runtime.

- **Extract** (mxgraph.flowchart.extract_or_measurement) and **Merge** (mxgraph.flowchart.merge_or_storage) nodes split a single flow into parallel branches and later reconverge them. By default, all branches run as concurrent threads within the same request and the workflow continues only after all have converged at the merge node. When the engine is configured with a public endpoint (ENGINE_URL), each branch is instead dispatched as an independent HTTP child request. The parent enters a wait state and resumes automatically once all branch callbacks are received, making parallel execution resilient to platform execution time limits.

- **Manual Input** (manualinput) nodes pause the workflow and wait for an external callback before continuing.

- Delay (mxgraph.flowchart.delay) nodes pause execution for a configured amount of time, either by holding the request open (blocking mode, wait_mode: blocking) or by returning immediately and waiting for an external resume call (scheduled mode, wait_mode: scheduled).

Node behaviour is configured through draw.io's custom properties panel, no separate configuration files are needed. The completed model is saved as a [.drawio] file and placed in the engine's model directory or in an object storage bucket.

3.2 Engine Execution

The engine is a single Python application that can run as a command-line tool or as an HTTP server. In server mode it exposes a small REST API.

To understand how the **command-line** execution can be made just run the engine without arguments;

```
> python3 main.py

usage: main.py [-h] [--id INSTANCE_ID] [--context CONTEXT] [--max-call-depth MAX_CALL_DEPTH] [--simulate-manual-input]
               [--process-delay-seconds PROCESS_DELAY_SECONDS] [--server] [--host HOST] [--port PORT]
               [model]
main.py: error: the following arguments are required: model
```


© IBM Corporation 2026

The **primary endpoint** is POST /workflows/start. A caller sends the model name, an optional instance ID, and an optional initial context. The engine decides autonomously what to do: if the instance does not yet exist it starts a new execution; if the instance exists and is paused it resumes from where it left off; if it is already completed it returns the current status. The response always tells the caller the outcome, whether the workflow started fresh, was resumed, is still running, waiting, or has completed. A caller can use this single endpoint unconditionally, without tracking whether a process has been seen before. Below is a sample REST call:

```
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" \
  -d
'{"model":"mymodel","instance_id":"0110","context":{"decision":"m"}}'
```

If API authentication is enabled, add -u username:password to the curl command. Every endpoint except GET /health requires credentials when authentication is configured. A GET /metrics endpoint returns runtime counters for the current process: total requests handled, completed, waiting, errored, and currently in-flight instance IDs with their start timestamps. Like /health, it is always open and requires no authentication.

The response is a compact JSON object with three fields: status, mode, and instance_id. Note that if you don't specify the instance_id, one will be generated by the engine. The status field is completed if the workflow finished within the same request, waiting if the instance has paused and is expecting a future event, or accepted if it is still processing. The mode field reflects which path the engine took; start for a new instance, resume if an existing paused instance was continued.

Three **additional endpoints** cover specific scenarios. POST /workflows/<id>/continue and POST /workflows/<id>/callback both advance a paused instance; they are semantically distinct, continue is intended for explicit manual advancement (e.g. after a form submission), while callback is intended for asynchronous event delivery (e.g. a downstream system signalling completion), but both accept the same JSON body with model name and optional context, and both return the same compact response. GET /workflows/<id> retrieves the full persisted state of any instance including its complete audit history. Note that this endpoint requires the model name as a query parameter: GET /workflows/0110?model=mymodel. A separate GET /health endpoint is always open (no authentication required) and returns a simple status and timestamp, intended for load balancer and Kubernetes readiness probes.

When a workflow is started, the engine loads the draw.io model, parses it, and begins walking the graph node by node. It maintains an execution state for each instance containing the workflow context (a key-value store that accumulates data as execution progresses), a history of every event, the current position of each active branch, and any active wait conditions.

If a wait state is reached, a manual input node, an async delay, or a callback dependency, the engine saves the full state to Cloudant or to a local file and returns immediately to the caller. The

© IBM Corporation 2026

instance picks up exactly where it left off when the next call arrives.

By default, parallel branches created by an extract node run as concurrent threads within the same request, and the engine waits for all branches to converge at the merge node before returning. When ENGINE_URL is configured, the engine instead dispatches each branch as an independent HTTP child request. The parent persists a wait state and returns HTTP 201 immediately; execution resumes automatically when all branch callbacks arrive. This makes parallel workflows resilient to platform-level request time limits.

3.3 How the Parts Work Together

A typical **end-to-end flow** looks like this:

A business process is drawn in draw.io and saved as a [.drawio] file. The model is made available to the engine, either placed in the local filesystem alongside the application, or uploaded to an object storage bucket.

A calling application starts the workflow by posting to /workflows/start with the model name and an optional initial context. The engine assigns a unique instance ID, loads the model, and begins execution. Simple workflows, those with no waits, no parallel branches, and no long-running steps, complete within the same request and return HTTP 200 with status [completed].

If the workflow reaches a wait state, the engine returns HTTP 201 with status waiting and the instance ID. The caller stores the instance ID. When the expected external event occurs (a form is submitted, a downstream system finishes, a timer fires), the caller delivers it via /callback or /continue. The engine reloads the saved state, validates that the wait condition is now satisfied, and resumes execution from the same point.

Sub-processes follow the same pattern: the parent workflow calls a child model, which executes as its own independent workflow instance. When the child completes, its final context is merged back into the parent and the parent continues forward. If the child pauses at a wait state (for example, at a manual input node), the parent also pauses and records a predefined_process_child wait state. The WAIT STATES output shows the child's instance ID directly in the details field — no log searching needed. Once the child completes, re-running the parent instance merges the child's final context and resumes execution.

Throughout execution the engine persists state after every node, so the full history is always available — in Cloudant when configured as the state backend, or in local JSON files when running with the file backend.

Known Limitations and suggested **Mitigation** for each of them are listed below;

© IBM Corporation 2026

1. Parallel branch execution; default thread mode has timeout fragility
In the default configuration, all parallel branches run as threads within a single HTTP request. If the platform terminates the request before all branches converge at the merge node (for example, due to Code Engine's 300-second execution limit), the in-memory merge barrier is lost and the instance will remain stuck.
Mitigation 1 (recommended for serverless deployments): Set ENGINE_URL to the engine's public endpoint. The extract node will dispatch each branch as an independent HTTP child request instead of a thread. The merge barrier is persisted to Cloudant before any branch is dispatched. A platform timeout on any single branch does not destroy the barrier; the remaining branches can still call back and the parent resumes normally. If a branch is abandoned (crash, not just timeout), the parent will wait until an operator re-triggers it via /continue.
Mitigation 2 (thread mode only): Design long-running logic as predefined sub-processes. Each child model call executes as its own independent request, resetting the platform timer.
On VMs and Kubernetes with long-lived pods, neither mitigation is necessary under normal operating conditions.

2. Scheduled (async) wait mode has no built-in self-waking mechanism
When a node is configured with wait_mode: scheduled — whether a Delay node or a scheduled REST Process node — the engine returns immediately and records the wait in the instance state. It does not schedule a timer or any internal callback to resume execution at the appropriate time. The workflow will remain paused indefinitely until the engine receives an explicit resume call.
Mitigation (recommended): For time-based delays, prefer wait_mode: blocking wherever platform timeout constraints allow — the engine holds the connection open and resumes automatically. For scheduled delays that cannot use blocking mode, implement an external polling or scheduling mechanism (a cron job, an IBM Code Engine periodic trigger, or a scheduler task) that re-invokes the engine via /workflows/start with the correct instance_id once the expected wait duration has elapsed. For manual input and scheduled REST nodes, ensure that the downstream system or operator that fills in the wait file or sends the completion signal is clearly defined in your operational runbook.

3. File-based state backend — single replica only
When STATE_BACKEND is local (the default), all workflow state is stored as .json files on the local filesystem (process_state_<id>.json, event_state_<id>_<node>.json, manual_in-

© IBM Corporation 2026

put_<id>_<node>.json). This works correctly when there is exactly one replica of the engine. In a multi-replica or horizontally scaled deployment, each pod would maintain its own separate filesystem; a resume request routed to a different replica than the one that started the workflow would find no state file and fail.
Mitigation (recommended): Set STATE_BACKEND=cloudant and configure the Cloudant credentials in CREDENTIALS.txt or as environment variables. IBM Cloudant provides a distributed, replicated document store; any replica can read and write the same state records. This is the recommended backend for all production deployments running more than one instance. For single-replica containerised deployments (e.g., a single Code Engine instance), local file state is sufficient provided you use a persistent volume mount so state survives pod restarts.

4. Authentication — HTTP Basic Auth only
The engine's built-in access control is limited to HTTP Basic Authentication, configured via API_USERNAME and API_PASSWORD in CREDENTIALS.txt or as environment variables. If neither variable is set, all endpoints are unauthenticated. More advanced authentication schemes — OAuth 2.0, JWT bearer tokens, HMAC-signed API keys, or mutual TLS — are not implemented within the engine itself.
Mitigation (recommended): For production deployments requiring stronger authentication, place the engine behind a reverse proxy or API gateway (such as IBM API Connect, nginx, or an OpenShift Route with OAuth proxy) that handles authentication externally. The engine then only needs to trust the proxy layer rather than implementing complex auth logic itself. Always terminate TLS at the ingress layer; the engine does not serve HTTPS natively.

5. Configurable hard limits
Three internal hard limits apply by default to protect against runaway workflows and oversized payloads:

- **Nesting depth:** Sub-model calls (predefined_process nodes) are capped at **5 levels** deep. A workflow that attempts to exceed this depth will raise a RuntimeError rather than running indefinitely.
- **History entries:** Each instance accumulates a history log. The log is capped at **500 entries** per instance to prevent state files from growing without bound on long-running or complex workflows.
- **Request body size:** Incoming HTTP requests to the server are rejected with HTTP 413 if their body exceeds **1 MB**. This limits the size of the context object that can be passed in a single call.

© IBM Corporation 2026

Mitigation: All three limits are adjustable at startup via CLI flags. Use --max-call-depth <N> to increase the nesting cap, --max-history <N> to raise the history limit, and --max-content-length <bytes> to increase the payload cap. Raise these values explicitly for use cases that require deeper sub-workflow chains, high-volume history logging, or large context payloads. Note that increasing the history cap and payload size will increase memory and storage usage proportionally.

6. Scheduled REST nodes — design for idempotency
Two related behaviours apply when Process nodes are used in wait_mode: scheduled REST mode:
One-at-a-time firing across parallel branches. When multiple parallel branches each contain a scheduled REST node, all pauses are recorded simultaneously in the instance state at the point of the split. However, on each subsequent --id resume call (or /start HTTP call with the same instance ID), the engine fires exactly **one** scheduled REST call, applies its response to the context, and defers the remaining waits to the next invocation. A workflow with k scheduled REST nodes across its branches will therefore require k resume invocations after the initial run to reach completion. The firing order follows the position of the wait in the wait list (first-registered, first-fired).
Concurrent-resume double-fire. If two resume calls arrive simultaneously — for example due to client retry logic, a network timeout that causes a duplicate send, or an operator triggering resume twice in rapid succession — and the first REST response has not yet been written to the event state file, the external endpoint may be called more than once within that window.
Mitigation (recommended): Avoid placing scheduled REST nodes on sibling parallel branches when their outputs must influence each other or feed into a shared downstream node. Place those nodes sequentially — either before the split or after the merge — so each fires in a deterministic order with the full accumulated context available. For REST endpoints called from scheduled nodes, always design them to be **idempotent**: receiving the same request payload twice should produce no additional side effects beyond those of the first call. Standard practices include checking for an existing record before creating one, using unique request identifiers, and returning a success response on duplicate calls without re-processing.

© IBM Corporation 2026

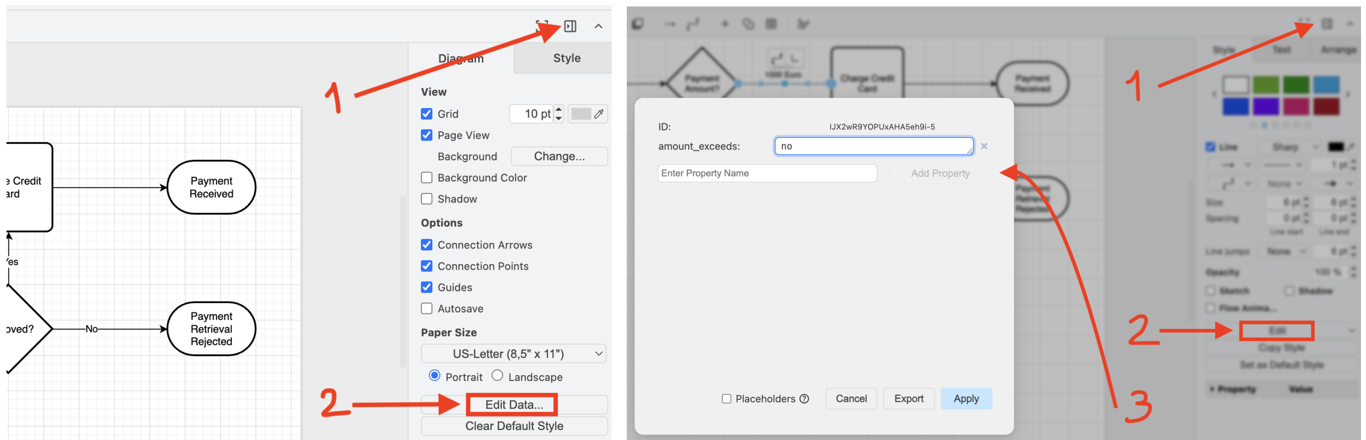
4 Hands-on Tutorial

This chapter explains step by step how to build executable workflows with draw.io and run them with the engine. The examples in the following sub-chapters are intended to show not only the supported modelling elements, but also the general way in which a workflow is prepared for execution.

Before looking at the sample processes, it is important to understand how configuration is stored inside a draw.io diagram. The engine does not use a separate workflow definition file. Instead, it reads its configuration directly from the model. For that reason, custom properties are a central part of the modelling approach.

Custom properties can be defined at two different levels. They can be attached to the draw.io model itself, or to individual model elements such as nodes and connectors. Model-level properties are useful for values that should be available globally within the workflow. Element-level properties are used to define the behaviour of a specific workflow element, for example a decision path, a REST call, a wait mode, or a fixed value that should be merged into the workflow context.

Diagram 3 shows both ways of adding such properties in draw.io. To add custom properties to the model itself, click on an empty area of the diagram and open "**Edit Data... "**. To add custom properties to a specific workflow element, first select that element, then click on **Edit** and then open "**Edit Data... "**, and finally use the **Add Property** button to create the required key-value pair and **Apply** button to save. This mechanism is used throughout all workflow examples in this chapter.



Dia. 3: Adding Custom Properties to draw.io model and elements

The examples in the following sub-chapters increase gradually in complexity. The first two focus on smaller business processes that are easy to understand visually, while the final example combines all supported workflow elements in a single model. This structure allows the reader to first become familiar with the basic modelling approach and then move on to more advanced execution patterns such as branching, waiting,

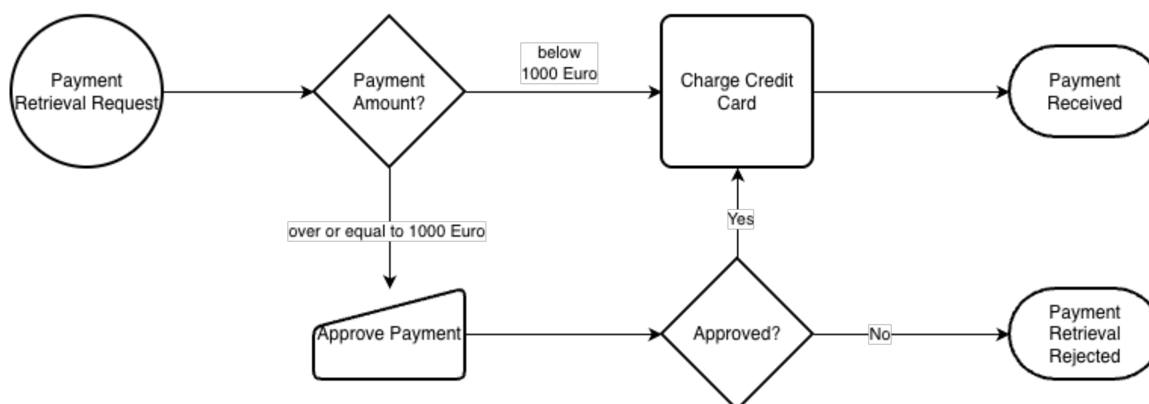
© IBM Corporation 2026

parallel execution, and sub-process calls.

Throughout these examples you will need only main.py, CREDENTIALS.txt .

4.1 Building a Sample "Payment" Workflow

After creating the model as shown in Dia. 4 using draw.io and saving it in root folder of main.py file, you will need to add some custom properties.



Dia. 4: Sample payment process drawn with draw.io

First we will run the engine as a command-line application, using no cloud resources. For this reason you need to make in `CREDENTIALS.txt` following change; `MODEL_BACKEND: local` and in `payment.drawio` model you will need to change the value `STATE_BACKEND: local` (if you don't have that value in your model yet, the engine will assume that is has the value local, so you can technically skip creating that value at this time). Also make sure that `ENGINE_URL` is either commented out or has no value when running the engine as a CLI tool.

Next you will need to select outgoing connectors from the decision "Payment Amount?". In the connector with label "below 1000 Euro" the property `amount_exceeds: no` and the connector with label "over or equal to 1000 Euro" the property `amount_exceeds: yes` . Similarly in the outgoing connectors from decision "Approved?" respective to the labels of connectors you need to put `amount_exceeds: yes` and `amount_exceeds: no` properties and values. Lastly put in "Payment Retrieval Request" Start element the property and value `credit_card_charged: no` and in process "Charge Credit Card" the property and value `credit_card_charged: yes`.

© IBM Corporation 2026

There are 3 paths the process can follow which we will call as Case A, Case B and Case C. We will test **Case A** and **Case B** only, since Case C is too similar to Case B. We will test these cases by running the engine as

- CLI application without using Cloud resources **(1-2)**,
- server without using Cloud resources **(3-4)** and
- server with connecting to Cloud resources **(5-6)** .

1. (Case A) You are now ready to run this model by using following command;
`python3 main.py payment --context '{"amount_exceeds":"no"}'`

You should see among the logs a similar output like;

```
================================================================================
MODEL         : payment
INSTANCE ID   : 3b558d5165e9476ab2d3a07dcc522ba4e22a527712e842f5b31a353c68b7939f
STATUS        : completed
COMPLETED     : True
WAIT STATES   : 0
FINAL CONTEXT: {
  "amount_exceeds": "no",
  "credit_card_charged": "yes"
}
================================================================================
```

2. (Case B) Now, run this model by using following command;
`python3 main.py payment --context '{"amount_exceeds":"yes"}'`

You will see among the logs a similar output like;

```
================================================================================
MODEL         : payment
INSTANCE ID   : 2f5a2aab597b40c197ece3771a69d2b46301c5be79544855a9272a08a99ea9bf
STATUS        : paused
COMPLETED     : False
WAIT STATES   : 1
FINAL CONTEXT: {
  "amount_exceeds": "yes",
  "credit_card_charged": "no"
}
================================================================================
```


© IBM Corporation 2026

You should also notice a new JSON document created that has a name starting with "manual_input_". You can now change it's status value to completed, put a valid completed_at value and add into input_data `"approved":"yes"`, so that it looks similar to this;


```
{
	"doc_type": "manual_input",
	"instance_id": "c7d0934b33464283b0bc1d1ea2dd65273fb7cb66a8b4488eb35dafc76a6c94a6",
	"model_name": "payment",
	"node_id": "IJX2wR9YOPUxAHA5eh9i-6",
	"status": "completed",
	"context_snapshot": {
		"amount_exceeds": "yes",
		"credit_card_charged": "no"
	},
	"node_data": {},
	"input_data": {
		"approved": "yes"
	},
	"created_at": "2026-03-30T12:12:35.795Z",
	"completed_at": "2026-03-30T12:12:45.795Z"
}
```


And now you can proceed with running following command using the instance_id generated by the engine;

```
python3 main.py payment --id
c7d0934b33464283b0bc1d1ea2dd65273fb7cb66a8b4488eb35dafc76a6c94a6
```

As a result you will get something like this in logs, indicating the process successfully completing;

```
================================================================================

MODEL        : payment

INSTANCE ID  : c7d0934b33464283b0bc1d1ea2dd65273fb7cb66a8b4488eb35dafc76a6c94a6

STATUS       : completed

COMPLETED    : True

WAIT STATES  : 0

FINAL CONTEXT: {

  "amount_exceeds": "yes",

  "credit_card_charged": "yes",

  "approved": "yes"

}

================================================================================
```

Note that you can specify the instance_id yourself when creating a new process by using the argument `--id` and use that id in follow up calls.

© IBM Corporation 2026

Next we will run the engine as server;

```
python3 main.py --server
```

3. (Case A) We will first repeat the initial case we did earlier;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"payment","context":{"amount_exceeds":"no"}}'
```

```
{"instance_id":"c97be78920a44e91a0e10934270de2c1953bd94a103347f5a35977c8
c982b4bf","mode":"start","status":"completed"}
```

4. (Case B) And next, we will try the second case we did earlier, where manual input was necessary, For this we will send following REST Call to start a process;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"payment","context":{"amount_exceeds":"yes"}}'
```

```
{"instance_id":"0eeebc3db9e641bf8ca8a5369ef42bed8c29bd2764544a0d97538dac
f82fb843","mode":"start","status":"waiting"}
```

At this point you should edit the manual_input_* JSON document as you previously did;

```
{
	"doc_type": "manual_input",
	"instance_id": "0eeebc3db9e641bf8ca8a5369ef42bed8c29bd2764544a0d97538dacf82fb843",
	"model_name": "payment",
	"node_id": "IJX2wR9YOPUxAHA5eh9i-6",
	"status": "completed",
	"context_snapshot": {
		"amount_exceeds": "yes",
		"credit_card_charged": "no"
	},
	"node_data": {},
	"input_data": {
		"approved": "yes"
	},
	"created_at": "2026-03-30T16:59:24.884Z",
	"completed_at": "2026-03-30T16:59:27.884Z"
}

curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"payment","instance_id":"d00b46229b0946679de4235582bba54c19763
```


© IBM Corporation 2026

```
4cedcf94152ad5d8559e0644efd"}'
{"instance_id":"d00b46229b0946679de4235582bba54c197634cedcf94152ad5d8559
e0644efd","mode":"resume","status":"completed"}
```

5. (Case A) Now we will try the same functionality using Cloud Object Storage and Cloudant NoSQL database in IBM Cloud. For this case and the next case you will need to upload your draw.io model to your bucket in Cloud Object Storage.
You need to add following custom properties to your model (by clicking on an empty space and then clicking on the button "Edit Data.."); `CLOUDANT_APIKEY`, `CLOUDANT_DB`, `CLOUDANT_URL`, `STATE_BACKEND`. The value of `STATE_BACKEND` needs to be changed to cloudant, so that you have `STATE_BACKEND: cloudant` . Do not forget to create the cloudant database which you enter as value into `CLOUDANT_DB` .
You will also need to make sure in `CREDENTIALS.txt` file that `MODEL_BACKEND=cos` . You will also need to remove the comment-out for following values `COS_APIKEY`, `COS_RESOURCE_INSTANCE_ID`, `COS_ENDPOINT`, `COS_BUCKET`, `COS_MODEL_PREFIX` . Note that a value for `COS_MODEL_PREFIX` will not be necessary if you intend to save the files directly into the bucket of Cloud Object Storage. And as final adjustment you will need to enter a value for engine url - in our case localhost- `ENGINE_URL=http://127.0.0.1:8080` . Note that the ENGINE_URL's value could be either Kubernetes cluster's ingress or Code Engine's URL.

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"payment","context":{"amount_exceeds":"no"}}'
{"instance_id":"b4933f627c3247ba85dd489ddd0214e9a78fc2e1335f45d1b1021411
9cc77985","mode":"start","status":"completed"}
```

6. (Case B) We will now repeat the same REST Call as in Nr. 4;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"payment","context":{"amount_exceeds":"yes"}}'
{"instance_id":"1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb
6aab9b8d","mode":"start","status":"waiting"}
```

And we will log in to Cloudant NoSQL database and make the necessary changes as in previous test case;

```
{
"_id":
"manual_input_1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d_IJX2wR9YOPUxAHA5eh9i-
```


© IBM Corporation 2026

```
6",
 "_rev": "2-4e0e83461811a210dd9b30b7c562c656",
 "doc_type": "manual_input",
 "instance_id": "1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d",
 "model_name": "payment",
 "node_id": "IJX2wR9YOPUxAHA5eh9i-6",
 "status": "completed",
 "context_snapshot": {
  "amount_exceeds": "yes",
  "credit_card_charged": "no"
 },
 "node_data": {},
 "input_data": {
  "approved": "yes"
 },
 "created_at": "2026-03-30T20:47:41.411Z",
 "completed_at": "2026-03-30T20:47:45.411Z"
}
```


After updating the manual_input_* file we will make the follow up REST Call so that the process can continue. Note that making this call without adjusting the manual_input_* file will result in the system staying in waiting mode!

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type: application/json" -d '{"model":"payment","instance_id":"1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d"}'

{"instance_id":"1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d","mode":"resume","status":"completed"}
```

To get the status of a process with full context you can run;

```
curl "http://127.0.0.1:8080/workflows/1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d?model=payment"
```

And you will get a response that looks like this (similar to the output in Terminal when running the engine as a CLI application instead of server);

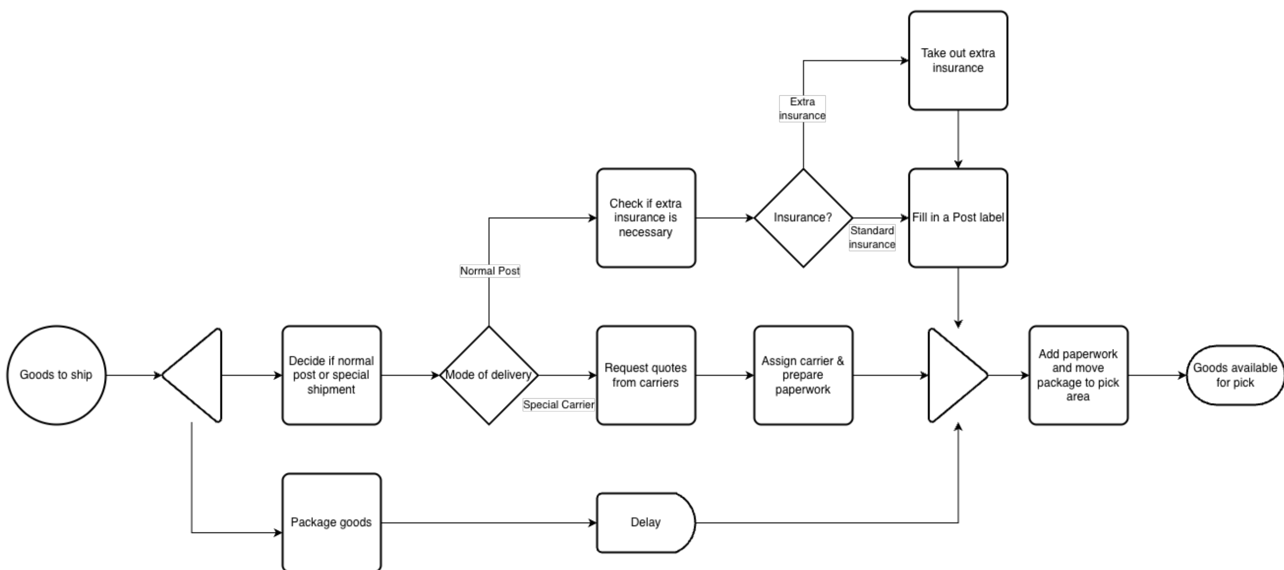

```
{
 "logs": [],
 "mode": "get",
 "state": {
 "_id": "process_state_1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d",
 "_rev": "23-28cbf0a39e1e8e5df902a3d0c0bb5ce0",
 "active_tokens": 0,
 "branch_id": "",
 "branch_parent_instance_id": "",
```


© IBM Corporation 2026

```
 "branch_stop_node": "",
 "child_call_depth": 0,
  "completed": true,
  "context": {
   "amount_exceeds": "yes",
   "approved": "yes",
   "credit_card_charged": "yes"
  },
  "current_tokens": [],
  "doc_type": "process_state",
  "history": […],
  "instance_id": "1451fc5b8c794a44ac837f13b9940f9ae3abfb7e6b664869a6a4e5cb6aab9b8d",
  "model_checksum": "2f75bfbfa76ca02d",
  "model_name": "payment",
  "saved_at": "2026-03-30T20:54:11.877Z",
  "status": "completed",
  "step_executions": {…},
  "waits": []
 },
 "status": "ok"
}
```

4.2 Building a Sample “Shipment” Workflow

After creating the model as shown in Dia. 5 using draw.io and saving it in root folder of main.py file, you will need to add some custom properties.



Dia. 5: Sample shipment process drawn with draw.io

First we will run the engine as a command-line application, using no cloud resources. For this reason you need to make in `CREDENTIALS.txt` following change; `MODEL_BACKEND: local` and in `shipment.drawio` model you will need to change the value `STATE_BACKEND: local` (if

you don't have that value in your model yet, the engine will assume that is has the value local, so you can technically skip creating that value at this time). Also make sure that `ENGINE_URL` is either commented out or has no value when running the engine as a CLI tool.

Next you will need to select outgoing connectors from the decision "Mode of delivery". In the connector with label "Normal Post" the property `delivery: normal` and the connector with label "Special Carrier" the property `delivery: special` . Similarly in the outgoing connectors from decision "Insurance?" respective to the labels of connectors you need to put `insurance: extra` and `insurance: standard` properties and values. Lastly put in "Goods to ship" Start element the property and value `goods_available: no` , in process "Package goods" the property and value `goods_packaged: yes` , in process "Take out extra insurance" the property and value `extra_insurance_applied: yes` , in process "Fill in a Post label" the property and value `post_label: yes` , in process "Request quotes from carriers" the property and value `quote_from_carrier: 9.99` , in process "Assign carrier & prepare paperwork" the property and value `carrier: fedex` . In delay element "Delay" you will need to add the properties and values `delay: 2m` and `wait_mode: blocking` . Note that `wait_mode: blocking` and `wait_mode` property missing result actually in exact same behaviour. Lastly in process "Add paperwork and move package to pick area" following properties and values;

- `goods_available: yes`
- `header: {"Content-Type":"application/json"}`
- `method: POST`
- `type: rest`
- `url: https://api.restful-api.dev/objects`
- `wait_mode: blocking`

Note that you can use both "header" and "headers" as the property name. Both are processed same way.

There are 3 paths the process can follow which we will call as Case A, Case B and Case C. We will test **Case B** only, since will concentrate us in testing Delay and -REST Call making- Process components in sync and async modes (i.e. `wait_mode: blocking` and `wait_mode: scheduled`). We will test these cases by running the engine as

- CLI application without using Cloud resources **(1-2)**,
- server without using Cloud resources **(3-4)** and
- server with connecting to Cloud resources **(5-6)** .

1. (Case A) You are now ready to run this model by using following command;

```
python3 main.py shipment --context '{"delivery":"normal",
"insurance":"extra"}'
```

You should see among the logs a similar output like;

© IBM Corporation 2026

```
================================================================================
MODEL         : shipment
INSTANCE ID   : 4a3558939e1b4f959610e66fea9baf6d535c04e10f7044b984b954295503bfe4
STATUS        : completed
COMPLETED     : True
WAIT STATES   : 0
FINAL CONTEXT: {
  "delivery": "normal",
  "insurance": "extra",
  "goods_available": "yes",
  "goods_packaged": "yes",
  "extra_insurance_applied": "yes",
  "post_label": "yes",
  "response_by": "test_api"
}
================================================================================
```

2. (Case B) Now, in process "Add paperwork and move package to pick area", change the value of the wait_mode property to scheduled:
wait_mode: scheduled

This puts the REST call into asynchronous mode: the engine will fire the REST request on a subsequent resume call rather than waiting for it synchronously. Run the model with the following command, notice that a context value is passed to set goods_available to "no" so the REST process node is reached:
python3 main.py shipment --context
'{"delivery":"normal","insurance":"extra","goods_available":"no"}'

You should see among the logs a similar output like;

```
================================================================================
MODEL         : shipment
INSTANCE ID   : e104fd3dd05d4ca8978fa5d9b7a8e16b60f4420bf111461091dd984e809a4280
STATUS        : paused
COMPLETED     : False
WAIT STATES   : 1
FINAL CONTEXT: {
  "delivery": "normal",
  "insurance": "extra",
  "goods_available": "no",
  "goods_packaged": "yes",
  "extra_insurance_applied": "yes",
```


© IBM Corporation 2026

```
  "post_label": "yes"
}
================================================================================
```

The engine returned STATUS: paused because the scheduled REST call has not yet fired. Resume the instance with the following command — this time the engine will execute the REST call, receive the response, merge it into the context, and complete:

```
python3 main.py shipment --id
e104fd3dd05d4ca8978fa5d9b7a8e16b60f4420bf111461091dd984e809a4280
```

And you will notice the process resuming until it completed;

```
================================================================================
MODEL         : shipment
INSTANCE ID   : e104fd3dd05d4ca8978fa5d9b7a8e16b60f4420bf111461091dd984e809a4280
STATUS        : completed
COMPLETED     : True
WAIT STATES   : 0
FINAL CONTEXT: {
  "delivery": "normal",
  "insurance": "extra",
  "goods_available": "yes",
  "goods_packaged": "yes",
  "extra_insurance_applied": "yes",
  "post_label": "yes",
  "response_by": "test_api"
}
================================================================================
```

3. (Case A) We will first repeat the initial case we did earlier;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"shipment","context":{"delivery":"normal",
"insurance":"extra"}}'

{"instance_id":"1fd3ec308cf84f2e85002edcd56fa43da57e47c74ba34cbbb9d2e7d6
ec2d30a3","mode":"start","status":"completed"}
```

4. (Case B) And next we will repeat the second case we did earlier with same changes;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
```


© IBM Corporation 2026

```
'{"model":"shipment","context":{"delivery":"normal",
"insurance":"extra"}}'
```

```
{"instance_id":"5502a2bc1d5045238e4734b49276cb4dba65b43435ac4cf89e44e1e9
29bedc0f","mode":"start","status":"waiting"}
```

And the follow up call with the instance id generated by the engine. Note that you can use your own instance id, but you need to provide it when starting a new workflow and make sure that you don't use the same id twice.

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"shipment","instance_id":"5502a2bc1d5045238e4734b49276cb4dba65
b43435ac4cf89e44e1e929bedc0f"}'
```

```
{"instance_id":"5502a2bc1d5045238e4734b49276cb4dba65b43435ac4cf89e44e1e9
29bedc0f","mode":"resume","status":"completed"}
```

5. (Case A) We will repeat the initial case we did earlier but this time by changing the `MODEL_BACKEND` and `STATE_BACKEND` values to `cos` and `cloudant` respectively;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"shipment","context":{"delivery":"normal",
"insurance":"extra"}}'
```

```
{"instance_id":"f12c90a05dbd44f59eb98840648eacec9a026f0ca29a4c6b8e69e97b
0cbcc93b","mode":"start","status":"completed"}
```

6. (Case B) We will repeat the second case we did earlier but this time by changing the `MODEL_BACKEND` and `STATE_BACKEND` values to `cos` and `cloudant` respectively;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"shipment","context":{"delivery":"normal",
"insurance":"extra"}}'
```

```
{"instance_id":"e5610321e4604e92b54a66368d98ac90ba48de453e7347b0956bf0c5
f3162649","mode":"start","status":"waiting"}
```

And the follow up call with the instance id generated by the engine;

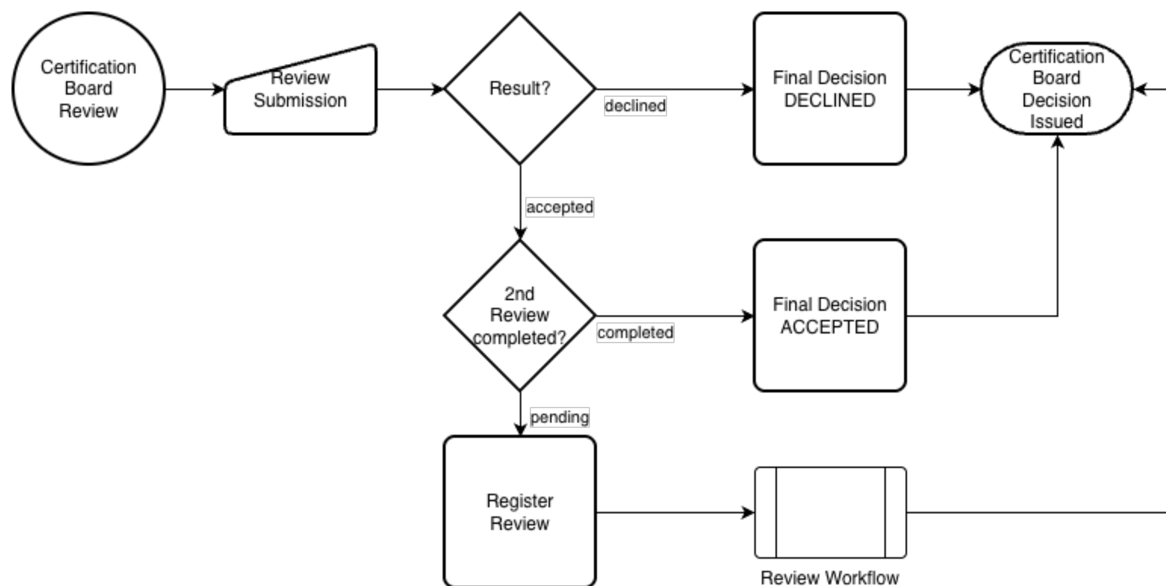
```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"shipment","instance_id":"e5610321e4604e92b54a66368d98ac90ba48
de453e7347b0956bf0c5f3162649"}'
```

```
{"instance_id":"e5610321e4604e92b54a66368d98ac90ba48de453e7347b0956bf0c5
f3162649","mode":"resume","status":"completed"}
```


© IBM Corporation 2026

4.3 Building a Sample "Review" Workflow

After creating the model as shown in Dia. 6 using draw.io and saving it in root folder of main.py file, you will need to add some custom properties.



Dia. 6: Sample board evaluation process drawn with draw.io

First we will run the engine as a command-line application, using no cloud resources. For this reason you need to make in CREDENTIALS.txt following change; MODEL_BACKEND: local and in review.drawio model you will need to change the value STATE_BACKEND: local (if you don't have that value in your model yet, the engine will assume that is has the value local, so you can technically skip creating that value at this time). Also make sure that ENGINE_URL is either commented out or has no value when running the engine as a CLI tool.

Next you will need to select outgoing connectors from the decision "Result?". In the connector with label "declined" the property `result: declined` and the connector with label "accepted" the property `result: accepted` . Similarly in the outgoing connectors from decision "2nd Review completed?" respective to the labels of connectors you need to put `second_review: completed` and `second_review: pending` properties and values. Lastly put in process "Register Review" the property and value `second_review:completed`, in process "Final Decision DECLINED" the property and value `final_decision: declined`, in process "Final Decision ACCEPTED" the property and value `final_decision: accepted`, in predefined process "Request quotes from carriers" the property and value `quote_from_carrier: 9.99`, in process "Review Workdflow" the property and value `model: review`.

© IBM Corporation 2026

There are 3 paths the process can follow which we will call as Case A, Case B and Case C. Due to the Predefined Process we will have a recursivity that lets the workflow terminate only using **Case A** or **Case B**. We will test **Case B** and the Predefined Process in `wait_mode: blocking` and `wait_mode: scheduled` by running the engine as

- CLI application without using Cloud resources **(1)**,
- server without using Cloud resources **(2-3)** and
- server with connecting to Cloud resources **(4-5)** .

1. (Case B) You are now ready to run this model by using following command;
`python3 main.py review`

You should see among the logs a similar output like;

```
================================================================================

MODEL         : review
INSTANCE ID   : 9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121
STATUS        : paused
COMPLETED     : False
WAIT STATES   : 1
FINAL CONTEXT: {
  "second_review": "pending"
}

================================================================================
```

You need now to edit the manual_input_* file with this instance id in its name as follows;

```
{
 "doc_type": "manual_input",
 "instance_id": "9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121",
 "model_name": "review",
 "node_id": "dUXyPl7EW0r6KnBbvOKN-29",
 "status": "completed",
 "context_snapshot": {},
 "node_data": {},
 "input_data": {
 "result": "accepted"
 },
 "created_at": "2026-04-02T15:03:57.578Z",
 "completed_at": "2026-04-02T15:03:58.578Z"
}
```

And resume the worklow with

```
python3 main.py review --id
9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121
```

And you will receive a similar output like below;

```
================================================================================

MODEL         : review
INSTANCE ID   : 9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121
```


© IBM Corporation 2026

```
STATUS       : paused
COMPLETED    : False
WAIT STATES  : 1
FINAL CONTEXT: {
  "second_review": "completed",
  "result": "accepted"
}

================================================================================
```

The child instance ID is shown directly in the WAIT STATES section above (see the details.child_instance_id field) — no log searching needed. Note the child process has already been started and paused by the parent process. Edit the manual_input file of the child process and resume it by running;

```
python3 main.py review --id
c2842ac978fd43d282f51f0222205e1f659e0bee78d9431fba4d966b13ed12d1
```

And you will receive a similar output like below;

```
================================================================================

MODEL        : review
INSTANCE ID  : c2842ac978fd43d282f51f0222205e1f659e0bee78d9431fba4d966b13ed12d1
STATUS       : completed
COMPLETED    : True
WAIT STATES  : 0
FINAL CONTEXT: {
  "second_review": "completed",
  "result": "accepted",
  "final_decision": "accepted"
}

================================================================================
```

Now resume the parent workflow;

```
python3 main.py review --id
9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121
```

And you will receive a similar output like below;

```
================================================================================

MODEL        : review
INSTANCE ID  : 9220e34236af4c1eb0c74b0c4dfe6cce19d6d07bf27e455c800b6ddc19791121
STATUS       : completed
COMPLETED    : True
WAIT STATES  : 0
FINAL CONTEXT: {
  "second_review": "completed",
  "result": "accepted",
  "final_decision": "accepted"
```


© IBM Corporation 2026

```
}
================================================================================
```

This output shows that the workflow is completed successfully with the results listed in final context.

2. (Case B) This time we will not make any changes in the model. We will only start the application as a server;

```
python3 main.py --server
```

This starts a new workflow instance. Because the review model contains a Manual Input node, the engine immediately pauses and returns HTTP 201 with status "waiting". The instance ID in the response must be stored — it is used for all subsequent resume and GET calls:

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type: application/json" -d '{"model":"review"}'

{"instance_id":"8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a033717980503c8","mode":"start","status":"waiting"}
```

Edit the manual_input_* JSON file for this instance (set status to "completed" and add the reviewer decision to input_data, e.g. approved: "yes"), then resume the parent instance by sending the same endpoint with the instance_id — this causes the engine to continue from the manual input node:

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type: application/json" -d '{"model":"review", "instance_id":"8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a033717980503c8"}'

{"instance_id":"8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a033717980503c8","mode":"resume","status":"waiting"}
```

To find the child instance ID, call: GET http://127.0.0.1:8080/workflows/8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a033717980503c8?model=review and look for the waits array in the JSON response. The predefined_process_child entry contains the child_instance_id in its details field.

Edit the manual_input file of child workflow and proceed;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type: application/json" -d '{"model":"review", "instance_id":"b8f258698c1f4033870286acf8614d58affa524720d84570bdc84fc77753551f"}'

{"instance_id":"b8f258698c1f4033870286acf8614d58affa524720d84570bdc84fc77753551f","mode":"resume","status":"waiting"}
```


© IBM Corporation 2026

After editing the child manual_input file, send a second resume call for the child. This time the child has no remaining wait states and completes:

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d '{"model":"review",
"instance_id":"b8f258698c1f4033870286acf8614d58affa524720d84570bdc84fc77
753551f"}'

{"instance_id":"b8f258698c1f4033870286acf8614d58affa524720d84570bdc84fc7
7753551f","mode":"resume","status":"completed"}
```

The child sub-process is now completed. Resume the parent workflow — the engine detects that the child finished, merges the child context back into the parent, and continues execution to completion:

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d '{"model":"review",
"instance_id":"8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a0337179
80503c8"}'

{"instance_id":"8da57f631d7045868f2391a371e2dab5a9c6d338b53142e49a033717
980503c8","mode":"resume","status":"completed"}
```

Now you can verify the completed state using the GET call mentioned in sub-chapter 4.1. If you need to find the child workflow instance ID during execution (without checking logs), call GET /workflows/<parent_id>?model=review. The response's waits array will contain a predefined_process_child entry whose details.child_instance_id field holds the child's instance ID directly.

a) Alternatively, if using Cloudant as the state backend, you can listen to database changes to detect new child instances as they are created.

b) Or you can run another application in-loop and check the database changes in a time interval.

3. (Case C — REST sub-process) Instead of using a predefined sub-process model, configure the Predefined Process node as a REST call. Make the following changes to the Predefined Process node properties in the review model:

- `headers: {"Content-Type":"application/json"}`
- `method: POST`
- `type: rest`
- `url: https://api.restful-api.dev/objects`
- `wait_mode: blocking`

© IBM Corporation 2026

Note that you can use both “header” and “headers”. Both will be processed same way.

And start the engine and make the first REST Call as shown below;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d '{"model":"review"}'

{"instance_id":"d098a962062b460e8bdb15049c4a525c1afe1af1f2594df1bddf188a
7fbfc001","mode":"start","status":"waiting"}
```

Change manual_input file and proceed;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"d098a962062b460e8bdb15049c4a525c1afe1a
f1f2594df1bddf188a7fbfc001"}'

{"instance_id":"d098a962062b460e8bdb15049c4a525c1afe1af1f2594df1bddf188a
7fbfc001","mode":"resume","status":"waiting"}
```

Change manual_input file and proceed with child workflow;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"3f89e083116a4d729e2d15291511bc8eae67ed
4edf904adb9534ee678976cdf3"}'

{"instance_id":"3f89e083116a4d729e2d15291511bc8eae67ed4edf904adb9534ee67
8976cdf3","mode":"resume","status":"completed"}
```

And resume mother workflow;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"d098a962062b460e8bdb15049c4a525c1afe1a
f1f2594df1bddf188a7fbfc001"}'

{"instance_id":"d098a962062b460e8bdb15049c4a525c1afe1af1f2594df1bddf188a
7fbfc001","mode":"resume","status":"completed"}
```

4. (Case C — Cloud resources) Repeat Step 2 (sub-process variant) but this time using Cloud Object Storage and Cloudant. Make the following changes: set STATE_BACKEND: cloudant in the model properties and MODEL_BACKEND=cos in CREDENTIALS.txt. Then start the server:

Start the application as a server;

© IBM Corporation 2026

```
python3 main.py --server
```

And we will send following request;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d '{"model":"review"}'

{"instance_id":"0990ed157ec54be5807e3047e8dfc24658ed3f63064743708f2b47ff
5266fe36","mode":"start","status":"waiting"}
```

Edit the manual_input file and proceed;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"0990ed157ec54be5807e3047e8dfc24658ed3f
63064743708f2b47ff5266fe36"}'

{"instance_id":"0990ed157ec54be5807e3047e8dfc24658ed3f63064743708f2b47ff
5266fe36","mode":"resume","status":"waiting"}
```

Edit the manual_input file of child workflow and proceed;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"12abe1deb3e54535b674b78e41b9d9a3b60818
a7d4e846d488f69bc84e59deaf"}'

{"instance_id":"12abe1deb3e54535b674b78e41b9d9a3b60818a7d4e846d488f69bc8
4e59deaf","mode":"resume","status":"completed"}
```

Resume the mother workflow;

```
curl -X POST http://127.0.0.1:8080/workflows/start -H "Content-Type:
application/json" -d
'{"model":"review","instance_id":"0990ed157ec54be5807e3047e8dfc24658ed3f
63064743708f2b47ff5266fe36"}'

{"instance_id":"0990ed157ec54be5807e3047e8dfc24658ed3f63064743708f2b47ff
5266fe36","mode":"resume","status":"completed"}
```

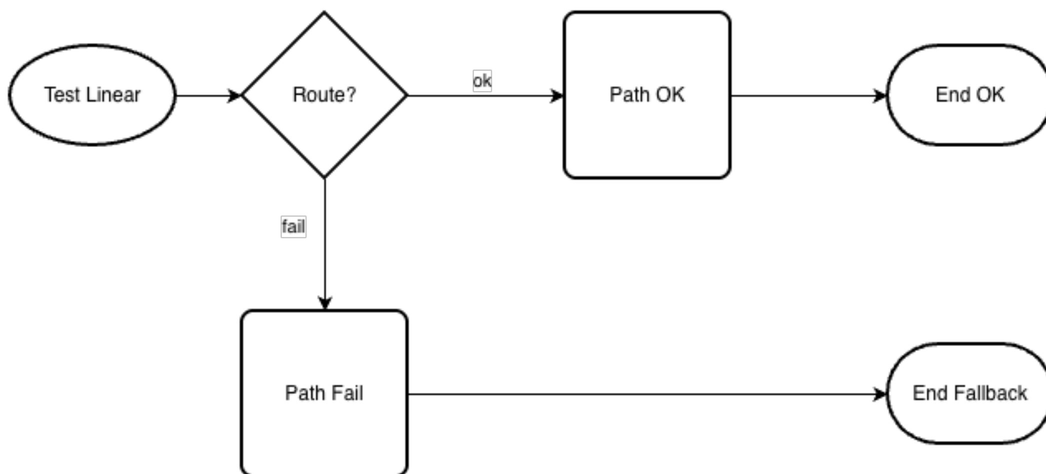

© IBM Corporation 2026

5 Test Model Walkthroughs

This chapter walks through each of the ten canonical test models included with the engine. Together they cover every node type, wait mode, and execution pattern the engine supports. For each model you will find a diagram, a brief description of what the model tests, the required node properties, and step-by-step CLI and REST execution examples with expected output.

5.1 W1 — Linear Flow with Decision (test_linear)

The simplest possible model: a Start node, a Process node that writes a static context value, a Decision node with two outgoing connectors (result=ok and result=fail), and two separate End nodes. This model is the primary vehicle for testing decision routing, context merging, and idempotency. No delays, no REST calls, no waits — execution is always synchronous.



Dia. 7: W1 — test_linear, linear flow with decision

Required custom properties:

```
Connector from Decision (label "ok"):    result: ok
Connector from Decision (label "fail"):  result: fail
Process node "Set processed":            processed: true
Process node "Handle failure":           handled: true
```

CLI execution — Case A (ok path): pass result=ok in the initial context. The engine follows the ok connector, merges processed=true, and completes:

```
python3 main.py test_linear --context '{"result":"ok"}'

==============================================================================
MODEL         : test_linear
STATUS        : completed    |    WAIT STATES : 0
FINAL CONTEXT: { "result": "ok", "processed": "true" }
```


© IBM Corporation 2026

CLI execution — Case B (fail path): pass result=fail. The fail connector is followed and handled=true is merged:

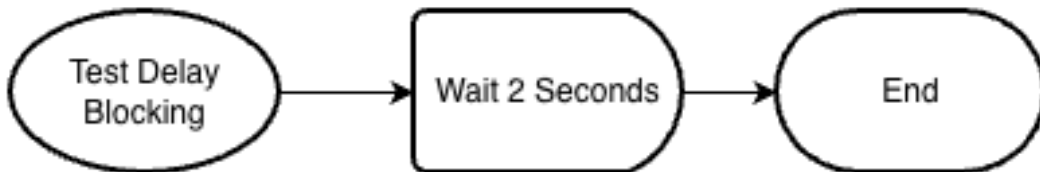
```
python3 main.py test_linear --context '{"result":"fail"}'

==============================================================================
MODEL         : test_linear
STATUS        : completed    |    WAIT STATES : 0
FINAL CONTEXT: { "result": "fail", "handled": "true" }
```

REST execution — start the server, then send a start request. The 200 response and status=completed confirm the decision was resolved in a single synchronous call:

```
python3 main.py --server

curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" \
  -d '{"model":"test_linear","context":{"result":"ok"}}'

{"instance_id":"<id>","mode":"start","status":"completed"}
```

5.2 W2 — Blocking Delay (test_delay_blocking)

A Delay node configured with wait_mode: blocking. The engine holds the connection open for the specified duration, then continues and completes. No resume call is needed — execution is synchronous end-to-end. Preferred for short delays on platforms with generous request timeouts.



Dia. 8: W2 — test_delay_blocking, synchronous blocking delay

Required custom properties on the Delay node:

```
delay:      2
wait_mode: blocking
```

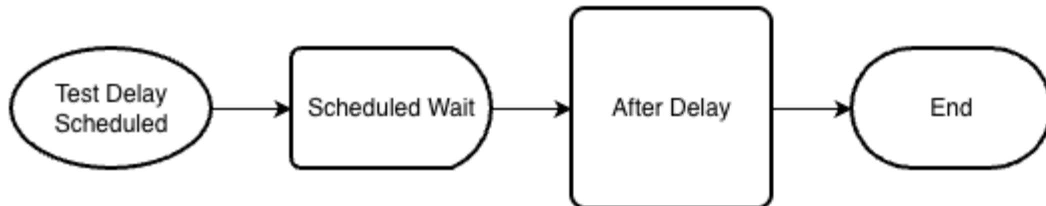
CLI execution: the engine sleeps for 2 seconds inline, writes delay_passed=true to context, and completes. No --id resume is required:

```
python3 main.py test_delay_blocking

==============================================================================
MODEL         : test_delay_blocking
STATUS        : completed    |    WAIT STATES : 0
FINAL CONTEXT: { "delay_passed": "true" }
```


© IBM Corporation 2026

5.3 W3 — Scheduled Delay (test_delay_scheduled)

The same topology as W2 but with wait_mode: scheduled. The engine records a wake_at timestamp and returns immediately with status=paused. An external trigger must resume the instance once the scheduled time has elapsed. If resume is called before wake_at, the engine returns paused again — this behaviour is tested explicitly by the test suite.



Dia. 9: W3 — test_delay_scheduled, asynchronous scheduled delay

Required custom properties on the Delay node:

```
delay:     2
wait_mode: scheduled
```

CLI execution — Run 1: records the wake_at timestamp and pauses immediately:

```
python3 main.py test_delay_scheduled

================================================================================
MODEL         : test_delay_scheduled
INSTANCE ID   : <id>
STATUS        : paused    |   WAIT STATES : 1
FINAL CONTEXT: {}
```

Wait at least 2 seconds, then resume. The engine checks that wake_at has passed, writes delay_passed=true, and completes:

```
python3 main.py test_delay_scheduled --id <id>

================================================================================
MODEL         : test_delay_scheduled
STATUS        : completed    |   WAIT STATES : 0
FINAL CONTEXT: { "delay_passed": "true" }
```

REST server execution — two calls: start (pauses) then resume after 2 s (completes):

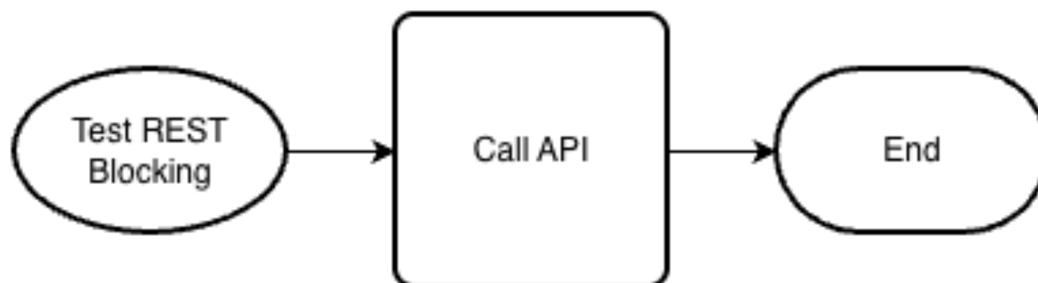
```
# Start — records scheduled wait
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" -d '{"model":"test_delay_scheduled"}'
{"instance_id":"<id>","mode":"start","status":"waiting"}

# Resume (after delay expires)
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" \
  -d '{"model":"test_delay_scheduled","instance_id":"<id>"}'
```


© IBM Corporation 2026

```
{"instance_id":"<id>","mode":"resume","status":"completed"}
```

5.4 W4 — Blocking REST Call (test_rest_blocking)

A Process node configured as type: rest with wait_mode: blocking. The engine sends the HTTP request synchronously, waits for the JSON response, merges all top-level response keys into the workflow context, and continues to the End node — all within a single CLI run or HTTP request. If the external endpoint is unreachable, the engine raises a RuntimeError.



Dia. 10: W4 — test_rest_blocking, synchronous REST integration

Required custom properties on the Process node:

```
type:      rest
method:    GET
url:       https://api.restful-api.dev/objects/7
wait_mode: blocking
```

CLI execution: the engine calls the API, receives the response, merges the JSON keys into context, and completes in a single run. The exact context keys reflect the API response at the time of execution:

```
python3 main.py test_rest_blocking

================================================================================
MODEL         : test_rest_blocking
STATUS        : completed   |   WAIT STATES : 0
FINAL CONTEXT: {
  "id": "7",
  "name": "Apple MacBook Pro 16",
  "data": { ... }
}
```

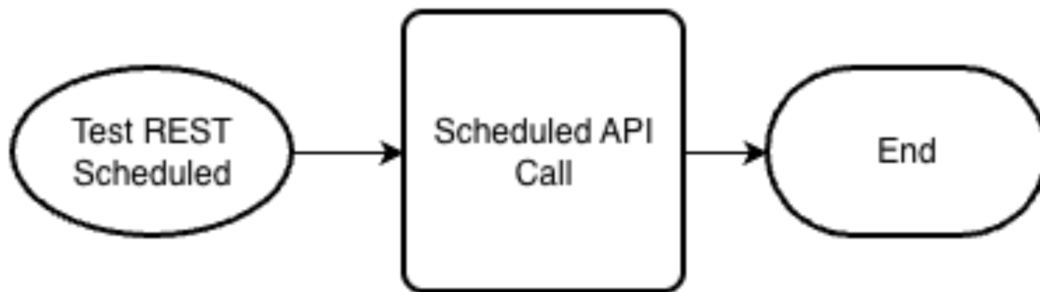
REST server execution — a single call starts and completes the workflow:

```
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" -d '{"model":"test_rest_blocking"}'

{"instance_id":"<id>","mode":"start","status":"completed"}
```


© IBM Corporation 2026

5.5 W5 — Scheduled REST Call (test_rest_scheduled)

A Process node configured as type: rest with wait_mode: scheduled. On the first run the engine records the pending REST call in the instance state and returns paused without calling the API. On the next resume call it fires the request, merges the response, and completes. This is the asynchronous variant of W4 and is subject to the idempotency requirement described in the Known Limitations chapter.



Dia. 11: W5 — test_rest_scheduled, asynchronous scheduled REST call

Required custom properties on the Process node:

```
type:      rest
method:    GET
url:       https://api.restful-api.dev/objects/7
wait_mode: scheduled
```

CLI execution — Run 1: records the REST wait and returns paused without contacting the API:

```
python3 main.py test_rest_scheduled

==============================================================================
MODEL         : test_rest_scheduled
INSTANCE ID   : <id>
STATUS        : paused    |    WAIT STATES : 1
FINAL CONTEXT: {}
```

CLI execution — Run 2 (resume): fires the REST request and completes:

```
python3 main.py test_rest_scheduled --id <id>

==============================================================================
MODEL         : test_rest_scheduled
STATUS        : completed    |    WAIT STATES : 0
FINAL CONTEXT: {
  "id": "7",
  "name": "Apple MacBook Pro 16",
  "data": { ... }
}
```

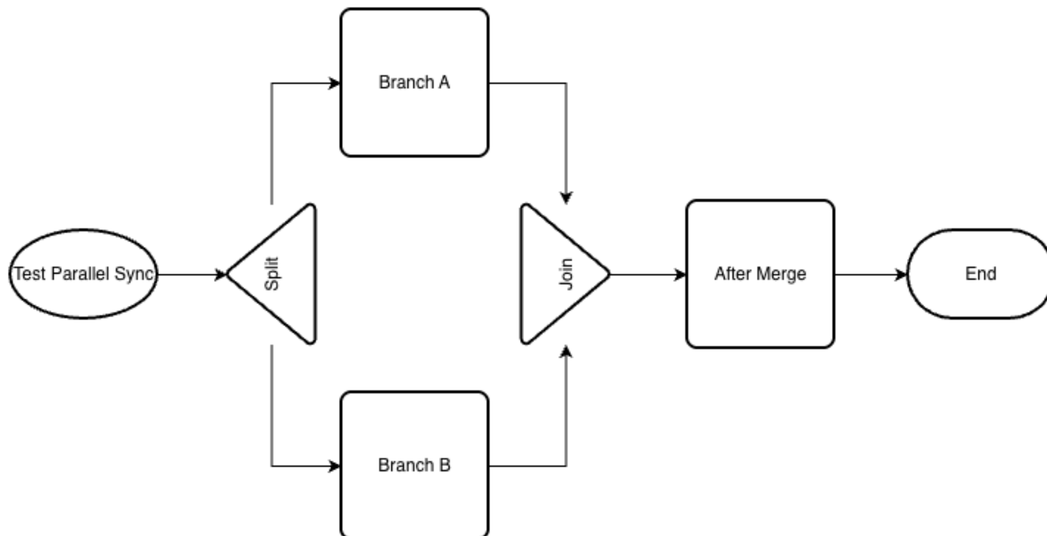
REST server execution — two calls: start (records wait) then resume (fires REST call):

© IBM Corporation 2026

```
# Start — records REST wait, returns immediately
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" -d '{"model":"test_rest_scheduled"}'
{"instance_id":"<id>","mode":"start","status":"waiting"}

# Resume — fires the REST call and completes
curl -X POST http://127.0.0.1:8080/workflows/start \
  -H "Content-Type: application/json" \
  -d '{"model":"test_rest_scheduled","instance_id":"<id>"}'
{"instance_id":"<id>","mode":"resume","status":"completed"}
```

5.6 W6 — Two-Branch Parallel Sync (test_parallel_sync)

An Extract node splits execution into two parallel branches. Each branch contains a Process node writing a distinct context key (branch_a and branch_b). A Merge node with merge_mode: sync waits for both branches to complete, joins execution back into a single path, and continues to the End node. Both keys must appear in the final context — if either is missing the merge did not work correctly.



Dia. 12: W6 — test_parallel_sync, two-branch sync merge

Required custom properties:

```
Process node (branch A):  branch_a: done
Process node (branch B):  branch_b: done
Merge node:               merge_mode: sync
```

CLI execution: both branches run concurrently as threads (or child HTTP requests when ENGINE_URL is set). The Merge node blocks until both complete:

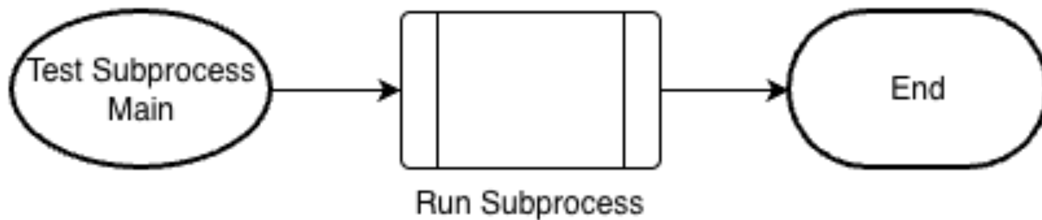
```
python3 main.py test_parallel_sync

==============================================================================
```


© IBM Corporation 2026

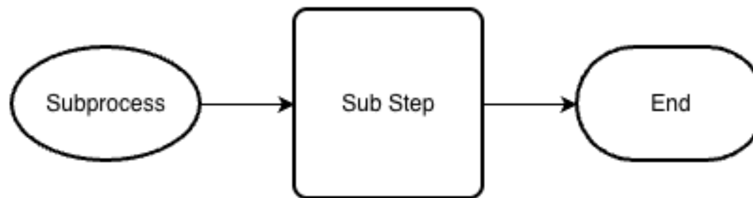
```
MODEL          : test_parallel_sync
STATUS         : completed    |    WAIT STATES : 0
FINAL CONTEXT: {
  "branch_a": "done",
  "branch_b": "done"
}
```

5.7 W7 — Sub-process Composition (test_subprocess_main + test_subprocess_sub)

A Predefined Process node in test_subprocess_main calls test_subprocess_sub as a child workflow. The child model runs inline, writes sub_result=done into its own context, and the parent receives the child context merged back before its own End node runs. This validates nested sub-workflow calls, call depth enforcement, and context propagation across model boundaries.



Dia. 13: W7 — test_subprocess_main (parent model)



Dia. 14: W7 — test_subprocess_sub (child model called by parent)

Required custom properties on the Predefined Process node in the parent model:

```
type:  predefined_process
model: test_subprocess_sub
```

CLI execution: the engine calls test_subprocess_sub inline, merges the child context, then continues the parent. Both sub_result (from child) and parent_result (from parent) appear in the final context:

```
python3 main.py test_subprocess_main

==============================================================================
MODEL          : test_subprocess_main
STATUS         : completed    |    WAIT STATES : 0
FINAL CONTEXT: {
```

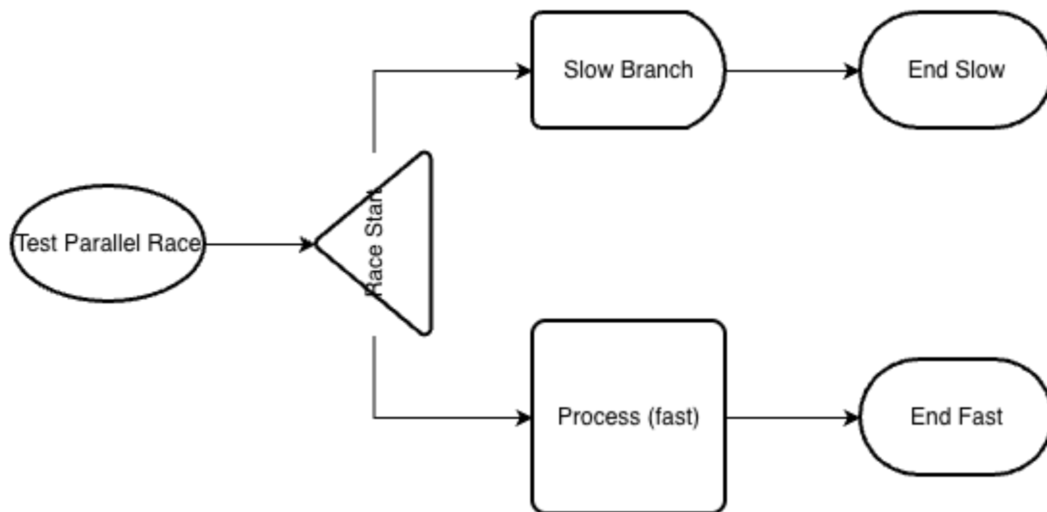

© IBM Corporation 2026

```
  "sub_result": "done",
  "parent_result": "done"
}
```

If --max-call-depth 0 is passed, the engine raises a RuntimeError instead of calling the child model. This limit exists to prevent infinite recursion in self-referential models.

5.8 W8 — Race Mode (test_parallel_race)

An Extract node dispatches two branches with merge_mode: race. There is no Merge node — the first branch to complete writes its winner value and the workflow ends immediately; the slower branch is abandoned. The final context contains only the keys from the winning branch. Because thread scheduling is non-deterministic, either branch may win on any given run.



Dia. 15: W8 — test_parallel_race, first-wins race mode

Required custom properties:

```
Extract node:              merge_mode: race
Process node (branch A):   winner: branch_a
Process node (branch B):   winner: branch_b
```

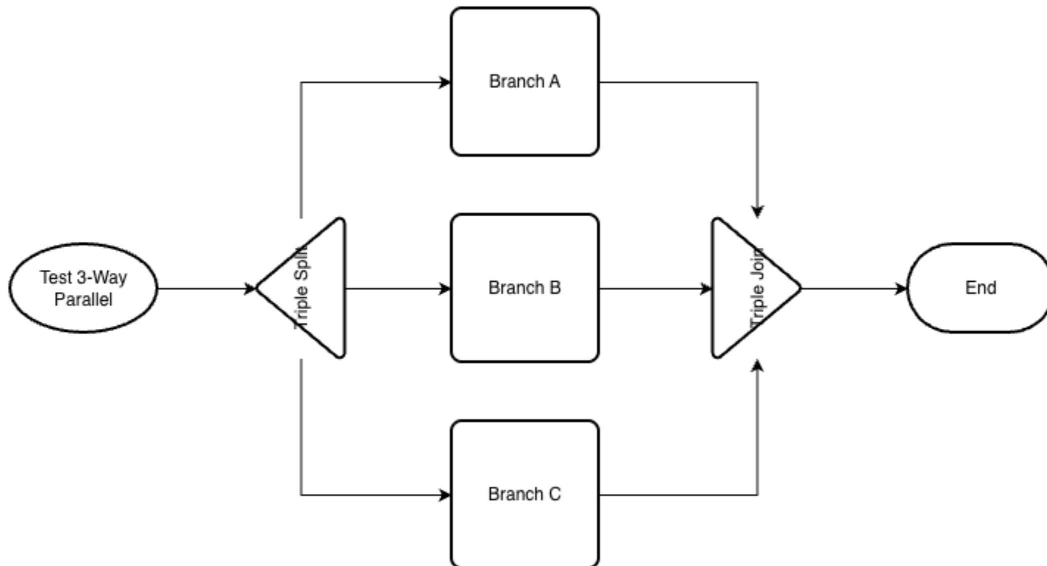
CLI execution: one branch completes and the workflow ends — winner is non-deterministic:

```
python3 main.py test_parallel_race

==============================================================================
MODEL         : test_parallel_race
STATUS        : completed    |   WAIT STATES : 0
FINAL CONTEXT: {
  "winner": "branch_a"    # or "branch_b" — depends on thread scheduling
}
```


© IBM Corporation 2026

5.9 W9 — Three-Branch Parallel Merge (test_parallel_3way)

An Extract node with three outgoing edges dispatches three parallel branches, each writing a distinct context key (branch_a, branch_b, branch_c). A Merge node with merge_mode: sync waits for all three to complete before continuing. This extends W6 to validate that the merge barrier counter correctly tracks three completions rather than two.



Dia. 16: W9 — test_parallel_3way, three-branch sync merge

Required custom properties:

```
Process node (branch A):  branch_a: done
Process node (branch B):  branch_b: done
Process node (branch C):  branch_c: done
Merge node:               merge_mode: sync
```

CLI execution: all three context keys must be present in the final context:

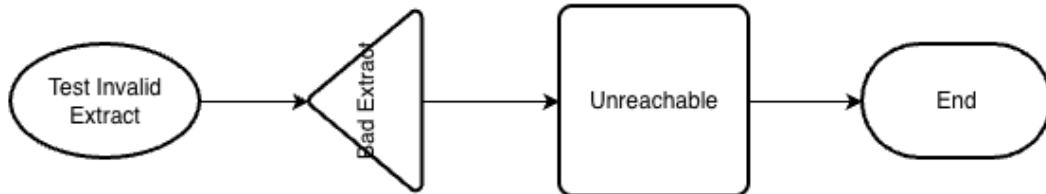
```
python3 main.py test_parallel_3way

==============================================================================
MODEL         : test_parallel_3way
STATUS        : completed    |    WAIT STATES : 0
FINAL CONTEXT: {
  "branch_a": "done",
  "branch_b": "done",
  "branch_c": "done"
}
```

5.10 W10 — Structural Validation (test_single_branch)

A deliberately minimal model used exclusively to exercise the engine's structural validation logic.

© IBM Corporation 2026

It contains an Extract node with only one outgoing edge — which is invalid, since an Extract node requires at least two branches. The engine must detect this during model loading and raise a validation error before any execution begins. This model is not intended to run successfully.



Dia. 17: W10 — test_single_branch, structural validation test (intentionally invalid)

Expected behaviour — the engine raises a validation error on load rather than executing:

```
python3 main.py test_single_branch

RuntimeError: Extract node requires at least 2 outgoing edges (found 1)
```

In the test suite this model is loaded via e.load_model("test_single_branch") to verify that the validation layer catches structural problems before attempting any execution. Do not use this model as a basis for real workflows.

© IBM Corporation 2026

6 Workflow Pattern Reference

This chapter provides a concise reference for all ten canonical workflow patterns included with the engine. Unlike Chapters 4 and 5, which walk through execution step by step, this chapter is a quick-lookup reference — one entry per pattern, with the node structure table, required properties, expected output, and one key thing to watch for.

All models require only main.py and the corresponding .drawio file in the same directory. No cloud credentials are needed.

6.1 Core Patterns

W1 — Linear Flow with Decision (test_linear)

Simplest end-to-end workflow. Tests decision routing, context merging, idempotency, and model validation.

Node	Type	Key properties
Start	Start	(none)
Set Result	Process	result: ok
Route?	Decision	Routes on context key "result"
Done	End	Taken when result = ok
End Fallback	End	Taken on any other value

```
python3 main.py test_linear --context '{"result":"ok"}'
```

Expected: STATUS: completed · FINAL CONTEXT: {"result":"ok","processed":"true"}

Watch for: Process nodes always perform an unconditional context merge — the node property overwrites any existing context key with the same name.

W2 — Blocking Delay (test_delay_blocking)

Delay node in wait_mode: blocking. Holds the process open for the full duration, then continues automatically. No resume call needed.

Node	Type	Key properties
Start	Start	(none)

© IBM Corporation 2026

Wait 2s	Delay	delay: 2, wait_mode: blocking
Set Flag	Process	delay_passed: true
End	End	(none)

```
python3 main.py test_delay_blocking
```

Expected: STATUS: completed after ~2 s · FINAL CONTEXT: {"delay_passed":"true"}

Watch for: The CLI process hangs for the full delay duration — this is expected. Use wait_mode: scheduled (W3) if the engine must return immediately.

W3 — Scheduled Delay (test_delay_scheduled)

Delay node in wait_mode: scheduled. Returns paused immediately with a wake_at timestamp; requires an external trigger to resume after the delay expires.

Node	Type	Key properties
Start	Start	(none)
Wait 2s	Delay	delay: 2, wait_mode: scheduled
Set Flag	Process	delay_passed: true
End	End	(none)

```
python3 main.py test_delay_scheduled              # Run 1 — pauses
python3 main.py test_delay_scheduled --id <id>  # Run 2 (after 2 s) — completes
```

Expected Run 2: STATUS: completed · FINAL CONTEXT: {"delay_passed":"true"}

Watch for: Resuming before wake_at re-pauses without error. The delay timer is wall-clock time stored in the state file.

W4 — Blocking REST Call (test_rest_blocking)

Process node (type: rest, wait_mode: blocking). Calls the external API synchronously, merges the JSON response, and completes — all in a single run.

Node	Type	Key properties
Start	Start	(none)

© IBM Corporation 2026

Call API	Process	type: rest · method: GET · url: <endpoint> · wait_mode: blocking
End	End	(none)

```
python3 main.py test_rest_blocking
```

Expected: STATUS: completed · FINAL CONTEXT includes all top-level keys from the API JSON response.

Watch for: API response keys overwrite existing context keys with the same name. Unreachable endpoint raises RuntimeError; no state is written.

W5 — Scheduled REST Call (test_rest_scheduled)

Process node (type: rest, wait_mode: scheduled). No REST call on Run 1 — records a pending wait. REST fires on Run 2; response is merged and the instance completes.

Node	Type	Key properties
Start	Start	(none)
Submit	Process	type: rest · method: GET · url: <endpoint> · wait_mode: scheduled
End	End	(none)

```
python3 main.py test_rest_scheduled              # Run 1 — records wait, no API call
python3 main.py test_rest_scheduled --id <id>  # Run 2 — fires REST call, completes
```

Watch for: Concurrent resumes before the response is written may call the endpoint twice — design REST endpoints to be idempotent.

6.2 Parallel Execution Patterns

W6 — Two-Branch Parallel Sync (test_parallel_sync)

Extract node splits into 2 concurrent branches. Merge node (merge_mode: sync) waits for both. Both context keys must appear in the final context.

Node	Type	Key properties

© IBM Corporation 2026

Start	Start	(none)
Split	Extract	(2 outgoing edges)
Branch A	Process	branch_a: done
Branch B	Process	branch_b: done
Join	Merge	merge_mode: sync
End	End	(none)

```
python3 main.py test_parallel_sync
```

Expected: STATUS: completed · FINAL CONTEXT: {"branch_a":"done","branch_b":"done"}

Watch for: Set ENGINE_URL to dispatch branches as independent HTTP requests — avoids serverless platform timeout issues.

W8 — Race Mode (test_parallel_race)

Extract node (merge_mode: race). First branch to complete wins; remaining branches are abandoned. Final context contains only the winning branch's keys.

Node	Type	Key properties
Start	Start	(none)
Split	Extract	merge_mode: race · 2 edges
Process A	Process	winner: branch_a
Process B	Process	winner: branch_b
End A	End	(none)
End B	End	(none)

```
python3 main.py test_parallel_race
```

Expected: STATUS: completed · FINAL CONTEXT: {"winner":"branch_a"} or {"winner":"branch_b"}

Watch for: Losing branch is silently abandoned. Use Merge (W6/W9) if all branches must contribute context.

W9 — Three-Branch Parallel Merge (test_parallel_3way)

© IBM Corporation 2026

Extract with 3 outgoing edges. Merge node (merge_mode: sync) waits for all three. Extends W6 to validate the merge barrier counter at N=3.

Node	Type	Key properties
Start	Start	(none)
Split	Extract	(3 outgoing edges)
Branch A	Process	branch_a: done
Branch B	Process	branch_b: done
Branch C	Process	branch_c: done
Join	Merge	merge_mode: sync
End	End	(none)

```
python3 main.py test_parallel_3way
```

Expected: STATUS: completed · FINAL CONTEXT: {"branch_a":"done","branch_b":"done","branch_c":"done"}

Watch for: Merge barrier count is derived from incoming edges at parse time — always validate after adding or removing branches.

6.3 Advanced Patterns

W7 — Sub-process Composition (test_subprocess_main + test_subprocess_sub)

Predefined Process node calls a child .drawio model. Parent context is passed to the child; child's final context is merged back on completion. Nesting capped at 5 levels.

Node (parent)	Type	Key properties
Start	Start	(none)
Initialize	Process	sub_initialized: true
Run Sub	Predefined Process	model: test_subprocess_sub
End	End	(none)

© IBM Corporation 2026

```
python3 main.py test_subprocess_main
```

Expected: STATUS: completed · FINAL CONTEXT: {"sub_initialized":"true","sub_result":"done","parent_result":"done"}

Watch for: Child Start node properties are defaults only — parent context values take precedence. Pass --max-call-depth 0 to confirm the depth limit raises RuntimeError.

W10 — Structural Validation (test_single_branch)

Intentionally invalid: Extract node with only 1 outgoing edge (minimum is 2). Confirms pre-execution validation. No state file is created; no nodes are executed.

Node	Type	Notes
Start	Start	(none)
Split	Extract	Only 1 outgoing edge — invalid
Process	Process	Never reached
End	End	Never reached

```
python3 main.py test_single_branch
```

Expected: RuntimeError raised at validation. No partial execution. No state file written.

Watch for: Validation failures are always safe — no external side effects occur. Use this as a template for writing negative test cases.

6.4 Summary Table

Model	Pattern	Key behaviour verified
W1 test_linear	Linear + Decision	Context merge, conditional routing, wildcard fallback
W2 test_delay_blocking	Blocking delay	Synchronous wait, auto-resume
W3 test_delay_scheduled	Scheduled delay	Async pause, external resume, expiry check
W4 test_rest_blocking	Blocking REST	Synchronous HTTP call, JSON response merge
W5 test_rest_scheduled	Scheduled REST	Async REST pause, event-state file,

© IBM Corporation 2026

		resume
W6 test_parallel_sync	2-branch merge	Parallel threads, barrier convergence, context union
W7 test_subprocess_main	Sub-process	Child model call, context pass-through, depth limit
W8 test_parallel_race	Race mode	Winner-takes-all, branch abandonment
W9 test_parallel_3way	3-way merge	Three-branch barrier, full context union
W10 test_single_branch	Validation	Pre-execution validation, safe rejection

© IBM Corporation 2026

7 Appendix

This appendix provides the reference content of all configuration and deployment files included in the project. Not all files are required in every deployment scenario. `main.py` and `CREDENTIALS.txt` are the only files needed for running the engine locally from the command line (except the draw.io model file in the root folder if you are running locally). The `requirements.txt` is only needed when setting up the Python environment (and as part of containerization). landing.html is additionally needed when running the engine in --server mode to serve the root landing page at GET /.

`Containerfile`, `containerignore`, `wsgi.py`, and `PODMAN.md` are specific to containerised deployments, `wsgi.py` in particular is only used when the engine is served by gunicorn inside a container and can be ignored entirely for local development.

The draw.io model files are included as working examples and is not part of the engine itself. Do not forget to put them into the root folder of `main.py` python file (i.e. the draw.io workflow engine) or to upload them into respective Cloud Object Storage bucket if they should be accessed using cloud resources.

7.1 Server Mode Web Interface

When the engine is started in --server mode (python3 main.py --server), it exposes three additional browser-friendly endpoints. These require no authentication and are always accessible.

7.2 GET / — Server Landing Page

Navigating to the root URL of the running server (e.g. http://localhost:8080/) displays a server landing page styled in the IBM Carbon Design System. The page shows a live status indicator, real-time runtime metrics (uptime, total requests, in-flight instances, completed workflows — refreshed every 10 seconds), and quick-access links to /docs, /openapi.json, and /health.

The landing page is served from the file landing.html, which must be placed in the same directory as main.py. If landing.html is missing at runtime, the engine falls back to a minimal plain-HTML page with navigation links.

7.3 GET /docs — Interactive API Documentation (Swagger UI)

The /docs endpoint serves an interactive Swagger UI page that documents all API endpoints.

© IBM Corporation 2026

Swagger UI is loaded from the public CDN (unpkg.com/swagger-ui-dist@5) — no additional files need to be deployed. The page reads the OpenAPI specification from GET /openapi.json and renders it as a browsable, try-it-out interface. Authentication credentials can be entered directly in the UI using the Authorize button when Basic Auth is enabled.

7.4 GET /openapi.json — OpenAPI 3.0 Specification

The /openapi.json endpoint returns the complete OpenAPI 3.0 specification of the engine API in JSON format. This machine-readable document can be imported into tools such as Postman, Insomnia, or any OpenAPI-compatible platform. The specification is built dynamically inside main.py and covers all six workflow and operations endpoints, including request schemas, response codes, path and query parameters, and the Basic Auth security scheme. An alias endpoint GET /docs/json returns the same specification.

7.5 landing.html — Server Landing Page File

landing.html is the HTML source file for the server root page served at GET /. It must be placed in the same directory as main.py. The file uses only standard browser APIs with no build step or bundler required. The only external resource it loads is the IBM Plex Sans and IBM Plex Mono fonts from Google Fonts (fonts.googleapis.com). Live status and metrics are fetched via JavaScript from GET /health and GET /metrics on the same server.

7.6 Distribution Package — Required Files

The following table lists all files that must be included in the distribution ZIP archive, including the new landing.html added in this release.

File	Required for	Notes
main.py	All deployments	The workflow engine. Single Python source file.
requirements.txt	All deployments	Run pip install -r requirements.txt to set up the Python environment.
CREDENTIALS.txt	All deployments	Runtime configuration template. Remove or sanitise credentials before distributing.

© IBM Corporation 2026

landing.html	Server mode (--server)	Root landing page served at GET /. Must be in the same directory as main.py.
Containerfile	Container deployments	OCI/Docker build file. Required only for podman/docker builds.
containerignore	Container deployments	Files excluded from the container image build context.
wsgi.py	Container deployments (gunicorn)	WSGI entry point for gunicorn. Not used in local development.
PODMAN.md	Container deployments	Podman-specific setup and run instructions.
README.md	Reference only	Quick-start reference card. Not required at runtime.
*.drawio files	Runtime (model execution)	Workflow model files. Place alongside main.py or configure COS backend.
User Guide PDF	Reference only	Not required at runtime; include in archive for documentation.

End of Document

© IBM Corporation 2026