LYFTDATA PRODUCT DOCUMENTATION

Deployments

Catalogs, deployment concepts, workflows, triggers, deployment manager, and a file-store round trip.

Version 2.1.0. Generated from the docs.lyftdata.com source corpus on 2026-05-25.

Contents

1. Deployments Overview
2. Catalog
3. Deployments Core Concepts
4. Deployment Manager
5. Triggers
6. Tutorial: Quickstart File Store Roundtrip
7. Workflows

Deployments Overview

Orchestrate multi-job systems with workflows and the Deployment Manager

Source: deployments/overview.md

Deployments is the part of LyftData that turns **workflows** (graphs) into **running systems** across one or more workers.

> note

A workflow is a **control-plane description** of what should run and how it is wired. It does not run as one long-lived process; the Deployment Manager expands it into concrete jobs deployed onto workers.

> Note

If **Jobs** teaches you how to author a single job (one input → actions → one output), Deployments teaches you how to:

- compose **many jobs** into a system,
- choose **how data moves** between steps (transports),
- decide **where** each step runs (placement),
- and safely **plan + apply** changes with a clear diff (“no surprises”).

When to use Deployments

Use Deployments when you need any of the following:

- A pipeline that is naturally **multi-stage** (ingest → route/transform → deliver).
- **Fan-out** (send the same events to multiple consumers) or **fan-in** (combine multiple sources).
- A **durable hop** between stages (spooling/landing before downstream processing).
- **Placement and scaling**: run different parts of the pipeline on different worker groups and scale them independently.
- A repeatable rollout flow with **plan** → **apply** → **reconcile** and an audit trail.
- A CLI-first GitOps workflow where platform operators keep deployment desired state in reviewed files and promote it across environments.

If you just need a single job running on one worker, you can stay in **Jobs**.

The Deployments lifecycle (high level)

1. Pick a starting point from the **Catalog** (a **workflow**, **perspective**, or **blueprint**). 2. If you chose a perspective, clone/start from it, complete bindings in **Workflows**, and publish a deployable workflow version. 3. Click **Plan & register** to open the “create deployment” flow. 4. Choose a name, a worker group, and fill in wizard fields when present. 5. **Plan**: preview what will be created/updated and where it will run. 6. **Apply**: stage and deploy the concrete jobs onto workers. 7. **Operate**: watch health/status, and use **reconcile** to converge drift back to desired state.

Where this lives in the UI

- **Catalog**: discover workflows, perspectives, blueprints, and library jobs (local or portal).
- **Workflows**: author drafts and publish versions.
- **Deployment Manager**: plan/apply/reconcile deployments.
- **Workloads / Workers**: see what is actually running and where.

> note

The Catalog can include locally installed items and (when licensed) Portal items. Some authoring primitives are server-shipped “builtins” that appear in the workflow editor palette rather than the Catalog.

> Note

Next pages

- Start with Catalog to understand what each catalog item type is used for.
- Read Core Concepts for the primitives (nodes, edges, channels, messages, KV state, control plane).
- Read Workflows for authoring/versioning and transports.
- Read Triggers for curated “run this” actions you can invoke from the UI and MCP.
- Read Deployment Manager for plan/apply/reconcile, placement, and drift.
- Read GitOps CLI Beta for file-based desired-state reconciliation, promotion, and rollback.
- Follow Tutorial: Quickstart File Store Roundtrip for an end-to-end walkthrough.

> note

Deployments is a licensed surface. In Community Edition, deployments APIs/UI flows are unavailable and external-worker deployment flows are blocked. See Licensing and Community Edition.

> Note

Catalog

Browse workflows, perspectives, blueprints, and library jobs to kick-start deployments and jobs

Source: deployments/catalog.md

The **Catalog** is where you browse reusable building blocks and starters for Deployments and Jobs.

It can include:

- **Local** items bundled with the server or created in your workspace.
- **Portal** items (licensed) from a shared catalog that you can install and keep up to date.

Catalog item types

Workflows

Workflows are end-to-end graphs (DAGs) composed of blueprints.

Use them when you want a multi-job system. In the Catalog, filter to **Workflows** and click **Plan & register** to create a deployment from a selected workflow version.

Perspectives

Perspectives are starter workflow skeletons with intentionally unbound slots.

Use them when you want a standard architecture but still need to choose concrete building blocks for your environment. Perspectives are not deployable as-is: start from a perspective, complete bindings in the Workflows editor, publish, then run **Plan & register**.

Blueprints

Blueprints are reusable deployment building blocks (modules). They can be used in two ways:

- Import them into a workflow while authoring, or
- Click **Plan & register** to create a deployment directly from a blueprint (useful for “single-module” systems).

Jobs (library jobs)

Jobs are managed job snippets from the catalog. They are useful as known-good single-job starting points.

In the Catalog, clicking **Plan & register** on a Job takes you to the “new job” flow with the job content prefilled.

Common actions

- **Search + filters**: use the tabs (Workflows / Perspectives / Blueprints / Jobs) and filters (Source, Availability, tags) to narrow down choices.
- **Preview**: preview graphs for workflows/perspectives/blueprints, and preview job content for library jobs.
- **Details**: open the details drawer to see provenance, tags, version, and related items.

- **Plan & register**:
- For workflows/blueprints: starts a new deployment.
- For perspectives: starts a workflow-authoring flow so you can complete bindings before deployment.
- For jobs: starts a new job draft in the Jobs UI.

Portal vs local

If your deployment is licensed, you can switch **Source** → **Portal** to browse shared catalog content. Portal items may be available as:

- **Portal-only**: not installed locally yet.
- **Installed**: installed into your server.
- **Update available**: a newer version exists in the portal.

Next

- Learn how workflows are authored and versioned in Workflows.
- Learn how deployments are planned/applied in Deployment Manager.
- If you only need a single job, start with Jobs.

Deployments Core Concepts

The primitives behind workflows, transports, triggers, and orchestration

`Source: deployments/core-concepts.md`

Deployments is easiest to learn when you separate **data flow** from **control flow**.

This page introduces a minimal set of primitives that match the current product:

- **Graph** (what's wired to what),
- **Data** (events moving over channels),
- **Trigger** (signals/messages that wake things up),
- **State** (KV, contexts, secrets),
- **Control** (the Deployment Manager that plans/applies/reconciles).

Graph: steps and edges

Workflow

A **workflow** is a versioned graph made of **steps** and **edges**. It is authored as a draft, then published as an immutable version.

Step (node)

A **step** (node) is one box in the workflow graph. Each step:

- has a stable `id`,
- may be bound to a building block (a blueprint or workflow module),
- declares inputs/outputs (`in` / `out` bindings),
- and can optionally declare branching (`dispatch`) for multi-port outputs.

Edges: channel edges vs message edges

Workflows have two different kinds of "wiring":

- **Channel edges (data flow):** events move from step A to step B over a named channel.
- **Message edges (trigger flow):** step A emits a tagged internal message, and step B runs when that tag is observed.

Message edges are for "wake up" signaling; they are not how bulk data moves.

Data: events, channels, transports

Event

An **event** is the payload that flows along channel edges (what most people think of as "the data").

Channel

A **channel** is a named logical link (for example `channel:orders_raw`) referenced by step inputs/outputs.

Transport kind

A **transport** defines how events are delivered along a channel edge. Two important starting points:

- **Worker channel** (fast, in-memory hop between jobs on workers).
- **Durable spool** (for example file-store) used as a landing zone between stages.

Durable spooling is essential for pipelines that need replay/backfill or blast-radius isolation between stages.

Ports and dispatch

Steps can have multi-port outputs (a map of named outputs). When a step has multiple outputs, its **dispatch mode** defines how events are routed:

- `clone` duplicates each event to all output ports,
- `round-robin` partitions events across ports,
- `conditional` routes events based on a field value.

Trigger: signals, messages, request/reply

> note

This section describes trigger flow as a message-wiring concept in workflows (message edges). If you mean Triggers as an operator-facing registry of “run this now” actions (UI + MCP dynamic tools), see Triggers.

> Note

Message bus (internal coordination)

LyftData uses an internal **message bus** for coordination and signaling. It is separate from channel-based data flow. For the concrete “emit + subscribe” mechanics (tags, filters, and the `internal-messages` input event shape), see Message Bus.

Message edges (trigger wiring)

A message edge compiles into “tag emitted” → “input triggers on tag received”. Sometimes the system inserts a relay step to convert “data events on a channel” into a tagged trigger message.

Request/reply (“serviceable”) messages

Some internal messages are request/reply (RPC-like): the server asks a worker to do something (or report something) and expects a reply. Public docs usually don’t require the details, but it’s helpful context when debugging control-plane ↔ worker behavior.

Emit hook (data → message bridge)

Jobs can emit sidecar messages (for checkpoints/backlog/progress) after a successful primary write, without changing their primary output. This is one of the bridges between the data plane and the control plane.

State: KV, contexts, secrets

Worker KV (memory/cache/work queues)

The Worker KV store is a lightweight key/value and work-queue abstraction used for:

- checkpoints and cursors,
- backlog queues with leases (at-least-once work processing),
- dedupe/idempotency tracking,
- and coordination state (for example, leader-election style locks).

Think of it as “memory + coordination state” that survives across runs and replicas.

Contexts, variables, secrets

Deployable systems need configuration and credentials. Contexts/variables let you parameterize workflows across environments; secrets let you safely deliver credentials to workers.

Control: the Deployment Manager

The **Deployment Manager** is the control plane that turns workflows into running jobs on workers.

At a high level it:

- **Plans** changes (expands a workflow into concrete jobs/channels and shows the diff),
- **Applies** the plan (stages and deploys jobs),
- **Places** steps onto worker groups and sets replicas,
- and **Reconciles** drift back to desired state.

Starter artifacts: jobs, perspectives, blueprints, workflows

- **Library job:** a single-job starter (useful when you want “one job that does X”).
- **Perspective:** a starter workflow skeleton with unbound slots that you clone/complete before deployment.
- **Blueprint:** a reusable workflow module (building block) used inside workflows.
- **Workflow:** an end-to-end DAG composed of blueprints (deployable via the Deployment Manager).

Next:

- Read Workflows for how edges/transports and publishing work in practice.
- Read Deployment Manager for plan/apply/reconcile.
- Read Catalog for how workflows/perspectives/blueprints and library jobs show up in the UI.

Deployment Manager

Plan, apply, and reconcile workflows across workers

Source: deployments/deployment-manager.md

The **Deployment Manager** is the control plane that turns a published workflow into concrete jobs running on workers.

Mental model: workflow → deployment → workloads

Think of Deployments as a compilation pipeline:

```
Published workflow version
→ deployment record (desired state + history) → per-worker job deployments (rendered
artifacts) → workloads (what is actually running)
```

This is why the Deployment Manager is separate from the workflow editor: workflows describe intent; deployments produce concrete runnable artifacts.

The three verbs: plan, apply, reconcile

Plan

Planning is where the system compiles the workflow into deployable concrete artifacts:

- expands blueprints and imported workflow modules,
- resolves transports (and may insert required adapter jobs),
- validates bindings (channels, imports, required parameters),
- and produces a **diff** preview (“what will change, and where”).

Apply

Applying executes the plan:

- stages the concrete jobs,
- renders per-worker deployment artifacts,
- deploys them to the selected worker group(s),
- and records history so you can audit and roll back by deploying an older workflow version.

Reconcile

Reconcile converges “what is running” back toward “what you asked for”:

- detects drift (workers out of sync, missing replicas, changed placement),
- and re-applies the desired state when needed.

Managed artifacts (what to edit vs what to regenerate)

When you **apply** a deployment, the system generates managed job artifacts from your workflow. In general:

- Change behavior by editing the **workflow** (or selected blueprints) and applying a new workflow version.
- Avoid “hot-editing” the generated jobs directly; those changes will look like drift and can be overwritten by reconcile.

If you need a bespoke job outside the deployment manager, clone it into an unmanaged copy and manage it through Jobs.

Placement and scaling (worker groups + replicas)

Workflows are designed to run across worker groups:

- **Placement** chooses where each step runs (which worker group).
- **Replicas** control horizontal scale per step (and isolate noisy stages).

This lets you isolate heavy ingest from sensitive delivery, or scale routing independently from collection.

> note

Placement is intentionally “no surprises”: after an apply, placement decisions are sticky unless you re-plan/re-apply (or explicitly choose **Recompute placement for next plan** in the UI).

> Note

Common reasons planning or apply can fail

- **Missing producers/consumers**: workflows must have a complete channel graph (every channel has a producer and a consumer).
- **Missing worker group / no workers available**: planning can still return a preview with warnings, but apply cannot place jobs until an eligible worker group exists.
- **Missing secrets/config** required by a selected blueprint (credentials, endpoints, storage paths).
- **Incompatible artifacts**: license or runtime feature requirements are not met (for example, external workers unavailable in Community Edition).

Next: follow Tutorial: Quickstart File Store Roundtrip for an end-to-end walkthrough.

Triggers

Publish and invoke curated “run this” actions for workflows and jobs from the UI and MCP

`Source: deployments/triggers.md`

Triggers are LyftData’s registry of curated “run this now” actions. A Trigger is a named invocation with schema-validated parameters that dispatches a message to a target job. This gives teams a safer alternative to handcrafting low-level message traffic when they want to run smoke checks, one-off operational workflows, or self-serve actions on demand.

> note

Triggers (the registry) are different from the DSL `trigger` input, which is a scheduling primitive used inside a job definition. See Trigger input.

> Note

When to use Triggers

Use Triggers when you want:

- A discoverable list of “ready to run” actions for operators or teammates.
- Parameter validation (a schema) instead of free-form JSON.
- Policy and guardrails around who can invoke what.
- An auditable invocation record with a stable `invocation_id` you can poll.

If you want something to run on a schedule, use the normal scheduling primitives (including the DSL `trigger` input) instead of a manual invocation surface.

How Triggers work

At a high level:

1. An admin publishes a Trigger definition (slug, schema, defaults, and target job). 2. A user or admin invokes the Trigger with parameters. 3. LyftData dispatches a tagged user-generated message to the target job. 4. The invocation is tracked and can be polled by `invocation_id` until it reaches a terminal state.

Trigger definitions can be published to one or more surfaces (UI and/or MCP). Invocation access is still enforced by server-side permissions and the trigger’s invocation policy.

Publish and manage Triggers (admin)

Triggers are intended to be curated. In most teams, admins publish and revise Triggers, and users/operators only invoke the approved set.

When publishing a Trigger, you typically define:

- **Slug + display name**: the stable identifier and user-facing name.
- **Parameter schema + defaults**: JSON schema for validation and optional default parameters.

- **Target job + tag**: which job receives the invocation message (and optionally a specific message tag). Use a tag if the target job needs stable message-trigger filtering.
- **Publication surfaces**: whether the Trigger should appear in the UI and/or as an MCP tool.
- **Invocation policy**: who can invoke it, and any per-user rate limits.
- **Optional response slot + timeout**: configure `response_slot` when the caller should receive a structured result (see “Trigger result shaping” below).
- **Optional proxy policy**: if enabled, the dispatch message carries a `trigger_proxy` envelope so sinks (for example `http-post`) can capture and report a proxy-style response.

Triggers are revisioned. If you change a Trigger’s definition, you publish a new revision; you can also deactivate a Trigger so it no longer appears as invokable.

Publish and revise over the API

If you prefer to treat Triggers as a deployable artifact, the admin API (admin principals) exposes publish/revise/deactivate endpoints:

- Publish (creates revision 1): `POST /api/triggers/publish`
- Revise (creates a new revision): `POST /api/triggers/<slug>/revise`
- Deactivate (marks inactive): `POST /api/triggers/<slug>/deactivate`

Example: publish a tenant-scoped Trigger with a response slot and proxy policy (note that proxy modes require `response_slot`):

```
{
"slug": "smoke_trigger", "tenant_id": "tenant-a", "display_name": "Smoke trigger",
"description": "Run a smoke check and return a structured result.", "parameter_schema": {
"type": "object", "properties": { "env": { "type": "string" }, "depth": { "type": "integer",
"minimum": 1 } }, "required": ["env"] }, "default_params": { "depth": 1 }, "target_job_name":
"smoke_trigger_handler", "target_message_tag": "smoke.invoke", "publication": { "mcp": true,
"ui_admin": true, "ui_user": false }, "invoke_policy": { "allow_admin": true, "allow_user":
true }, "response_slot": "smoke.reply", "response_timeout_seconds": 30, "proxy_policy": {
"mode": "passthrough", "timeout_seconds": 30 } }
```

To revise a Trigger, send the full draft again to the revise endpoint (the server does not patch-in-place; it creates a new revision). For platform-scoped Triggers, use `tenant_id: "platform"` (platform-scoped principals only).

Trigger Dispatch Payload (`trigger_invoke`)

Trigger invocations are delivered to the target job as a `user-generated` message whose payload (`job_event`) has a stable top-level shape:

```
{
"type": "trigger_invoke", "version": 1, "invocation_id": "…", "trigger": { "id": "…", "slug":
"smoke_trigger", "revision": 3, "tenant_id": "tenant-a" }, "caller": { "username": "alice",
"role": "admin" }, "params": { "env": "prod", "depth": 2 }, "requested_at_ns":
1739370000000000000 }
```

If the Trigger revision has proxy mode enabled, the payload also includes a `trigger_proxy` object (tenant/id/slot/nonce plus policy and bounds). Jobs can pass that object through to sinks; some sinks emit correlated `trigger-proxy-response` messages automatically.

In a message-triggered job run, this payload is available through `${msg|...}` (see Variable Expansion).

Wire Up The Target Job

Most Trigger target jobs are **message-triggered** so they run only when invoked.

Example: a single-event job that runs on a tag and can read parameters via `${msg|job_event.params.*}`.

```
name: smoke_trigger_handler
input: echo: trigger: message: filter-kind: user filter-type: [user-generated] filter-tag:
smoke.invoke json: true event: "{}"
```

> tip

Use a dedicated tag namespace (for example `smoke.invoke` or `trigger.smoke.invoke`) so your job's filter stays stable even if other messages exist in the environment.

> Note

Trigger Result Shaping (`response_slot`)

By default, a Trigger invocation is **dispatch-only**: the server records whether the invocation message was accepted for dispatch, but it does not wait for a structured result from the workflow.

If you configure a `response_slot` on the Trigger revision, LyftData switches to **response-slot mode**:

- The server keeps the invocation state in `dispatched` after message dispatch.
- The invocation reaches a terminal state only when the server observes a correlated `trigger_response` message for the same `invocation_id` and `response_slot`, or when the response timeout is reached.
- The invocation `result` is shaped to include the structured response payload.

> note

`trigger_response` and `trigger_proxy_response` envelopes require a non-empty `tenant_id`. Response-slot mode and proxy capture are currently supported only for tenant-scoped Triggers (platform-scoped invocations cannot be completed through these envelopes).

> Note

Emit A `trigger_response` From A Job

To complete an invocation in response-slot mode, emit a `trigger_response` payload through the message bus.

Practical rules:

1. The payload is strict: it must match the v1 envelope exactly (extra top-level keys are rejected). 2. `response_slot` must match the Trigger revision's configured slot. 3. `emitted_at_ns` must be a number. If you build it from `${...}` expansions, use `convert` with `conversion: json` (not `num`) to preserve integer precision for nanosecond timestamps.

This pattern keeps your job's normal output separate from the Trigger response envelope by using `output.message.input-field`.

```
name: smoke_trigger_handler
input: echo: trigger: message: filter-kind: user filter-type: [user-generated] filter-tag:
smoke.invoke json: true event: "{}"


actions:



  - add:



output-fields: trigger_response: type: trigger_response version: 1 tenant_id:
"${msg|job_event.trigger.tenant_id}" invocation_id: "${msg|job_event.invocation_id}"
response_slot: "smoke.reply" nonce: "${msg|job_event.invocation_id}" outcome: succeeded
result: ok: true message: "Smoke check passed" emitted_at_ns:
"${msg|job_event.requested_at_ns}"



  - convert:



conversions:



  - field: trigger_response.emitted_at_ns



conversion: json


output: message: input-field: trigger_response
```

If you need to report failure, set `outcome: failed` and include `error_code` and/or `error_message` (you can still include a `result` object for structured details).

What Callers Receive

The invoke call (UI/MCP/API) returns an invocation record. When a response is observed, the server stores it under `invocation.result.trigger_response`, and the invocation transitions to `succeeded` or `failed`.

If no response is observed before the configured response deadline, the invocation transitions to `timed_out` and the result is shaped with the deadline.

Proxy Capture (`trigger_proxy_response`)

If the Trigger revision has proxy mode enabled, the dispatch payload includes a `trigger_proxy` envelope. When that envelope is present:

- Pass the envelope through as part of the event you send to the sink (outputs read event fields, not `${msg|...}` bindings).
- The `http-post` output captures allowlisted response headers and a bounded body and emits a `trigger_proxy_response` message automatically.
- The server records it under `invocation.result.trigger_proxy_response` (plus a `trigger_proxy_recorded_at` timestamp) when the correlation matches.

Recording a proxy response does not complete the invocation by itself. If you want the invocation to reach a terminal state quickly, emit a `trigger_response` as well.

Example: copy the `trigger_proxy` object from the Trigger dispatch payload into the current event (and parse it back into JSON) so `http-post` can observe it:

```
- add:
output-fields: trigger_proxy: "${msg|job_event.trigger_proxy}"


  • convert:


conversions:


  • field: trigger_proxy


conversion: json
```

Invoke Triggers in the UI

The Triggers UI is the “run now” surface:

- Pick a Trigger from the list.
- Fill in the schema-defined fields (defaults can be prefilled).
- Run it and follow status by `invocation_id` until it completes.

Invoke Triggers over the API (automation)

For automation, the API exposes a list/invoke/status flow:

- List invokable Triggers for a surface (for example UI): `GET /api/triggers?surface=ui&invokable_only=true`
- Invoke a Trigger: `POST /api/triggers/<slug>/invoke`
- Fetch status/result: `GET /api/trigger-invocations/<invocation_id>`

Invocations are asynchronous. If you need a definitive completion record, always poll by `invocation_id`, even if the initial invoke call returns quickly.

Invoke Request Body

`POST /api/triggers/<slug>/invoke` accepts a JSON body with optional control knobs:

```
{
"params": { "env": "prod", "depth": 2 }, "idempotency_key": "smoke-prod-2026-03-
13T10:15:00Z", "wait_timeout_seconds": 15, "tenant_id": "tenant-a", "callback": { "url":
"https://automation.example/hooks/lyftdata-trigger", "headers": { "authorization": "Bearer …"
} } }
```

Practical behavior:

- `params` must be a JSON object. `null` is treated as `{}`.
- `tenant_id` is required when the caller has multiple tenant memberships. Platform-scoped callers use an empty tenant scope.
- `idempotency_key` deduplicates by `(tenant scope, trigger slug, caller username, idempotency_key)`. A retry with the same key returns the original `invocation_id` instead of dispatching a second message.
- `wait_timeout_seconds` controls how long the invoke endpoint waits before returning the current invocation record. Default: 15 seconds. Max: 300. It does not change how long the workflow has to produce a response.

Timeouts: Wait vs Response vs Proxy

These timeouts control different parts of the system:

- `wait_timeout_seconds` (invoke request) bounds the API call's "wait for a more up-to-date invocation record" behavior.
- `response_timeout_seconds` (Trigger definition, only when `response_slot` is set) is the deadline for observing a correlated `trigger_response` before the invocation becomes `timed_out` (default: 15 seconds).
- `proxy_policy.timeout_seconds` (Trigger definition, proxy mode) bounds the proxied sink attempt (for example, the `http-post` request timeout for capturing a `trigger_proxy_response`). Max: 300 seconds.

Callback Delivery (automation)

If you provide `callback`, the server posts invocation state updates to your URL as best-effort automation glue.

Callback request rules:

- `callback.headers` must be a JSON object of string values (max 32 headers). Invalid header names/values are rejected.
- Delivery is retried up to 5 times with backoff on transport errors and retryable status codes (`408`, `429`, `5xx`).
- Callback delivery does not block invocation dispatch or completion.

Callback payload shape:

```
{
"version": 1, "event_id": "…", "event_type": "trigger_invocation_state", "emitted_at":
1739370000000000000, "invocation": { "invocation_id": "…", "trigger_slug": "smoke_trigger",
"trigger_revision": 3, "tenant_id": "tenant-a", "state": "dispatched", "result": { "status":
"accepted_for_dispatch", "mode": "response_slot", "state": "dispatched" } } }
```


Invoke Triggers from MCP (dynamic tools)

When a Trigger is published for MCP, it appears as a tool in your MCP client. These tools are generated dynamically from the registry at tool-list time, and tool names are derived from the Trigger slug, for example `trigger_invoke__smoke_trigger`.

Operationally:

- Trigger tools are write-gated and require starting the MCP server with `--allow-write`.
- Trigger tools show up only if the Trigger is published for MCP and your identity is allowed to invoke it.
- The tool returns an `invocation_id`; use `trigger_invocation_get` to fetch status/result.
- Trigger tools accept the Trigger's schema fields plus reserved control keys. Reserved keys: `_idempotency_key` (dedupe retries) and `_wait_timeout_seconds` (default 15, max 300).

Where to go next

- Workflows for message edges and control-plane composition.
- Message Bus for how the aggregator message model maps into job DSL (`internal-messages` and `output.message`).
- Messages for the live UI stream (channels, filters, export).
- MCP Overview for setup, auth, and why trigger tools are treated as writes.

Tutorial: Quickstart File Store Roundtrip

Deploy the built-in quickstart workflow (write → read → print) using the Deployment Manager

Source: deployments/tutorials/file-store-roundtrip.md

This tutorial walks through the built-in **Quickstart: File Store Roundtrip** workflow (`quickstart-file-store-roundtrip`), which proves an end-to-end deployment lifecycle using a durable hop on local disk.

What you'll build

The workflow is a simple two-stage system:

```
echo → file-store (write) → file-store (read) → print
```

You should see:

- files created under the configured file-store path, and
- events printed to worker logs.

Prerequisites

- A **licensed** LyftData deployment with the Deployment Manager enabled.
- At least one online worker in a worker group.
- A filesystem path on the worker host that the worker process can read/write.

Step 1: Find the workflow in the Catalog

1. Open **Catalog**. 2. Filter to **Workflows**. 3. Search for **Quickstart: File Store Roundtrip** (or `quickstart-file-store-roundtrip`). 4. Choose the latest version and click **Plan & register**.

Step 2: Create the deployment

In the "create deployment" flow:

1. Give the deployment a name (for example `file-store-roundtrip-demo`). 2. Select a worker group to run it on. 3. If a wizard is shown, set:

- **File-store path**: a directory on the worker host (for example `/var/lib/lyftdata-worker/file-store`).
- **GUID prefix**: any short prefix to keep generated file names recognizable.

Step 3: Plan and apply

1. Review the **Plan** diff (jobs, transports, placement). 2. Click **Apply** to deploy the generated jobs to the selected worker group.

Step 4: Verify data flow

- Open **Workloads** and confirm the deployment is running.
- Open **Logs** in the product UI (or worker logs) and confirm events are being printed.
- On the worker host, confirm files appear under your file-store path.

Troubleshooting

- No output in logs: confirm the deployment applied successfully and the worker is online.
- File-store errors: confirm the path exists and the worker process has permissions to write to it.
- TLS/connectivity errors: verify `LYFTDATA_URL` and certificate trust for the worker group (see Networking & TLS).

Workflows

Author graphs of steps and edges, then publish immutable versions

Source: deployments/workflows.md

A **workflow** is a versioned graph of **steps** (nodes) connected by **edges** (channels and messages).

> note

Workflows are **desired state**: the Deployment Manager turns them into concrete jobs deployed onto workers.

> Note

If **Jobs** is about authoring *one job*, workflows are about composing *many jobs* into a system.

Lifecycle: draft → publish → new version

- **Draft**: editable while you iterate.
- **Validate**: checks schema and binding rules before you can publish.
- **Publish**: makes the version immutable (the version you deploy).
- **New version**: to change a published workflow, clone it into a new draft and publish again.

This “immutable published artifact” rule is the same idea used for jobs.

Perspectives vs workflows

- **Workflow**: a deployable graph once bindings and validation are complete.
- **Perspective**: a starter workflow skeleton with unbound slots. You clone/start from it, complete bindings, then publish a normal workflow version before planning deployment.

Steps: binding to building blocks

Each step is one node in the graph. A step becomes runnable when it is bound to a building block:

- **Blueprint**: a reusable module from the Catalog (connectors, adapters, patterns).
- **Workflow module**: an imported workflow that exposes an `in` / `out` interface (a reusable subgraph).

Steps have `in` and `out` bindings. Those bindings name the channels the step consumes/produces.

Edges: channels and transport kinds

Channel edges are the **data plane**: events flow along a named channel from producer steps to consumer steps.

Each edge also has a **transport kind** that controls *how* those events move (for example, fast in-memory vs durable spooling). The Deployment Manager can “stitch in” the required transport adapters when you choose a transport kind; you usually do not need to add extra steps manually.

If you pick a durable transport, the plan may include additional adapter jobs (for example “writer/reader” pairs). That is expected.

Rule of thumb:

- Use **worker channels** for low-latency hops within a worker group.
- Use **file store** or **object store** transports at stage boundaries where you need durability, replay/backfill, or blast-radius isolation.

Message edges: trigger and coordination

Message edges are the **trigger plane**: they wire “wake up” signals between steps. They are not for bulk event transport.

If you want the deeper mechanics, see Deployments Core Concepts and Messages. For how message tags/filters map into job DSL (`output.message.tag` and `input.internal-messages.filter-tag`), see Message Bus.

If you want a curated “run this now” invocation surface (UI + MCP) for message-trigger flows, see Triggers.

Starting points (Catalog workflows)

The Catalog includes end-to-end workflows you can use as starting points (for example the built-in “Quickstart: File Store Roundtrip” workflow), plus perspectives for guided architecture starters.

Typical flow:

1. Open **Catalog** and filter to **Workflows** or **Perspectives**. 2. If you selected a workflow, click **Plan & register**. 3. If you selected a perspective, use **Start from this** / **Clone to workspace**, finish bindings in the editor, then publish. 4. Provide a deployment name, select a worker group, and complete any wizard fields if the workflow provides them.