LYFTDATA PRODUCT DOCUMENTATION

Build Pipelines

Pipeline authoring, inputs, actions, outputs, scheduling, transformations, automation, and tutorials.

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

Contents

1. Pipelines Overview
2. Job Actions
3. Advanced Scheduling
4. Automating Deployments (CI/CD)
5. From Sample Job to Production Connector
6. Dealing With Time
7. Deploying Jobs
8. Enriching Data
9. Job Inputs
10. Job Outputs
11. Transform Google Analytics 4 Exports
12. Transforming Data
13. Chaining Jobs With Channels (Advanced)
14. Ingesting Local Files
15. Polling An HTTP API
16. Reading From Amazon S3
17. Variable Expansion
18. Visual Editor

Pipelines Overview

Build pipelines with LyftData

Source: build/overview.md

LyftData pipelines are authored as **jobs**: one input → zero or more actions → one output. Jobs are usually created in the UI's visual editor and executed by workers.

> tip

Need a multi-stage workflow that spans many jobs and worker groups? Use Deployments.

> Note

Build checklist

1. Pick an authoring path: the Visual editor for guided configuration or the raw YAML view for power users. 2. Shape the pipeline: review the Inputs catalog, Actions overview, and Outputs catalog to understand the primitives you can compose. 3. Test early: use **Run & Trace** plus the variable expansion guide to validate field substitutions and context. 4. Plan promotion: once a job works locally, read Deploying jobs and Advanced scheduling so staging and rollout stay predictable. 5. Keep iterating: borrow patterns from the tutorials collection and reference the DSL docs whenever you need exact syntax.

Job definition

A **job definition** is the saved configuration describing how a job should run. You author it in the visual editor (including the **YAML** tab for “pipelines as code”), and you can export it as YAML/JSON when you want to store jobs in Git.

A job definition includes:

- **Name** - A user-defined label for the Job.
- **Input** - Exactly one data source (file, S3, HTTP, etc.), with optional scheduling or triggers.
- **Actions** - Zero or more transformations, enrichments, or filters applied to data as it flows.
- **Output** - Exactly one destination to write data (file, API endpoint, cloud storage, etc.).

Job definitions live on the LyftData Server, where they are versioned and deployed to workers.

Important: A single Job has exactly one input and one output. If you need multiple inputs or outputs in a larger pipeline, use **worker channels** to compose multiple Jobs together.

LyftData enforces one input and one output per Job. If you need:

- **Multiple outputs** (e.g., archiving to S3, plus sending to a SIEM), or
- **Multiple inputs** (fan-in from various sources),

you can chain Jobs using **worker channels**.

Job types

LyftData supports several types of Jobs:

- **Once-off Jobs**: Run once, complete their task, then stop (no trigger configured).
- **Streaming Jobs**: Remain active and continuously process new data as it arrives.
- **Scheduled Jobs**: Trigger at a regular interval or according to a user-defined schedule.

Scheduling and Triggers

Scheduling in LyftData is configured within the **input**'s **Trigger** section if relevant. Some inputs don't require a schedule or trigger. Those inputs are "always on," continuously reading data as it appears.

For advanced cron, interval, and conditional patterns, see Advanced scheduling.

Job components

Each Job in LyftData consists of:

1. **Input**

- The single source that reads events (plaintext, compressed, or otherwise) from files, object storage, APIs, or any other supported data location.
- If scheduling is applicable for this input type, configure it under the **Trigger** section (e.g., a cron-like schedule or a specific time interval).

2. **Actions** (Zero or more)

- Actions process the input data-called "event data"-to filter, enrich, transform, or extract fields.
- A variety of actions is available, from simple field renaming to advanced data manipulation or enrichment with external lookups.

3. **Output**

- The single destination that writes event data (for example, to files, APIs, or cloud storage).
- If you need to send data to multiple destinations, the next section addresses **Worker Channels** which can help you compose multiple Jobs together.

Worker channels for multi-job pipelines

Because each job can only have one input and one output, you may occasionally need to **fan in** from multiple data sources or **fan out** to multiple destinations. LyftData provides **worker channels** to stitch these scenarios together.

> caution

Worker channels are **in-memory and local to a worker**. Jobs connected by a worker channel must run on the **same worker**, and in-flight events are lost if that worker restarts.

> Note

Worker channels have a **driver** configured in the worker settings, which controls delivery semantics:

- `standard`: queue semantics; if multiple jobs subscribe, events are distributed (not duplicated).
- `clone`: broadcast semantics; every subscribed job receives a copy (use this for true fan-out to multiple downstream jobs).
- `round-robin`: load distributes events across subscribed jobs in a fixed rotation.
- `rule-based`: routes events to one or more target jobs based on a field value.

In practice, you can create: 1. A job with your real data **input** and an **output** pointing to a worker channel. 2. One or more downstream jobs that read from that channel and deliver to final destinations.

This modular design gives you flexibility while keeping each individual Job simple.

For a guided example, see Chaining Jobs with Channels (Advanced).

Creating and editing jobs

1. **Open the Visual Editor** Navigate to **Jobs** in the LyftData UI and click **New Job** (or edit an existing one).

2. **Job Quick Setup Wizard**

- Choose an Input type, where the data is coming from.

- Choose an Output type, where the data is going.
- Optionally, add timestamps to all events.
- Or, skip the wizard and start with the default canvas.

3. **Configure the Job Name**

- Click on the job name at the top of the canvas to rename it.

4. **Configure the Input**

- Select a single input type (e.g., file, S3, HTTP) and fill in any required connection details by selecting "Change Input" and then "Configure".
- If scheduling is relevant (for example, you want the input to run once daily), set it up under the **Trigger** field of the input.
- When you are done press the "Close Input" button.

5. **Add Actions**

- Add, remove, or reorder actions as needed to transform or enrich your data.

6. **Specify the Output**

- Choose your single output destination.
- Fill in connection details (e.g., credentials, endpoint URLs).

7. **(Optional)** Test the Job

- Use the **Run** or **Run & Trace** buttons to run the Job with a small sample of data to verify that it's working as expected.
- The number of samples and other parameters can be configured under "Run Output".

8. **Save and Deploy**

- Review the configuration, and then save it.
- When you're ready, stage and deploy your Job to one or more workers.

Once deployed

You can monitor Jobs' status in the UI, where you'll see logs, any error messages, and performance metrics.

Job execution

When you **run** or **deploy** a job, a worker pulls the staged version from the server and starts a **job runtime** using your current context values.

- **Multithreaded Scaling**

The runtime is multithreaded and will leverage as many CPU cores as available, making LyftData highly scalable even under heavy loads.

- **Single Config, Single Runtime**

Each job execution uses one staged version at a time. If you modify a job definition and stage a new version, future runs pick up the change, but ongoing executions keep running the version they started with until you redeploy or restart them.

- **Monitoring and Logs**

You can monitor each Job's execution in the UI. Logs, error messages, and performance metrics help you diagnose issues or tune performance.

Summary

- **One Input, One Output** per Job keeps configurations straightforward.
- **Actions** let you shape and transform data en route.
- **Worker Channels** allow you to chain multiple Jobs together for more complex pipelines.
- **Scheduling** is set in the **input** configuration for inputs that support triggered runs.

For more details on available inputs, actions, and outputs, explore the related reference pages or check the examples in our Getting Started Guide.

Where to go next

- Move from samples to real data with the Day 1 production pipeline guide.
- Automate promotions and staging checks using the CI/CD workflow.
- Deep-dive into connectors via the integrations catalog.
- Learn how to monitor and scale your new jobs in the Operate section.

Job Actions

Choose and sequence LyftData actions, including AI-first pipeline patterns

Source: build/actions.md

Jobs transform events after the input parses them and before the output delivers them. Use this page to choose actions by outcome, then jump into the DSL reference for exact fields.

Choose by outcome

Outcome	Typical sequence	Edition
Normalize payloads	`json` -> `convert` -> `rename` -> `remove`	Both
Parse unstructured logs	`extract` -> `key-value` -> `assert`	Both
Enrich with lookup data	`enrich` -> `time`	Both
Build LLM features from text	`pdf-text` or `docx-to-text` -> `chunk` -> `tokenize`	Both
Generate model output in-pipeline	`chunk` -> `infer` -> `assert`	`infer` is Enterprise
Score or detect anomalies	`infer` (`anomaly-detect`) -> `scoring` -> `filter`	`infer` is Enterprise

AI pipeline patterns

Structured LLM extraction

```
actions:

  • chunk:


input-field: body output-field: chunks


  • infer:


workload: llm-completion: llm: provider: openai-compat model: your-model input-field: chunks response-
field: ai_result response-format: json prompt: system: Extract only the requested fields. schema:
'{"type":"object"}' timeout-ms: 15000 on-error: dlq:ai_failures


  • assert:


behaviour: drop-onfailure schema: schema-string: '{"type":"object"}'
```

Embeddings and clustering

```
actions:

  • infer:
```

```
workload: embedding: embedding: provider: openai-compat model: your-embedding-model input-field: text
response-field: vector


  • cluster:


input-field: vector output-field: cluster_id
```

`infer` guardrails (recommended defaults)

- Use `response-format: json` plus `prompt.schema` when downstream systems expect structured output.
- Set `timeout-ms`, `rate-limit`, and `concurrency` before production rollout.
- Configure `cache` (`namespace`, `ttl`, `max-entries`) for repeated prompts.
- Set `on-error` explicitly (`fail`, `skip`, or `dlq:name`) instead of relying on implicit behavior.
- Store provider credentials in variables (for example `${dyn|OPENAI_API_KEY}`), not inline literals.

Common actions

add

`add` creates or overwrites fields from literals, template placeholders (`{{ }}`), and runtime expansions (`${ }`).

convert

`convert` normalizes data types (string, number, datetime, boolean) and lets you define failure behavior per conversion.

filter

`filter` gates events using schema rules, pattern matches, or expressions so invalid data does not reach outputs.

enrich

`enrich` joins event fields against CSV or SQLite lookup assets and maps matched values back into the event.

Run & Trace checklist for AI jobs

- Confirm `input-field` receives the expected text payload.
- Verify `response-field` shape is stable across multiple samples.
- Inspect token usage fields when configured to estimate cost and rate-limit pressure.
- Test malformed or empty inputs to validate `on-error` behavior.
- Re-run with representative data volume to validate latency and concurrency settings.

For complete parameter details, use the DSL index and open the linked action pages.

Advanced Scheduling

Cron, interval, and message triggers for LyftData jobs

Source: build/advanced-scheduling.md

Many polling inputs in LyftData run once when `trigger` is omitted, while long-lived inputs (for example `files`, `http-server`, and `worker-channel`) run continuously. Add a `trigger` block to your input when you need recurring cadence. This guide explains each trigger type, how random offsets and run windows work, and how to test schedules safely.

Trigger variants

Cron

```
input:
echo: json: true event: "{}" trigger: cron: cron: "0 0 * * * *"
```

- Uses six fields: seconds, minutes, hours, day-of-month, month, day-of-week. `0 0 0 * * *` runs daily at midnight UTC.
- Supports `random-offset` to jitter the schedule per deployment and avoid thundering herds across large fleets.
- Optional `window` block lets you bound the run to a rolling time range (`size`, `offset`, `start`). The scheduler persists the watermark so retries or restarts pick up where the last successful window ended.

Interval

```
input:
echo: json: true event: "{}" trigger: interval: duration: 5m
```

- Accepts human-friendly durations parsed via `humantime` (`30s`, `5m`, `2h`, etc.).
- Shares the same `random-offset` and `window` options as cron, so you can stagger runs and limit processing to a moving window.

Message

```
input:
echo: json: true event: "{}" trigger: message: filter-kind: user filter-type: [user-generated] filter-tag:
stage1-done
```

- Fires when the server publishes an internal message that matches your configured filters (kind, source, job, worker, type, or tag).
- Ideal for reactive pipelines: downstream jobs ingest the control-plane event and run immediately without waiting for a timer.
- In message-triggered runs, the triggering message's public payload is available through `${msg|...}` (see Variable Expansion).

For the message model and common `filter-type` values, see Message Bus.

> **Note**: Earlier documentation referenced `condition:` blocks inside the trigger. The current schema does not accept trigger conditions; gate execution through context variables or message triggers instead.

Context-driven cadence

Job context files provide a convenient knob for operators. Reference context keys inside the trigger and update them without redeploying the job:

```
input:
http-poll: url: "https://api.example.com/events" trigger: interval: duration: "{{dynamic_interval}}"
context: dynamic_interval: "5m"
```

Changing `dynamic_interval` in the context file takes effect on the very next run.

Time macros and offsets

Runtime expansions expose the scheduler window to actions and outputs:

- `${time|now_time_secs}`, `${time|start_time_iso}`, `${time|end_time_fmt %Y-%m-%d}`
- Add offsets with `+` or `-` plus a duration: `${time|now_time_iso - 5m}` backs off five minutes; `${time|start_time_fmt %H:%M + 1h}` jumps one hour ahead.
- Invalid offsets raise runtime errors, so stick to supported units (`ms`, `s`, `m`, `h`, `d`).

Use these macros to label batches, compute backfill windows, or pass the current schedule window to downstream APIs.

Testing schedules safely

1. Configure the trigger and save your job. 2. Use **Run & Trace** in the editor to validate the pipeline without waiting for the schedule. Transient runs ignore the trigger and execute immediately. 3. Stage the job and deploy it to a non-production worker. 4. Monitor **Operate > Job status** and worker logs to confirm the trigger fires on the expected cadence. Cron entries log the next fire time; interval entries log the delay after `random-offset` is applied. 5. Once validated, deploy to production workers and keep the trigger under watch using Operate monitoring.

Operational tips

- Keep cron expressions in UTC unless you explicitly manage timezones with context or variable expansion.
- Use `random-offset` on interval jobs that rely on shared APIs to spread traffic evenly.
- When two jobs must never overlap, pair the trigger with the `window` block so each run processes a distinct, bounded slice.
- Combine message triggers with worker-channel outputs to build reactive pipelines without timers.

These scheduling primitives cover the production feature set. If your cadence requirements extend beyond cron, interval, or message triggers, contact the platform team to discuss dedicated scheduler support.

Automating Deployments (CI/CD)

Promote LyftData jobs safely with staging APIs and pipeline automation

Source: build/automation.mdx

This guide is for platform teams who want to ship LyftData jobs like code. It shows how to stage, test, and promote jobs via the CLI and integrate with a standard CI pipeline.

Prerequisites

- LyftData CLI installed locally and in your CI runners.
- API keys for staging and production servers.
- Jobs stored in Git with environment-specific context files.
- Familiarity with the Day 1 production pipeline guide.

1. Structure jobs in Git

Organise each job under `jobs/<job-name>/`:

- `job.yaml` - canonical definition (input, actions, output, triggers).
- `context.dev.yaml`, `context.prod.yaml` - overrides for staging and production.
- `README.md` - owners, SLAs, rollback instructions.

1. Maintain branches such as `main` (production) and `develop` or `staging` for pre-production. 2. Require pull requests so reviewers can check context changes and scaling impact.

2. Validate before committing

- Use your preferred linting/tests plus `lyftdata jobs import --dry-run --file jobs/<job>/job.yaml` to catch syntax issues before trying to publish changes.
- Use **Run & Trace** in the UI with sample data to sanity-check transformations end-to-end.
- Keep validation in CI so every pull request exercises it automatically.

3. Stage via CI

Typical pipeline snippet:

```
# Review what would change on the staging server
LYFTDATA_URL="$STAGING_SERVER_URL" \ lyftdata jobs import \ --file jobs/my-job/job.yaml \ --dry-run
```

```
LYFTDATA_URL="$STAGING_SERVER_URL" \ lyftdata jobs import \ --file jobs/my-job/job.yaml \ --update
```

After publishing, trigger a smoke test run through the UI (Run & Trace) or your preferred API checks before merging the change.

4. Gate production promotion

- Require a manual approval or release tag once staging checks pass.
- Import the same job definition into production after approvals:

```
LYFTDATA_URL="$PROD_SERVER_URL" \
lyftdata jobs import \ --file jobs/my-job/job.yaml \ --update
```

5. Manage secrets and context

- Store secrets in your CI manager (GitHub Secrets, Vault, etc.).
- Keep sensitive values out of YAML; reference them via context placeholders (`{{ }}` and `${ }`).
- Document merge precedence clearly: job overrides -> worker overrides -> global defaults -> job definition defaults. Test with dry runs before the first production rollout.

6. Monitor and roll back

- Watch Monitoring dashboards immediately after promotion.
- Export the job definition (`lyftdata jobs export --dir backups/jobs --dry-run` first, then run without `--dry-run`) so you can restore quickly if needed.
- Log deployment outcomes in your change management system.

CI/CD checklist

- [ ] Job definitions and context files stored in Git
- [ ] Automated dry-run `lyftdata jobs import` in CI
- [ ] Staging deployment with smoke test hook
- [ ] Manual approval or release process before production deploy
- [ ] Secrets injected via CI, not committed to the repo
- [ ] Monitoring and rollback plan documented

Related docs

- Day 1 production pipeline guide
- Deploying jobs
- Context management reference
- Operate & scale overview

From Sample Job to Production Connector

Turn the default echo job into a production-ready pipeline with real data sources

Source: build/day-one.mdx

This guide is for data engineers who have completed the Day 0 quick start and want to wire in real data. It focuses on choosing the right connector, validating transformations, and promoting a job safely.

1. Capture requirements (10 minutes)

- Business goal: what question or downstream system are you serving?
- Data source: protocol (files, object store, HTTP API, database dump), expected frequency, size, and authentication.
- Destination: target format and ingestion expectations (batch vs streaming, retention requirements).

Document these answers; they drive connector selection and batching decisions.

2. Pick the right input (10 minutes)

Use the Build catalog to choose an input:

- Object stores: S3, GCS, Azure Blob, or FileStore for on-prem.
- APIs: HTTP Poll or HTTP Server depending on push vs pull models.
- Files: Log Files input (`files`) for tailing logs or processing directories.

> tip

Many connectors share authentication and batching concepts. If you're unsure, start with the integration page for your vendor; it links directly to the relevant input/output reference.

> Note

3. Design transformations (15 minutes)

- Map input fields to the schema you need downstream.
- Identify enrichment needs (lookups, timestamps, context values).
- Select actions: Actions overview describes field edits, filters, scripts, and enrichers.
- Plan for error handling (discard vs reroute) and record outlier handling.

4. Configure the job in the visual editor (30 minutes)

1. Start with a copy of the default job or create a new job in the editor. 2. Swap the **input** for the real connector and fill required fields (bucket, keys, URL, credentials). 3. Add actions for transformations and enrichment. Use add, convert, filter, or enrich as needed. 4. Configure the **output** (S3, HTTP, file-store, etc.) with batching if required. 5. Use **Run & Trace** with sample data to validate the end-to-end flow. Adjust until the output matches expectations.

5. Handle secrets and context (10 minutes)

- Use job context values for API keys or environment-dependent settings.
- Document required environment variables and verify they are covered in staging/production.
- Reference the context management guide for merge rules and overrides.

6. Stage, test, and promote (20 minutes)

- Stage the job and deploy it to a non-production worker first.
- Validate metrics and logs after a sample run. Look for retries, error counts, or shape mismatches.

- Update runbooks with monitoring requirements (dashboards, alerts). See the Monitoring guide for key metrics.
- When satisfied, deploy to production workers and monitor closely during the first full run.

7. Share status and iterate

- Record the job's purpose, owners, and SLA in your team docs.
- Schedule periodic reviews with downstream consumers to confirm the pipeline meets their needs.
- Add lessons learned back into the build tutorials so future jobs benefit.

Quick checklist

- [ ] Requirements documented (source, destination, schedule, success criteria)
- [ ] Input connector selected and tested with sample data
- [ ] Actions configured and validated using Run & Trace
- [ ] Output batching and delivery confirmed
- [ ] Context/environment variables defined
- [ ] Job staged, promoted, and monitored in production

Once you're comfortable building manually, graduate to automation with the CI/CD guide. The actions, batching, and context patterns above form the foundation of every production job.

Dealing With Time

Normalize, format, and classify time values inside LyftData jobs

Source: build/dealing-with-time.md

Time appears in almost every pipeline. Jobs need to stamp run metadata, convert vendor timestamps, or decide whether an event falls into a window. This guide collects the canonical tools for dealing with time in LyftData so you can wire them together with confidence.

Stamp run context with field expansions

Use `${time|...}` macros when you need a timestamp derived from the job's current run window. The macros resolve before actions execute and do not require a separate step.

```
actions:

  • add:



output-fields: exported_at: "${time|now_time_iso}" start_bucket: "${time|start_time_fmt %Y-%m-
%dT%H:00:00Z}"
```

Key points:

- `now_time_*`, `start_time_*`, and `end_time_*` expose the window's `now`, `start`, and `end` timestamps.
- Append `fmt` to supply a strftime pattern: `start_time_fmt %Y/%m/%d`.
- Add offsets with `+` or `-` and a `humantime` duration: `now_time_fmt %Y-%m-%d - 5m`.
- The syntax is literal `${...}`; templating helpers such as `{{ date(...) }}` are not parsed by the runtime.

You can combine macros inside object keys, for example to produce partitioned paths:

```
output:
s3: bucket-name: lyftdata-exports object-name: name: analytics/${time|now_time_fmt %Y/%m/%d}/events.jsonl
```

Convert timestamps with the `time` action

When events carry timestamps that need normalising, reach for the `time` action. It parses strings or numbers from one or more fields and writes the result in the format you choose.

```
- time:
input-field: vendor_ts input-formats:



  • default_iso

  • '%s'



output-format: default_iso output-field: '@timestamp'
```

Highlights:

- `input-field` selects the event field; fall back to the run `now` when omitted.

- `input-format` accepts chrono/strftime specifiers or shortcuts such as `default_iso`, `epoch_secs`, and `epoch_msecs`. Use `input-formats` to provide a list.
- `input-timezone` lets you parse ambiguous local times explicitly (`America/Los_Angeles`, for example).
- `output-field` writes the converted value. The default format is ISO-8601 UTC with millisecond precision.

The action also emits metadata about the run window so you can reuse it in later steps without re-parsing.

Classify by time ranges

`time` can tag events that fall within specific intervals using `time-range-conditions`. Supply ranges using the `day:start-end` syntax and choose where to store the result.

```
- time:
input-field: '@timestamp' time-range-conditions: times:


  - 'mon-fri:09:00-17:00'

  - 'sat:10:00-13:00'


output-fields: business_hours: true time-range-output: start-field: range_start_secs length-field:
range_length_secs
```

To expose the derived range bounds, set both `time-range-output.start-field` and `time-range-output.length-field`.

Handle timezones explicitly

By default, the `time` action assumes:

- Input strings without offsets represent local time.
- Output values are UTC.

Override this behaviour with `input-timezone` and `output-timezone`, using IANA names (`Europe/Paris`, `America/New_York`). Pair these options with `utc-input: true` when upstream timestamps already include an offset but you want to treat them as UTC.

For jobs that run in containers, remember that the `TZ` environment variable controls the runtime's notion of "local". Set it in your worker context if you need a different default.

Build full timestamps from partial data

Some sources expose only a time-of-day. You can synthesise a full timestamp by stitching the current date to the captured time and sending it back through the `time` action.

```
- time:
output-field: current_date output-format: '%Y-%m-%d'


  - add:


output-fields: combined_ts: '${current_date} ${event_time}'
```

```
  • time:


input-field: combined_ts input-format: '%Y-%m-%d %H:%M:%S' output-field: '@timestamp'
```

This pattern keeps everything inside the DSL, with no external scripting required.

Use schedule-aware offsets

Schedulers drive run windows, so pairing them with `${time|...}` offsets works well for backfills or catch-up scenarios. For example, subtract five minutes when polling an API that lags slightly behind real time:

```
- add:
output-fields: fetch_from: "${time|now_time_iso - 5m}"
```

Offsets respect any random jitter or windowing configured on the trigger, ensuring downstream systems receive consistent, deduplicated slices.

Deploying Jobs

Stage, deploy, and roll back LyftData jobs

Source: build/deploying-jobs.md

Deploying a job is a two-step process:

1. **Stage** the saved job to create an immutable, deployable version. 2. **Deploy** the staged version to one or more workers.

Before staging, validate the pipeline with **Run** or **Run & Trace** in the editor. See Running a job.

> note

This page covers deploying a **single job**. If you need multi-job orchestration (workflows, durable hand-offs, placement across worker groups), start with Deployments.

> Note

Save vs stage vs deploy

- **Save**: stores the editable job definition on the server.
- **Stage**: validates the job and freezes a version for deployment (this is what workers run).
- **Deploy**: assigns the staged version to a worker and starts (or schedules) execution.

Stage a job

Jobs are saved on the server after **Save** is pressed in the editor. Closing the editor shows the job page:

Job page showing save and stage controls: ../../../assets/job-page.png

The job needs to be staged before it can be deployed to workers. Click **Stage** (or **Stage job**).

Job staged and ready for deployment: ../../../assets/staged-job.png

Staging is also where the UI surfaces configuration issues early. If the job cannot be staged, open the **Issues** panel and fix the reported warnings or errors. See Logs & Issues.

Deploy to a worker

From the job page, open the Deployments panel/tab and deploy the staged version to a worker. In Community Edition you can use the **Built-in Worker**.

Click **Deploy** (or **Add**) and wait for the job to start. The UI shows the deployment status:

Deployment status showing job complete: ../../../assets/deployment-done.png

Use the job logs and the Job status feed to confirm runs, throughput, and errors.

To stop a deployed job, remove it from the worker (the UI may show **Remove**).

Update and promote safely

When you change a job:

1. Make the change in the editor (or update the YAML in Git). 2. Run **Run & Trace** to validate the new behavior. 3. Save, then stage again to publish a new immutable version. 4. Deploy the new staged version to a non-production worker first, then promote to production.

If you automate promotions, follow the CI/CD automation guide.

Example: catching a bad field conversion

Staging and Run & Trace are designed to catch regressions before they hit production. For example, a `convert` action can warn when the input field is missing.

Convert action configuration with missing field warning: ../../../assets/converting-field.png

After adding the conversion, make sure you click **Add item** (so the rule is actually saved), then save and stage again.

If the job now has problems, the **Issues** icon turns red. Open it and review the warning details:

Issues panel highlighting job validation errors: ../../../assets/first-problem.png

These warnings also show up in the job logs, and often indicate that the pipeline is making assumptions about input data that no longer hold.

Enriching Data

Add derived fields, hashes, and lookup results inside LyftData jobs

Source: build/enriching-data.md

After you have raw events in JSON form, the next step is enrichment: stamping metadata, deriving new values, and joining with reference tables. This page walks through the built-in actions that cover those cases.

Tag events with deployment context

Use the `add` action to attach static metadata, pulling values from job context or the runtime helpers (`{{job}}`, `{{worker}}`). `add` defaults to the `unless-exists` merge strategy, so it will not overwrite fields that are already present.

```
- add:
output-fields: site: '{{job}}' worker_id: '{{worker}}'
```

Set `overwrite: true` if you need to replace an existing value.

Generate runtime fields

You can enrich events without leaving the pipeline:

```
- time:
output-field: '@timestamp'


  • script:


let:


  • seq: count()
  • event_uuid: uuid()
  • normalized_status: md5(status)
```

The Lua environment that backs the `script` action includes helpers such as `count()` counters, `md5()` hashing, `uuid()` generation, and the `cond()` ternary helper. It also exposes encryption and decryption helpers (`encrypt()`, `decrypt()`) when you need reversible protection.

Conditional fields

Choose between literal values with `script let` and `cond`, or guard an entire block with `condition`.

```
- script:
let:


  • quality: cond(a > 1, 'good', 'bad')
```

Because scripts run in place, they can both inspect and update the same event.

Table lookups with `enrich`

The `enrich` action loads lookup tables from CSV or Sqlite and matches them against event fields using typed comparisons (`str`, `num`, `ip`, `cidr`, `num-range`, `num-list`, `str-list`). When the underlying file changes, the runtime notices the new modification timestamp and reloads it automatically.

Include lookup assets under the job's `files:` so workers download them alongside the job definition.

```
name: crm-enrich
files:


  - lookups/names.csv


input: echo: json: true event: '{ "id": 12 }' actions:


  - enrich:


lookup: lookups/names.csv match:


  - type: num


event-field: id lookup-field: user_id add: event-field: full_name lookup-field: name event-fields:
nickname: '' department: 'Unknown'
```

`event-fields` provides a shorthand for adding several lookup columns at once. Each entry uses the lookup column name as the event field and the YAML value as the fallback when no match is found.

Example CSV

```
user_id,name,nickname,department
12,Alice,ac,Finance 99,Bob,b2,Sales
```

If you need richer joins, set `lookup` to a Sqlite database, or chain the `enrich` action with a `script` to post-process the lookup results.

Sqlite lookups

When a flat file is not enough, point `lookup` at a Sqlite database. Workers ship the database alongside the job files and the enrich action queries the specified table.

```
files:

  - lookups/reference.db


input: echo: json: true event: '{ "email": "alice@example.com" }' actions:
```

```
  - enrich:

lookup: sqlite: path: lookups/reference.db table: users match:

  - type: str

event-field: email lookup-field: email event-fields: role: 'guest'
```

In this pattern, `event-fields` still provides defaults for missing rows, and the runtime reloads the database file when it changes.

Job Inputs

Select and configure data sources for LyftData jobs

Source: build/inputs.md

Inputs determine how events enter a job. Every job selects exactly one input and can optionally attach a trigger to control cadence. The visual editor exposes the most common settings; the DSL reference covers the full schema for advanced scenarios.

Supported inputs

Input	Trigger style	Common uses	Notes
`azure-blob`	Scheduled poll	Ingest blobs from Azure storage accounts.	Supports wildcard object selection and incremental checkpoints.
`echo`	Manual or triggerless	Seed jobs with inline sample data.	Handy for quick tests and template jobs.
`exec`	Scheduled poll	Run a command and capture STDOUT as events.	Ship logs from legacy tools without building wrappers.
`file-store`	Scheduled poll	Read from managed FileStore buckets.	Best for hybrid deployments that mirror data into the control plane.
`files`	Continuous	Tail directories or batch uploaded files.	Detects new files and streams them line by line.
`gcs`	Scheduled poll	Pull objects from Google Cloud Storage.	Shares batching semantics with S3 and Azure connectors.
`http-poll`	Scheduled poll	Call REST APIs on a cadence.	Configure headers, query params, and request bodies; supports retries and timeouts.
`http-server`	Event-driven	Accept inbound webhooks.	Runs an embedded HTTP listener on the worker.
`internal-messages`	Event-driven	React to platform events or job-emitted messages.	Filter by kind, source, type, job, or tag.
`s3`	Scheduled poll	Read from Amazon S3 buckets.	Handles compressed objects and streams large files.
`windows-event-log`	Event-driven	Capture Windows system events.	Available on Windows workers only.
`worker-channel`	Event-driven	Chain jobs together inside a worker.	Consumes messages emitted by upstream jobs in the same worker.

If you do not see a specific connector in the UI, check your edition and licensing; some inputs are only available with certain licenses.

Scheduling and triggers

Inputs that poll external systems expose a **Trigger** block. Choose between cron expressions, fixed intervals, or internal message triggers. The scheduler guarantees that each job run receives a distinct window, and you can offset or jitter schedules to avoid thundering herds. Continuous inputs such as `files`, `http-server`, and `worker-channel` ignore the trigger block. They stay active as long as the job runtime is running.

Parsing options

Most inputs share two common switches:

- **JSON**: Treat payloads as JSON documents so fields are available without extra parsing. Leave disabled when ingesting free-form text; the runtime will wrap the payload in `_raw` instead.
- **Ignore line breaks**: Combine the entire payload into a single event. Enable when a single response spans multiple lines (for example, pretty-printed JSON from an HTTP API).

Document-oriented connectors (`exec`, `http-poll`, the object stores) group the events produced during one fetch into a *document*. Downstream actions can reference document metadata or preserve the grouping by using output batching in **Document** mode.

Reliability controls

Inputs that reach out to external services include a `retry` block. Use:

- `retry.timeout`: per-attempt timeout (defaults to 30s).
- `retry.retries`: number of retry attempts (omit for unlimited).

Most connectors apply a bounded backoff internally between attempts; keep `retries` low to avoid overwhelming an unhealthy upstream system.

For field-level schemas and advanced options, consult the DSL index and open the linked input pages.

Job Outputs

Deliver events to downstream systems from LyftData jobs

Source: build/outputs.md

Every job writes to exactly one output. Outputs stream the event payloads that actions produced, optionally batching or wrapping them before delivery. The table below summarizes the built-in connectors; follow the links for full DSL options.

Supported outputs

Output	Delivery style	Ideal for	Notes
`azure-blob`	Batched upload	Landing data in Azure storage.	Supports append or replace strategies and server-side encryption settings.
`discard`	Null sink	Measuring upstream performance without delivery.	Useful for load testing actions.
`file`	Streaming	Writing to local files on the worker.	Combine with volume mounts for on-prem delivery.
`file-store`	Batched upload	Managed FileStore buckets.	Generates stable object names and handles deduplication metadata.
`gcs`	Batched upload	Google Cloud Storage targets.	Mirrors the batching semantics of S3.
`http-get`	Request/response	Triggering downstream HTTP endpoints that expect GETs.	Rarely used; most jobs prefer `http-post`.
`http-post`	Batched POST	REST APIs and webhooks.	Configure headers, authentication, and templated bodies.
`message`	Control-plane	Broadcasting structured messages.	Downstream jobs consume them via the `internal-messages` input.
`print`	Streaming	Writing to STDOUT/STDERR.	Handy for development and demos.
`s3`	Batched upload	Amazon S3 sinks.	Supports server-side encryption, storage classes, and multi-part uploads.
`splunk-hec`	Batched POST	Sending events to Splunk HTTP Event Collector.	Automatically wraps batches according to HEC expectations.
`worker-channel`	Streaming	Chaining jobs in-memory.	Feeds the `worker-channel` input of downstream jobs.

If you need multiple destinations, emit to a worker channel configured with the `clone` driver and fan out with additional jobs (on the same worker).

Message output (control-plane)

`output.message` publishes to the internal message bus (the aggregator). Use it for coordination, wake-ups, and structured control-plane signals, not bulk data movement. Downstream jobs consume these messages via `internal-messages` or message triggers.

Important gotchas:

- The message output only emits JSON events. If your job's output event is not JSON, the published payload becomes `null`.
- `set-variable-name` stores the JSON event as a string (for example `{"ok":true}`), not as a structured object.

- Use `input-field` to emit a strict envelope (for example `trigger_response`) without mixing it into your normal output event.

See Message Bus for the message model and Triggers for response-slot envelopes and proxy capture.

Batching strategies

Outputs send events individually unless you enable batching. Choose a `mode`:

- **`fixed`**: Flush after `fixed-size` events or when the `timeout` expires. This is ideal for APIs that accept arrays or bulk uploads.
- **`document`**: Preserve the grouping generated by the input (for example, all records pulled from one file or API response). This mode keeps document metadata intact for downstream consumers.

Set **Header** and **Footer** strings to wrap each batch. With runtime variable expansion you can insert counters and timestamps (for example `${stat|_BATCH_NUMBER}` and `${time|now_time_iso}`); see Variable expansion.

Enable **Wrap as JSON** when the receiver expects a valid JSON array. The runtime adds brackets and commas automatically, so you can focus on formatting headers and footers.

Reliability and retries

Networked outputs (`http-post`, `s3`, `azure-blob`, `gcs`, `splunk-hec`) expose a `retry` block. Use:

- `retry.timeout`: per-attempt timeout (defaults to 30s).
- `retry.retries`: number of retry attempts (omit for unlimited).

Most connectors apply a bounded backoff internally between attempts; keep `retries` low to avoid overwhelming an unhealthy downstream system.

Testing outputs

Before promoting a job, run it with the **print** output in staging to inspect the exact payload and headers. Once satisfied, swap back to the production connector and stage the job. Keep the Deploying jobs guide handy for promotion workflows.

For exhaustive field documentation, use the DSL index and open the linked output pages.

Transform Google Analytics 4 Exports

Normalize GA4 exports into a stable analytics-ready schema

Source: build/transformations/transform.md

This how-to converts GA4 export files into a predictable event schema you can route to storage, APIs, or downstream jobs.

What this pipeline should produce

- One event per GA4 `events[]` item.
- A canonical `@timestamp` field.
- Consistent, flattened field names for downstream consumers.
- Optional AI-derived labels (Enterprise) for segmentation.

1. Configure the input

1. Create a job (for example `ga4-normalized`). 2. Select your object-store input (`s3`, `gcs`, `azure-blob`, or `file-store`). 3. Set the object prefix (for example `exports/ga4/daily/`) and enable fingerprinting. 4. Under response handling, split records from the `events` array.

Tip: keep replay simple by using date-based prefixes and rotating job versions rather than editing production definitions in place.

2. Baseline transformation stack

```
actions:
  - json:

input-field: data

  - expand-events:

array-field: events

  - flatten:

input-field: events separator: "."

  - rename:

fields: "events.event_params.key": param_key "events.event_params.value.string_value": param_value

  - filter:

how: expression: "events.name == 'purchase'"
```

```
  - convert:

fields: events.event_timestamp: num units:

  - field: events.event_timestamp

from: microseconds to: milliseconds

  - time:

input-field: events.event_timestamp input-formats:

  - epoch_msecs

output-field: '@timestamp' output-format: default_iso

  - add:

output-fields: dataset: "{{dataset}}" environment: "{{environment}}" source_object:
"${msg|message_content.object_name||unknown}"
```

Why this order

1. Parse first (`json` , `expand-events`). 2. Shape the schema (`flatten` , `rename` , `filter` , `convert`). 3. Add operational metadata last (`time` , `add`), including timestamp normalization.

This ordering keeps traces easier to read and reduces accidental field drift.

`events.event_timestamp` in GA4 exports is typically in microseconds, so this example normalizes it to milliseconds before using `time` with `epoch_msecs` . If your upstream payload already uses milliseconds, remove the units conversion.

3. Optional AI enrichment (Enterprise)

Use `infer` when you want automated classification (for example purchase intent, campaign grouping, or anomaly flags) without a separate service.

```
- infer:
workload: llm-completion: llm: provider: openai-compat model: your-model input-field: events.name response-
field: ai_labels response-format: json prompt: schema: '{"type":"object"}' timeout-ms: 10000 on-error: skip
```

Recommended defaults for production: set `rate-limit` , `concurrency` , and `cache` , and validate `ai_labels` shape in Run & Trace.

4. Add quality gates

Use filters for recoverable issues and assertions for hard contract failures.

```
- filter:
how: expression: "exists(events.user_pseudo_id)"


  • assert:


schema: schema-string: '{"type":"object","required":["events.event_timestamp","events.user_pseudo_id"]}'
behaviour: abort-on-failure
```

5. Run & Trace checklist

- Confirm event count after `expand-events` matches GA4 array size.
- Verify `@timestamp` conversion on real samples.
- Check renamed fields used by downstream dashboards.
- For AI steps, verify `ai_labels` remains valid JSON across multiple runs.

6. Stage, deploy, and monitor

1. Stage the job after trace validation. 2. Deploy to non-production first and monitor throughput/error rates. 3. Promote to production via your standard release path (manual or CI/CD automation). 4. Add alerts for parse failures, assertion failures, and output delivery errors.

Transforming Data

Design reliable transformation pipelines, including AI-assisted patterns

Source: build/transformations/transforming-data.md

LyftData jobs transform events between one input and one output. This guide helps you choose the right action chain, apply AI where it adds value, and harden pipelines before production.

Choose the right transformation path

Goal	Recommended action chain	Edition
Parse logs into fields	`extract` -> `key-value` -> `convert`	Both
Normalize JSON payloads	`json` -> `flatten` -> `rename` -> `remove`	Both
Join reference data	`enrich` -> `time`	Both
Prepare documents for AI workflows	`pdf-text` or `docx-to-text` -> `chunk` -> `tokenize`	Both
Generate structured model output	`chunk` -> `infer` -> `assert`	`infer` is Enterprise
Embed and cluster records	`infer` (`embedding`) -> `cluster`	`infer` is Enterprise

Sequence actions for predictable behavior

Use this order unless you have a specific reason not to:

1. Parse (`json` , `csv` , `xml` , `extract`). 2. Shape fields (`rename` , `copy` , `remove` , `flatten` , `convert`). 3. Enrich (`add` , `enrich` , `time` , optional `script`). 4. Validate and gate (`filter` , `assert` , `abort`). 5. Route signals (`message`) and deliver via output.

This keeps validation close to the final payload shape and reduces hard-to-debug side effects.

AI-assisted transformation patterns

Structured extraction with guardrails

```
actions:

  • chunk:


input-field: body output-field: chunks


  • infer:


workload: llm-completion: llm: provider: openai-compat model: your-model input-field: chunks response-
field: ai_result response-format: json prompt: schema: '{"type":"object"}' timeout-ms: 15000 on-error:
dlq:ai_failures


  • assert:
```

```
behaviour: drop-onfailure schema: schema-string: '{"type":"object"}'
```

Embedding pipeline for downstream analytics

```
actions:

  • infer:


workload: embedding: embedding: provider: openai-compat model: your-embedding-model input-field: text
response-field: vector


  • cluster:


input-field: vector output-field: cluster_id
```

Defaults, filters, and empty events

When you write line-oriented outputs (for example CSV/text) through `output.file` with `input-field`, treat missing/null/empty values explicitly:

- **Missing or null** `input-field`: LyftData emits nothing (no line is written).
- **Empty payload events** (for example events dropped/filtered earlier): LyftData emits nothing.
- **Empty strings** (`""`) are valid values: a blank line is written.

Recommended pattern:

1. Use `add` to set default values before output when a field must always exist. 2. Use `filter` to drop rows you do not want (for example empty-string rows). 3. Use `csv-stringify` to create correctly escaped CSV rows, then point `output.file.input-field` to that generated field.

Example:

```
actions:

  • add:


kv-pairs: csv_row: ""


  • csv-stringify:


fields: [name, price, category] output-field: csv_row


  • filter:


condition: "csv_row ~= ''" output: file: path: /tmp/export.csv input-field: csv_row
```

Production guardrails

- Treat `script` as an escape hatch; prefer declarative actions first.
- Add explicit conversion and validation behavior (`convert` + `assert` / `filter`) so bad events fail predictably.
- For `infer`, always set `timeout-ms`, `rate-limit`, `concurrency`, and `on-error` explicitly.
- Use `response-format: json` and a schema when model output feeds downstream systems.
- Keep secrets in variables such as `${dyn|OPENAI_API_KEY}`, never inline in job definitions.

Run & Trace checklist

- Verify each step changes only the fields you expect.
- Check field types after `convert` and `time`, not just field names.
- Test malformed inputs to validate failure behavior (`drop`, `abort`, or DLQ).
- For AI jobs, inspect response shape stability across multiple samples and confirm latency under realistic payload sizes.

For full field-level options, use the DSL index and open the linked action pages.

Chaining Jobs With Channels (Advanced)

Advanced tutorial covering fan-out and fan-in using worker channels

Source: build/tutorials/channels-advanced.md

This end-to-end tutorial builds a four-job pipeline that fans events out to two downstream jobs and then fans them back in on a shared channel. It highlights how to design channel names, configure channel drivers, deploy jobs in the correct order, and verify flow end to end.

> note

Worker channels are **local to a worker**. For this tutorial, deploy all four jobs to the **same worker**.

> Note

Scenario overview

We implement the following flow:

1. Job A ingests events from an HTTP input and publishes each record to channel `alpha`. 2. Channel `alpha` is configured with the `clone` **driver** so multiple downstream jobs receive a copy. 3. Job B (enrichment) listens on channel `alpha` and writes enriched events to channel `beta`. 4. Job C (alerting) also listens on channel `alpha` and writes alerts to channel `beta`. 5. Job D consumes `beta` and emits final records to an output (for example Elasticsearch).

Channels let you decouple workloads while keeping jobs simple—each job still has a single input and output, but you can compose multi-stage pipelines without external scripts.

Prerequisites

- Server and at least one worker online (the built-in worker is sufficient).
- Familiarity with the visual editor and staging/deployment workflow (see the build overview).

Step 1 – design the channel contract

Before building jobs, document the schema each channel carries. This avoids downstream validation errors.

Channel	Producer	Consumers	Fields
alpha	Job A	Jobs B, C	event_id, ts, raw_payload
beta	Jobs B, C	Job D	event_id, ts, stage, payload

Store this table in your runbook or a shared document so future contributors know which fields are available.

Step 2 – configure the channels on the worker

1. In the UI, open **Workers** and select the worker you will deploy these jobs to. 2. Edit worker settings and add two channels:

- `alpha` with driver `clone` (broadcast fan-out).
- `beta` with driver `standard` (single shared channel for fan-in).

If you skip this step, jobs that use `worker-channel` will fail with a “worker channel not found” issue.

Step 3 – build Job A (producer)

1. Create a new job named **channel-producer**. 2. Choose an HTTP input (or another source) and configure authentication. 3. Add any necessary actions (for example, parse JSON). 4. Set the output to **Worker Channel** with channel ID `alpha`. 5. Save the job, close the editor, and stage it.

You can verify the payload shape by using the **Run Output** tab—confirm the fields match the `alpha` contract.

Step 4 – build Jobs B and C (parallel consumers)

Create two jobs based on the channel contract:

- **channel-enrich**

1. Input: Worker Channel `alpha`. 2. Actions: add geographic enrichment, compute scores, or call external APIs. 3. Add a field such as `stage: "enrich"` so downstream jobs can distinguish records. 4. Output: Worker Channel `beta`.

- **channel-alert**

1. Input: Worker Channel `alpha`. 2. Actions: filter on severity, map to alert levels. 3. Add a field such as `stage: "alert"` so downstream jobs can distinguish records. 4. Output: Worker Channel `beta`.

Use the **Preview** tab for each action to ensure Job B and Job C emit the expected fields. When both jobs stage successfully, deploy them to workers that have capacity for the new workload.

Step 5 – build Job D (fan-in)

Job D combines the outputs from Jobs B and C by consuming the shared `beta` channel.

Create a job named **channel-aggregate** with an **Input: Worker Channel** set to `beta`.

Finish configuration:

1. Use `filter` or conditional logic based on `stage` to handle enriched vs alert records differently (if needed). 2. Add business logic or scoring based on enriched and alert data. 3. Set the output to your destination—e.g., Elasticsearch, Splunk, or S3.

Sample YAML for the worker-channel input:

```
input:
worker-channel: worker-channel-name: beta
```

Stage Job D but wait to deploy until Jobs A–C are running to avoid empty channel warnings.

Step 6 – deploy in order

Deploy all four jobs to the same worker:

1. Deploy **channel-producer** and confirm the worker shows the job as running. 2. Deploy **channel-enrich** and **channel-alert**; watch worker logs for channel subscription messages. 3. Deploy **channel-aggregate** once `beta` shows new events. 4. Trigger sample traffic (curl, replay file, etc.) and verify the final output destination receives the expected records.

Monitor the **Issues** panel and worker logs throughout deployment. Channel mismatch errors typically indicate a schema drift between the jobs.

Troubleshooting

- **Job D never receives events**: confirm Jobs B and C publish to the exact channel IDs listed in the contract. Channel IDs are case sensitive.

- **Jobs B/C split events instead of both receiving them**: ensure channel `alpha` is configured with driver `clone` (broadcast). With `standard` or `round-robin`, events are distributed across subscribers.
- **Validation errors in Jobs B/C**: use **Preview** for the last action in Job A to confirm the payload contains the fields referenced downstream.
- **No events flow between workers**: worker channels are local to a worker; deploy the connected jobs together.
- **Lost events after restart**: channels are in-memory; for guaranteed delivery, persist to a queue or storage service before fan-out.

Validate and monitor

1. Use **Run & Trace** on **channel-producer** to capture a live payload and confirm the channel contract before deployment. 2. Stage and deploy Jobs A-D in order, then confirm they remain **Running** with expected event throughput under **Operate > Job status**. 3. Generate synthetic traffic (curl, replay, or QA fixtures) and monitor worker logs for channel subscription updates and aggregate outputs.

After validation, you can:

- Add automated tests or synthetic traffic to continuously verify the multi-job pipeline.
- For cross-worker isolation, switch to a durable transport between stages (for example Kafka or object storage); worker channels remain single-worker only.
- Combine these checks with reference/troubleshooting when new failure modes appear.

Fold the channel topology into Operate daily operations and configure health alerts using Operate monitoring.

Ingesting Local Files

Configure the files input for one-off imports and continuous directory monitoring

Source: build/tutorials/files.md

The `files` input tails a directory on the worker and turns each line into an event. This tutorial shows how to import an existing batch of files, how to switch the job into watch mode, and how to avoid reprocessing data when the worker restarts.

Prerequisites

- A worker with access to the directory you want to ingest (local path or mounted volume).
- Sample files containing one JSON record per line.
- Access to the Jobs visual editor.

1. Build a one-off import job

1. Create a new job named `files-import` and choose **Files** as the input. 2. Set **Path** to the directory or glob pattern that should be scanned, for example `C:/data/logs/*.json`. 3. Enable **JSON** so the runtime parses each line into fields instead of wrapping it in `_raw`. 4. Enable **Stop reading after**. This tells the input to exit once every matching file has been processed, which is perfect for backfills. 5. (Optional) Set **File path field** to `source_path` if you want the event to include the file name. Use **File basename** to keep only the leaf name.

`files` fingerprints every object it reads (path plus metadata), so rerunning the job later will skip files it has already processed unless you explicitly reset the fingerprints.

2. Add transformations and an output

Attach the actions and output you need for your downstream system. For example:

```
actions:

  • add:


output-fields: ingested_at: "${time|now_time_iso}" output: print: output: stdout
```

You can replace the Print output with S3, Splunk HEC, or another sink once validation is complete.

3. Validate with Run & Trace

Use **Run & Trace** to execute the job once. The UI sends the current definition to `/api/jobs/run`, so the worker runs it transiently and returns each event and its trace. Confirm the expected number of files and records arrive, and inspect the metadata fields you added.

If you need to rerun the import from the top, set **Start at beginning** to true. Otherwise the fingerprints make repeat runs idempotent.

4. Switch to continuous monitoring (optional)

To turn the job into a directory watcher instead of a one-off import:

1. Disable **Stop reading after** so the runtime keeps listening for new files. 2. (Optional) Reduce **Run time limit** or set an **Output event limit** during transient runs so tests stop promptly. 3. Stage and deploy the job. Whenever a new file matching the glob appears, the worker processes it once and records its fingerprint.

5. Stage, deploy, and monitor

1. Save, close the editor, and click **Stage job**. 2. Deploy to a worker. The job immediately processes any unseen files, then idles while waiting for new ones. 3. Monitor progress in **Operate > Job status** and inspect worker logs if files are skipped. Most often the fingerprints show the file was already processed or the glob did not match.

Operational tips

- Keep the fingerprint database (stored alongside the worker state directory) when upgrading workers so you do not reprocess old files by accident.
- Use Advanced scheduling with a message trigger if another pipeline should signal when to scan a directory.
- Pair the job with the Dealing with time guide to stamp ingestion timestamps or derive partitions for downstream storage.
- Follow the runbook patterns in Operate monitoring to alert when file backlogs build up or when the job encounters repeated read errors.

Polling An HTTP API

Build a resilient job that calls an HTTP endpoint and captures the response

Source: build/tutorials/http.md

This tutorial walks through a minimal `http-poll` job that calls a public API, parses the JSON response, and prepares the pipeline for promotion. The same pattern applies to internal services once you swap the URL and credentials.

Prerequisites

- A running LyftData server with at least one worker attached.
- Outbound network access from the worker to `https://httpbin.org` (or your real API).
- An operator account with access to the Jobs visual editor.

1. Create the job and configure the request

1. Open **Jobs** in the UI and click **New job**. Name it `http-sample`. 2. In the visual editor, select the **Input** panel and choose **HTTP Poll**. 3. Set **URL** to `https://httpbin.org/anything`. Leave **Method** as `GET` for now. (`http-poll` also supports POST/PUT/PATCH/DELETE plus custom methods, headers, query parameters, basic auth, bearer tokens, or API keys.) 4. Expand the **Response** section and set:

- **Body field** to `response_body` so the payload lands in that field.
- **Status field** to `response_status` if you want to store the HTTP status code.
- Enable **JSON** and **Ignore line breaks** because `httpbin` returns multi-line JSON. The runtime honours these flags when parsing responses and emits a single event per request.

5. (Optional) Add headers, query parameters, or a request body to mimic your production call. The `http-poll` input merges all of these options into the outgoing request each time the trigger fires.

At this point the job issues a GET request and records the raw response body and status in the event payload.

2. Parse the response

Add actions to shape the HTTP payload:

```
- json:
input-field: response_body


  • remove:


fields:


  • response_body
```

This pipeline parses the JSON body into event fields (for example `json.args`, `json.headers`) and removes the raw string once it is no longer needed. You can now add filters, enrichments, or variable expansion just like any other job.

3. Choose an output for validation

While testing, point the job at the **Print** output so you can inspect events in the Run & Trace panel:

```
output:
```

```
print: output: stdout
```

When you are ready for production, swap the output for S3, HTTP POST, Splunk HEC, or another destination.

4. Test with Run & Trace

Click **Run & Trace** in the editor. The UI sends the current job definition to `/api/jobs/run`, so the worker executes it once without staging. Inspect the trace to confirm the request URL, status code, and parsed fields look right. Adjust headers or actions until the event matches your downstream schema.

5. Schedule the polling cadence

Back in the input configuration, open **Trigger** and choose:

- **Interval** with a value such as `5m` for simple polling, or
- **Cron** with a six-field expression (seconds, minutes, hours, day-of-month, month, day-of-week) when you need precise alignment.

`http-poll` also honours message-based triggers, so another pipeline can publish an internal message to fire this job on demand.

6. Stage, deploy, and monitor

1. Save the job, close the editor, and click **Stage job**. 2. Deploy to a worker and confirm it shows **Running** under **Operate > Job status** once the trigger fires. 3. Watch worker logs for `http-poll` status entries. Retries follow the `retry` settings you configured on the input (timeout + retries). 4. When the pipeline is stable, replace the Print output with your production sink and follow the promotion steps in Deploying jobs.

Next steps

- Add authentication using headers or bearer tokens before pointing at a real API.
- Capture response headers by setting the **Headers field**; they arrive as a JSON object alongside the body and status.
- Combine with the guidance in Advanced scheduling and Dealing with time for backfills or sliding windows.
- Add alerting using the run health patterns in Operate monitoring.

Reading From Amazon S3

Configure S3 polling jobs with listing, filtering, and fingerprinting

Source: build/tutorials/s3.md

Object-store inputs (S3, GCS, Azure Blob, and FileStore) all share the same capabilities. This tutorial shows how to configure the S3 variant to list objects, download payloads, and avoid reprocessing files.

Prerequisites

- S3 credentials with read access to the bucket you want to ingest. Access key, secret key, and (if required) a session token or role ARN.
- A worker that can reach the S3 endpoint. When using MinIO or another compatible service, provide its custom endpoint URL.
- Familiarity with the Jobs visual editor.

1. Create the job and connect to S3

1. Create a new job named `s3-import` and choose **S3** as the input. 2. Fill in **Endpoint** (leave blank for AWS), **Bucket**, **Access key**, and **Secret key**. Add a **Session token** or **Role ARN** if your environment requires them. 3. Set **Object names** to the prefixes you want to read, for example `logs/2025/` or `exports/`. S3 inputs treat these as prefix matches rather than glob patterns. 4. Choose a **Mode**:

- **List objects** emits one event per object with metadata only.
- **Download objects** fetches the payload for each name you specify exactly.
- **List and download** combines the two: it lists matching objects, filters them, then downloads each remaining object.

The runtime fingerprints processed objects, so repeat runs skip files that were already seen unless you clear the fingerprint cache.

2. Refine the candidate set

When you list or list-and-download, you can narrow the matches without changing the prefix:

- **Include regex**: only keep object names that match one of the regular expressions.
- **Exclude regex**: drop any object whose name matches the pattern.
- **Maximum age**: ignore objects older than the configured number of seconds.

These filters run after the initial prefix scan, making it easy to pull subsets (for example `*.jsonl` files inside a daily partition).

3. Handle payload formats

Downloads can be preprocessed automatically:

- Enable **Ignore line breaks** to treat the entire object as a single event instead of splitting by newline.
- Set **Preprocessors** to `gzip`, `parquet`, `base64`, or `extension` depending on the file type. The runtime streams gzip transparently and loads Parquet files into JSON events once the download completes.
- Use **Events field** when the payload is a JSON array and you want each array entry emitted as a separate event.

Combine these settings with downstream actions (for example `json`, `csv`, or `enrich`) to normalise the data before delivery.

4. Add actions and an output

Attach any transformations you need, then send the events to a suitable destination. During validation the **Print** output makes it easy to inspect the payload; in production you might forward to S3, Splunk HEC, or a worker channel.

5. Test with Run & Trace

Use **Run & Trace** to execute the job once and confirm the expected objects appear. The UI streams each event and its trace so you can verify metadata fields such as `object_name`, `object_size`, or custom headers. Adjust filters and preprocessors until the sample run matches your expectations.

6. Schedule and deploy

Most S3 jobs rely on the **Trigger** block to poll on a cadence:

- Choose **Interval** (for example `15m`) to run at a fixed frequency.
- Choose **Cron** when you need runs at the top of the hour or another precise schedule.
- Use message triggers to kick off ad-hoc replays from another pipeline.

After testing, save, stage, and deploy the job. Watch **Operate > Job status** to confirm new objects are processed and previously fingerprinted keys are skipped.

Operational tips

- Keep the fingerprint database persistent between worker restarts so historic objects are not re-downloaded by accident.
- Pair the job with Advanced scheduling when coordinating across multiple buckets or regions.
- Use Dealing with time to stamp ingestion timestamps or convert vendor time formats during processing.
- Follow the alerting guidance in Operate monitoring to surface repeated download failures or credential issues.

Variable Expansion

Template events, context, and metadata inside jobs

Source: build/variable-expansion.md

LyftData jobs ship with two families of placeholders. Double curly braces (`{{ }}`) resolve before the job ever runs, while dollar curly braces (`${ }`) resolve at runtime for each event. Using the right style in the right place keeps pipelines portable across environments.

Context placeholders (`{{ }}`)

Context values render when you stage or deploy a job. Effective precedence (highest to lowest) is:

1. Job overrides from the control plane (`/api/contexts/update-job-map/{job}`). 2. Worker-level overrides (worker context). 3. Global defaults. 4. Job definition defaults in the job's own `context:` block.

Reference nested maps with dot notation (`{{credentials.api_key}}`). Missing keys cause staging to fail, so provide defaults in context files rather than relying on silent fallbacks. Use `{{ }}` for non-sensitive constants such as dataset names, feature toggles, and environment-specific endpoints. For secrets, prefer runtime lookups like `${secret|scope/name}` (or `${dyn|NAME}`) so you do not bake secret material into staged job YAML.

Runtime expansions (`${ }`)

Runtime expressions evaluate inside the worker after an event arrives. Many expansions support an optional `||fallback` suffix (for example `${field||default}` or `${dyn|NAME||default}`). Secret references (`${secret|scope/name}`) do not support defaults.

Syntax	Source	Example						
`${field}`	Current event	`${client.ip		0.0.0.0} `pulls` client.ip` or substitutes a default.				
`${msg	path}`	Message payload	`${msg	job_event.invocation_id}` when triggered by a user-generated message (for example a Trigger invocation).					
`${dyn	NAME}`	Variables	`${dyn	region		us-east-1}` uses the variable or falls back.		
`${secret	scope/name}`	Secrets	`${secret	pki/example.ca_pem}` references a stored secret.				
`${cred	id		field}`	Managed credentials	`${cred	graph-prod		access_token} `reads a credential field. Add		fallback` to make it optional.
`${stat	_BATCH_NUMBER}`	Output metadata	Stamp batch counters when batching					

			is enabled.	
`${time	...}`	Scheduler clocks	`${time	start_time_iso - 5m}` offsets the run window.

Event data

`${field}` reads values from the current event using dot-paths (`${client.ip}`) or slash-paths (`${client/ip}`). For strict JSON Pointer selectors, start with `/` (for example `${/client/ip}`). Supply fallbacks for optional keys to keep streaming jobs healthy:

```
- add:
output-fields: hostname: "${server||unknown-host}" disk_usage_pc: "${metrics.disk_usage_pc||0}"
```

Message payloads

Jobs that run on a **message trigger** can read the triggering message's public payload with `${msg|...}` (see Advanced scheduling). The available fields depend on the message type.

If you use the `internal-messages` input instead of a message trigger, the message wrapper is delivered as the event. In that case, use normal event expansions (for example `${message_data.job_event...}`) rather than `${msg|...}` . See Message Bus.

For user-generated messages emitted by the `message` **output**, the original event is available under `job_event` :

```
- add:
output-fields: upstream_order_id: "${msg|job_event.order_id||unknown}" upstream_tag: "${msg|tag||}"
```

For Trigger invocations (from the Triggers registry), the dispatched payload is also a `user-generated` message. The invocation details are available under `${msg|job_event...}` :

```
- add:
output-fields: invocation_id: "${msg|job_event.invocation_id}" requested_by:
"${msg|job_event.caller.username}" param_env: "${msg|job_event.params.env||}"
```

Dynamic variables

`${dyn|NAME}` resolves values from the Variables service. If you reference a variable without a default, LyftData expects it to exist. Staging and transient runs reject missing required variables early; long-lived deployed jobs can stall until the variable is provided. Use `||` defaults for optional settings.

Batch metadata

Outputs that batch events expose two counters:

- `${stat|_BATCH_NUMBER}` - zero-padded batch index.
- `${stat|_SCHEMA_NUMBER}` - schema revision when schema tracking is enabled.

Use these to build stable object keys:

```
output:
s3: bucket-name: logs-{{environment}} object-name: name: "runs/${time|now_time_fmt
%Y/%m/%d}/run-${stat|_BATCH_NUMBER}.json"
```

Time macros and offsets

`${time|...}` expressions read from the scheduler window. Common variants include:

- `${time|now_time_secs}` - current wall clock in epoch seconds.
- `${time|start_time_iso}` / `${time|end_time_iso}` - window bounds in ISO-8601.
- `${time|now_time_fmt %Y-%m-%d}` - custom `strftime` formatting.

Append `+ 5m`, `- 1h`, or any `humantime` duration to apply offsets. Invalid offsets fail the run, so stick to supported units (`ms`, `s`, `m`, `h`, `d`).

Putting it together

A typical production job mixes both placeholder families:

```
context:
dataset: security bucket: logs-{{environment}}


input: http-poll: url: "https://api.example.com/{{dataset}}/events?since=${time|start_time_iso - 5m}"


actions:



  - add:



output-fields: request_id: "${dyn|request_id}" source_host: "${hostname||unknown}"


output: s3: bucket-name: "{{bucket}}" object-name: name: "${stat|_BATCH_NUMBER}/${time|now_time_fmt
%Y/%m/%d}/events.json"
```

Tips for reliable templates

- Use `||` fallbacks on `${ }` expressions fed by optional fields or third-party payloads.
- Keep context files in source control so reviewers can verify how `{{ }}` placeholders resolve across environments.
- For long-lived deployed jobs, `${dyn|...}` without a default can stall processing when a required variable is absent; monitor worker logs for "waiting for variable" warnings.
- Batch counters reset when an output restarts. Treat them as per-run identifiers rather than global sequences.

For additional operators (math helpers, string casing, metadata keys), search this docs site for `${` examples and cross-check with the in-product DSL documentation (Docs → Job DSL).

Visual Editor

Build, test, and stage pipelines from the visual canvas

Source: build/visual-editor.md

Visual editor canvas showing input, actions, and output panels: ../../../assets/visual-editor.png

The Visual Editor is the fastest way to build and test a pipeline (job). You configure an input, chain actions, and pick an output — then validate the event flow with **Run** or **Run & Trace** before staging and deploying.

Editor layout

- **Input** (left): where events come from. Pick the connector, parsing options, and (when supported) triggers. See the Inputs catalog.
- **Actions** (middle): transformations and enrichment steps applied in order. See Actions and Transforming data.
- **Output** (right): where processed events are delivered. See the Outputs catalog.

Typical workflow

1. Create (or clone) a job and give it a stable name early. 2. Configure the input, then add actions one by one. 3. Use **Run** for a quick smoke test, then **Run & Trace** to inspect how each action changes events. See Running a job. 4. Fix warnings surfaced in the editor or the **Issues** panel. See Logs & Issues. 5. Save, then **Stage** to create an immutable version for deployment. 6. Deploy the staged version to a worker (or automate it via CI/CD automation).

YAML tab (pipeline as code)

Switch to the **YAML** tab to work directly with the underlying job definition. The editor renders the current pipeline as YAML, validates your changes as you type, and lets you copy or download the definition when you need to share it. Internally, the job is stored as JSON, but the UI round-trips through YAML for readability.

yaml tab: ../../../assets/raw-job.png

With very large jobs the YAML editor can be faster than the visual canvas for precise adjustments to inputs, actions, or outputs. It is also the easiest way to review diffs in Git when you export jobs and treat them like code.