LYFTDATA PRODUCT DOCUMENTATION

Reference Core

Configuration, licensing, commands, scripting, data handling, troubleshooting, and core reference pages.

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

Contents

CLI

Common CLI commands for operators and job authors

Source: reference/cli.md

LyftData includes a CLI for running the server in evaluation mode, administering the control plane, and automating common workflows.

Connect to a server

The CLI targets a server URL (HTTPS by default). Set it once per shell:

- Linux/macOS: `export LYFTDATA_URL=https://server-host:3000/`
- Windows (PowerShell): `$env:LYFTDATA_URL = "https://server-host:3000/"`

Then authenticate:

```
lyftdata login admin
```

If the server is using the default self-signed certificate, re-run with `--tls-insecure` (or set `LYFTDATA_TLS_INSECURE=true`) until you install a trusted certificate.

Common commands

- Help: `lyftdata --help`, `lyftdata <command> --help`
- Version: `lyftdata --version`
- Run components: `lyftdata run server`, `lyftdata run worker`
- MCP server (for assistants): `lyftdata mcp-server`
- Jobs: `lyftdata jobs …`
- Workers: `lyftdata workers …`
- Deployments: `lyftdata deployments …`
- GitOps: `lyftdata git-ops …`
- Diagnostics: `lyftdata diag`, `lyftdata doctor`

External worker hosts

For dedicated worker hosts, use the worker-only binary:

- `lyftdata-worker --help`
- Windows service management: `lyftdata-worker service install --help`

See Workers and your OS-specific worker guide for full setup steps.

Community Edition

Community Edition limits, behavior, and upgrade path

Source: reference/community-edition.md

> note

Review Licensing if you expect to move from Community Edition to a paid tier; it explains how to request keys and automate activation.

> Note

LyftData exposes two editions in the product UI:

- **Community**: constrained surfaces and limits for evaluation, local development, and CI.
- **Licensed**: production-oriented surfaces and limits based on your license claims.

Runtime tiering includes:

- **Free Community**: no license configured.
- **Community Plus**: a configured license that still uses Community Edition surfaces/caps.
- **Licensed**: a configured Licensed Edition key.

Community edition at a glance

Community Edition surfaces/caps are active when no license key is configured, and also for configured Community-tier keys.

- **No activation required** – the system recognizes the missing license and treats it as a valid Community Edition state.
- **One worker** – the built-in worker is always available; external workers cannot be added or enrolled and the built-in worker cannot be removed.
- **15 job limit** – job creation and cloning calls are capped at 15 total jobs (existing jobs can still be edited or redeployed).
- **Daily volume cap** – processing is rate-limited to approximately 250 GB/day in Community Edition.
- **Feature gating** – licensed-only connectors and capabilities remain disabled in Community Edition.
- **Core API availability** – control-plane APIs remain available so dashboards and automation can operate in constrained mode.

Why community edition exists

Community Edition gives teams a frictionless way to explore LyftData and to run lower-scale automation locally:

- **Try the platform quickly** – no keys or activation steps are needed before creating jobs and running them.
- **Local & CI usage** – ideal for test suites, proofs of concept, or developer environments that only need the built-in worker.
- **Seamless migration** – once you activate a licensed key, Community limits are lifted without rebuilding your existing jobs.

Feature comparison

Capability	Community Edition	Licensed Deployment
License requirement	None (Free Community) or Community-tier key	Valid Licensed Edition key
Worker management	Built-in worker only	License-defined (external workers allowed)
Job count	15 jobs	License-defined
Daily processing	Approx. 250 GB/day cap	License-defined
Feature access	Licensed-only connectors/capabilities are disabled	License-defined
API surface	Core control-plane APIs available	Full
Support	Community-driven	Enterprise support options

Operating in community edition

When a server is running in Community Edition you will see the following behaviours:

- License and features views in the UI show Community surfaces/tiering.
- Job creation requests return a descriptive error once the 15-job limit is reached.
- Job cloning is also blocked once the job cap is reached.
- Worker APIs reject attempts to add or enroll additional workers and refuse to remove the built-in worker.
- Throughput is rate-limited against the Community daily-volume cap.

For automation and health checks, prefer `/api/features` (`license_status.edition` and `license_status.tier`) plus `/api/license/show` details, rather than assuming “license configured” means unrestricted.

Upgrading to a licensed deployment

Switching out of Community Edition is straightforward:

1. **Acquire a license key** from LyftData. 2. **Activate the license** in **Settings** → **License** or provide `LYFTDATA_LICENSE` at startup for automation. 3. **Refresh** – as soon as activation succeeds, Community Edition limits are lifted (extra workers can be added, job limits disappear, and Community rate limiting is disabled). 4. **No reconfiguration required** – jobs, history, and settings remain intact.

Monitoring & best practices

- **Detect edition/tier explicitly** – call `GET /api/features` and inspect `license_status.edition` (`Community` or `Licensed`) plus `license_status.tier` (`FreeCommunity` , `CommunityPlus` , or `Licensed`) instead of assuming `license_configured=true` means unrestricted.
- **Plan for limits** – watch job counts and worker requirements; upgrade before hitting caps.
- **Observe volume** – add alerts around daily quota consumption in CE environments.
- **Document your upgrade process** – because activation is seamless, most teams just need a runbook describing where the license key lives and who can apply it.

Community Edition is ideal for learning and early pipeline development. When you need external workers, higher throughput, or licensed-only integrations, move to a Licensed Edition deployment.

Configuration

Server, Worker, and environment configuration reference

Source: reference/configuration.md

This page lists commonly used flags and environment variables for the server, workers, and CLI. Values can be set via CLI flags or environment variables; environment variables are shown in ALL_CAPS.

Server (`lyftdata run server`)

LyftData serves HTTPS by default (self-signed if you do not provide TLS material). Most production deployments configure a trusted certificate or terminate TLS behind a reverse proxy—see Networking & TLS.

Setting	Flag	Env	Default	Notes
Bind address	`--bind-address`	`LYFTDATA_BIND_ADDRESS`	`127.0.0.1:3000`	Use `0.0.0.0:3000` to accept remote connections.
Disable TLS	`--disable-tls`	—	off	Serve HTTP only (use behind a TLS-terminating reverse proxy, or for local-only eval).
TLS certificate	`--tls-cert`	`LYFTDATA_TLS_CERT`	—	Requires `--tls-key` .
TLS private key	`--tls-key`	`LYFTDATA_TLS_KEY`	—	Requires `--tls-cert` .
Staging directory	`--staging-dir`	`LYFTDATA_STAGING_DIR`	OS default	Recommended for services; stores control-plane state and logs.
Accept EULA non-interactively	—	`LYFTDATA_LICENSE_EULA_ACCEPT`	—	Set to `yes` , `true` , or `1` on first run to avoid prompts in automation.
Admin bootstrap password	`--admin-init-password`	`LYFTDATA_ADMIN_INIT_PASSWORD`	unset	If unset, the server enters **Initial Setup Required** and writes `<staging-dir>/bootstrap/initial-admin.url` . On a local TTY the ready splash may also show the full setup URL.
Retention (days)	`--db-retention-days`	`LYFTDATA_LOG_RETENTION_DAYS`	`30`	Cleanup applies when retention or disk thresholds are reached.
Disk usage threshold (%)	`--disk-usage-max-percentage`	`LYFTDATA_DB_DISK_USE_MAX_PERCENT`	`80`	Cleanup attempts begin when disk usage exceeds the threshold.
Bootstrap license	`--bootstrap-license`	`LYFTDATA_LICENSE`	unset	Apply a license non-interactively at startup (automation/scripted installs).

For the full server flag catalog, see `lyftdata run server --help` .

Worker (`lyftdata-worker`)

External workers connect to the server using `LYFTDATA_URL` and authenticate using either a pre-issued API key or auto-enrollment. The worker binary defaults to `http://localhost:3000`, but most deployments should set an explicit `https://...` server URL.

Setting	Flag	Env	Default	Notes
Server URL	`--url`	`LYFTDATA_URL`	`http://localhost:3000`	Worker default is `http://localhost:3000` for local compatibility. In most setups set `https://…`; use `http://…` only when the server runs with `--disable-tls`.
Worker ID	`--worker-id`	`LYFTDATA_WORKER_ID`	unset	If unset, the worker can persist identity locally after enrollment.
Worker API key	`--worker-api-key`	`LYFTDATA_WORKER_API_KEY`	unset	Required for pre-issued auth; keep it secret.
Auto-enrollment key	`--auto-enrollment-key`	`LYFTDATA_AUTO_ENROLLMENT_KEY`	unset	Alternative to pre-issued API keys; use only on trusted networks.
Worker name	`--worker-name`	`LYFTDATA_WORKER_NAME`	unset	Human-friendly label for dashboards.
Jobs directory	`--worker-jobs-dir`	`LYFTDATA_JOBS_DIR`	OS default	Recommended for services; stores cached job configs and local state.
Startup state reset	`--reset-state`	—	unset	One-time troubleshooting reset mode: `auth`, `repair`, or `full`.
Insecure TLS (evaluation only)	`--tls-insecure`	`LYFTDATA_TLS_INSECURE`	off	Accept self-signed certificates until you install a trusted certificate.

For the full worker flag catalog, see `lyftdata-worker --help`.

Master keys (secrets encryption)

LyftData uses master keys to encrypt sensitive state at rest (for example: Variables, credentials, and worker settings). On developer desktops, you can often use the OS keyring. In headless environments (servers, CI, Docker), keyring calls may fail (commonly with DBus/Secret Service errors), so configure env-backed keys explicitly.

Generate 32 random bytes (64 hex chars) for each key:

```
openssl rand -hex 32
```

Purpose	Env	Used by	Notes
Variables encryption	`LYFTDATA_VARIABLES_MASTER_KEY_SOURCE` + `LYFTDATA_VARIABLES_MASTER_KEY`	server	Set `*_SOURCE=env` in headless environments to avoid keyring/DBus issues.
Credential manager encryption	`LYFTDATA_CREDENTIAL_MANAGER_MASTER_KEY_SOURCE` + `LYFTDATA_CREDENTIAL_MANAGER_MASTER_KEY`	server	Required when the credential manager feature is enabled (default in release builds).
Worker settings / credential transport	`LYFTDATA_SETTINGS_MASTER_KEY_SOURCE` + `LYFTDATA_SETTINGS_MASTER_KEY`	built-in worker + external workers	Required in headless environments because the built-in worker runs inside the server process.

- Master keys can be provided as 64-character hex or base64-encoded 32 bytes.
- Keep these values secret and do not bake them into container images.
- For Docker/Compose examples wiring these variables, see Docker and Docker Compose.

Notes

- Prefer setting secrets via environment files managed by your service manager.
- Keep ports firewalled to trusted networks; see Security hardening.
- For troubleshooting and quick checks, see Troubleshooting.

Context Management

Manage variables and configuration context for dynamic, reusable pipelines

Source: reference/context-management.md

Context management

Context variables let you reuse the same job definition across environments without copying and editing YAML for every deployment. LyftData resolves context values before a job runs, so inputs, actions, and outputs can pull in credentials, destinations, or environment-specific toggles with `{{variable}}` placeholders.

What is context?

A context is a key-value map that the control plane merges into a job definition at deploy time. Any field that supports interpolation can reference a context value with `{{name}}`. Context can contain strings, numbers, booleans, arrays, or JSON objects.

Why use context?

- Parameterise jobs for staging, production, or specific customers without duplicating pipelines.
- Centralise shared configuration (API tokens, bucket names, feature toggles).
- Override values for a single worker or job when you need host-specific tweaks.
- Roll out changes safely by editing context instead of redeploying code.

Context scopes and precedence

LyftData layers multiple scopes when building the runtime configuration. The most specific scope wins when the same key appears in more than one place:

1. **Job overrides (control plane)** – Values saved for a job inside the UI or via `/api/contexts/update-job-map/{job}`. Highest precedence. 2. **Worker overrides** – Values attached to a worker ID through the worker detail page or `/api/contexts/update-worker-map/{worker}`. Override global defaults and job definition values. 3. **Global defaults** – Environment-wide settings managed on the Context workspace or with `/api/contexts/update-global`. Used when no job or worker override is present. 4. **Job definition defaults** – The `context:` block stored with the job YAML. Acts as the final fallback.

This merge order matches how LyftData builds the runtime configuration for a deployment.

Manage context in the UI

Global defaults

1. Sign in to the LyftData UI with an administrator account. 2. Open **Context** from the navigation sidebar. 3. Add, edit, or remove keys. Save changes to publish them immediately to all jobs. 4. Use the export and import controls on the page to back up or restore global context JSON.

Job-specific overrides

1. Open **Jobs** and select the job you want to adjust. 2. Switch to the **Context** tab in the job editor. 3. Add overrides for keys such as `destination_path` or `schedule_cron`. 4. Save the context, then stage or redeploy the job so workers pick up the new values.

Worker-specific overrides

1. Navigate to **Workers**, then open the worker you want to customise. 2. Use the **Worker context** card to set host-specific values (for example `region` or `local_path`). 3. Save the changes; the worker receives the updated map immediately.

Manage context via the API

Automation and scripting workflows can call the REST endpoints directly. Authenticate with an admin JWT or session cookie.

```
# Update a global setting
curl -s -X POST "https://lyftdata.example.com/api/contexts/update-global" \ -H "Authorization:
Bearer ${ADMIN_JWT}" \ -H "Content-Type: application/json" \ -d '{"s3_export_bucket": "analytics-
prod"}'
```

```
curl -s -X POST "https://lyftdata.example.com/api/contexts/update-job-map/daily-sync" \ -H
"Authorization: Bearer ${ADMIN_JWT}" \ -H "Content-Type: application/json" \ -d
'{"schedule_interval": "15m"}'
```

```
curl -s -X POST "https://lyftdata.example.com/api/contexts/update-worker-map/worker-01" \ -H
"Authorization: Bearer ${ADMIN_JWT}" \ -H "Content-Type: application/json" \ -d
'{"local_cache_dir": "/var/cache/lyftdata"}'
```

Use `update-worker-nomerge` if you need to replace a worker context wholesale instead of merging keys.

Inline job defaults

Jobs can ship default values alongside the definition. Add a `context` block to the YAML and reference those keys throughout the pipeline:

```
name: ping
context: addr: https://status.example.com/api interval: 5m input: http-poll: url: "{{addr}}"
trigger: interval: "{{interval}}" json: true output: print: {}
```

Operators can later override `addr` or `interval` globally, per job, or per worker without editing the job file.

Built-in context variables

The control plane injects helper keys that are always available when a job is rendered:

- `{{job}}` – The job name being deployed.
- `{{worker}}` – The identifier of the worker executing the job.

Best practices

- Keep secrets in the secrets/variables store and reference them via `${secret|scope/name}` or `${dyn|NAME}`; avoid embedding credentials directly in context defaults.
- Prefer descriptive key names (`analytics_export_bucket`) over abbreviations.
- Update context through staging or change management so overrides are reviewed like code.
- Document critical keys in your team runbooks so on-call engineers know how to adjust them safely.

Debugging context

- Review the **Context** workspace for a live view of global variables.
- Download the merged context for a job by staging it and viewing the **YAML** tab, or fetch it through the API:

```
  curl -s "https://lyftdata.example.com/api/contexts/global" \
-H "Authorization: Bearer ${ADMIN_JWT}" | jq curl -s
"https://lyftdata.example.com/api/contexts/worker/worker-01" \ -H "Authorization: Bearer
${ADMIN_JWT}" | jq
```

- If a value is not taking effect, confirm there is not a higher-precedence override (job or worker) masking your change.

Example: multi-level override

1. Define `context.trigger_interval: 10m` inside the job definition to supply a safe default. 2. Set a global override `trigger_interval=5m` for production in the Context workspace. 3. Pin `trigger_interval=30m` on a single worker that talks to a rate-limited API.

The deployed job resolves to `30m` on that worker, `5m` on every other worker, and falls back to `10m` in environments where no overrides exist.

Executing Commands

Run shell commands in LyftData jobs using the exec input

Source: reference/executing-commands.md

Executing Commands

LyftData exposes the `exec` **input** so you can reuse shell scripts, operating system tools, and CLIs inside a job. The worker launches the command through the host shell and turns its output into events for downstream actions.

When to use exec

- Reuse system utilities or legacy scripts without building a custom connector
- Gather diagnostics (uptime, disk usage) alongside streaming telemetry
- Fan results into tools that expect newline-delimited JSON or plain text
- Prototype quick integrations before you invest in a dedicated input or output

Exec input

Exec inputs run commands on a schedule or once at startup and treat the output as incoming events. LyftData runs the command with `/bin/sh` on Unix platforms or PowerShell/cmd on Windows, so multi-line command blocks and shell features are available.

Key capabilities:

- Preserve multi-line command strings with `no-strip-linefeeds`
- Control output framing with `json` (treat each line as JSON) and `ignore-line-breaks` (emit the entire run as one event)
- Schedule recurring runs via `trigger.interval.duration`, and bound execution with `timeout.value`
- Provide environment variables via `env.file` (path) or `env.values` (multiline `KEY=value` pairs)

```
input:
exec: command: | ./bin/collect-metrics \ --tenant retail-eu trigger: interval: duration: 2m ignore-
line-breaks: true result: status-field: exit_status stdout-field: metrics_stdout stderr-field:
metrics_stderr env: values: | API_TOKEN=${secret|api/token} TENANT=retail-eu timeout: value: 30s
retry: timeout: 30s retries: 5
```

When `json: true`, each line is parsed as JSON instead of being wrapped in the `_raw` field. Enable `ignore-line-breaks` to combine multi-line output (for example, certificate dumps) into a single event for downstream parsing.

Operational considerations

- Commands run inside the worker runtime, so ensure the binary or script is present on every worker host
- Treat exec jobs like any other external dependency: capture exit codes and stderr into fields and alert on changes
- Prefer idempotent commands and explicit timeouts; use `retry` to bound repeated failures
- Store secrets in the secrets store and reference them with `${secret|scope/name}` (for example `${secret|api/token}`), instead of embedding credentials directly in the command string

File Handling

Choose between log files and block stores, and understand naming, partitioning, and durability

Source: reference/file-handling.md

LyftData can write data to two broad classes of destinations:

- **Log files** via the `file` output (newline-delimited events written to the local filesystem)
- **Block stores / object stores** via outputs like `file-store`, `s3`, `gcs`, `azure-blob`, and `web-dav-store` (objects written per batch; no append)

These can look similar in the editor, but their on-disk (and on-cloud) semantics differ in important ways.

Pick the right output

You need	Prefer	Why
Append events to a local file you can tail	`file`	Writes one line per event and keeps a single growing file.
Durable, partitioned exports to a "bucket"	Block store outputs (`s3`, `gcs`, `azure-blob`, `file-store`, `web-dav-store`)	Writes whole objects; supports batching + preprocessors; avoids "append to object" pitfalls.
"Object store semantics" on disk	`file-store`	Mirrors cloud object stores while writing to a local directory.

Where data is written

All outputs run on the **worker executing the job**:

- If you run on the **built-in worker**, paths are on the server host/container.
- If you run on an **external worker**, paths are on that worker host/container.

Use absolute paths and ensure the target directory is persistent (volume-mounted in containers) and writable.

Log Files (`file`): append vs overwrite (and flushing)

The Log Files output writes **one line per event** (NDJSON-style when writing full JSON events).

- Default behavior is **append**: `file-per-event: false` appends to an existing file (and creates it if missing).
- `file-per-event: true` writes each event by opening the path for a fresh write (so a constant `path` will be overwritten repeatedly). Use this only with a unique `path` (for example, by including `${}` expansions in the filename).
- `compress-after: true` gzips the previous file **when the expanded path changes** (and removes the uncompressed file).
- `truncate: true` truncates a file the first time it is seen during a run (useful when re-running a job into the same path).

Flushing: as of LyftData `2.0.2`, the `file` output flushes each event write (the `flush-at-end` field exists in the schema but is not yet honored). For high-throughput exports, prefer a block store output with batching instead of relying on file buffering.

Block store outputs: objects, GUIDs, and “no append”

Block store outputs write **objects**, not log files:

- There is **no append** operation for an existing object. Each put writes a complete object body.
- By default, the outputs append a **GUID** to avoid collisions. The default `guid-prefix` is `/`, so `object-name.name: processed/summary.json` becomes keys like `processed/summary.json/<uuid>`.
- If you want filename-style keys, keep the GUID enabled but set `guid-prefix` / `guid-suffix`. For example:

- `object-name: { name: processed/${partition||unknown}/summary }`
- `guid-prefix: -`
- `guid-suffix: .json.gz`

...yields keys like `processed/2026-03-19/summary-<uuid>.json.gz`.

- Set `disable-*-name-guid: true` only when you **intend deterministic overwrites** or your name is already unique.
- Preprocessors like `gzip` compress bytes but do not rename objects; include `.gz` in the key yourself.

For large batched uploads, prefer streaming multipart writes when supported:

```
runtime-options:
prefer-streaming-outputs: true
```

This reduces peak memory by streaming batched uploads instead of buffering the entire combined batch in memory.

Naming and partitioning with `${}` expansions

Many output fields accept `${}` expansions evaluated per event (including file paths and object names). Use `||` to provide defaults:

- `${partition||unknown}`
- `${host||unassigned}`

It is usually clearer to create a **single partition field** in your actions (and sanitize it), then reference it from outputs.

When batching is enabled on a block store output, expansions are resolved once per batch (using the last event in the batch). Ensure each batch contains only one logical partition (or use `${stat|batch_number}` instead of per-event fields).

Example: build a partition prefix and use it in an object-store key:

```
actions:

  • slugify:


input-field: host output-field: host_slug


  • add:
```

```
output-fields: partition: host/${host_slug}


output: s3: bucket-name: analytics-prod-archive object-name: name:
exports/${partition||unknown}/events guid-prefix: "-" guid-suffix: ".ndjson"
```

If you need a simple per-batch uniqueness knob, outputs can also expand `${stat|batch_number}` when batching is enabled (for example: `events-${stat|batch_number}.ndjson`).

Glossary

Glossary

Source: reference/glossary.md

General terms

`Server`

The server is the control plane for a LyftData deployment. It hosts the UI and APIs, manages workers and jobs, and stores metrics, logs, and traces.

`Worker`

A worker is a process that connects to the server and runs jobs.

`Worker group`

A worker group is an operator-defined set of workers used for placement and scaling decisions in Deployments.

`Job`

A job is a pipeline with one input, zero or more actions, and one output. The job definition captures that configuration.

`Job definition`

A job definition is the saved configuration for a job. Authors build and edit definitions in the visual editor before staging them for deployment.

`Event data`

Event data is the payload that flows through a job from input to output.

`Input`

The input is the first stage of a job and determines where event data originates (for example, S3, HTTP, or files).

`Output`

The output is the final stage of a job and controls where processed event data is delivered.

`Action`

An action transforms, enriches, or routes event data between the input and output stages of a job.

`Visual editor`

The visual editor is the UI workspace for creating, inspecting, and modifying job definitions.

`Staging`

Staging locks a job definition so it can be deployed to workers. A staged version is immutable; further edits create a new staged revision.

`Deployment`

In LyftData, “deployment” can refer to two related concepts:

- **Job deployment**: applies a staged job definition to a worker so it can execute the job (see Deploying Jobs).
- **Workflow deployment**: a Deployment Manager record that plans/applies a workflow across workers (see Deployments Overview).

`Job deployment`

A job deployment applies a staged job definition to a worker.

`Workload`

A workload is the running execution state of a deployed job on a worker (what is actually running after a deploy/apply).

`Deployment Manager`

The Deployment Manager is the UI/API that plans, applies, and reconciles workflows (and some blueprints) into running jobs on workers.

`Deployment record`

A deployment record captures the desired state for a workflow/blueprint deployment (what should run, where it should run, and with what parameters).

`Workflow`

A workflow is a versioned graph of steps and edges that composes multiple jobs/modules into a system.

`Blueprint`

A blueprint is a reusable building block used inside workflows (connectors, adapters, and patterns). Some blueprints can also be planned/deployed directly.

`Template`

A template is a reusable job starter (snippet, sometimes parameterized) that you can apply when creating a new job draft.

`Library job`

A library job is a managed job snippet stored in the Catalog. It is a known-good starting point when you want “one job that does X”.

`Auto-enrollment`

Auto-enrollment authenticates workers with a pre-shared key instead of individual API keys.

`Worker API key`

A worker API key authenticates a specific worker when it connects to the server.

`Metrics`

Metrics are usage counters and rates generated by jobs and stored by the server according to retention settings.

`Logs`

Logs record operational messages from the server, workers, and jobs for troubleshooting and auditing.

`Traces`

Traces capture step-by-step execution data for events so you can inspect them in Run & Trace.

`Message system`

The message system is the internal bus workers and jobs use to exchange events, metrics, and user-defined messages.

`Context`

Context is configuration data defined at the server, worker, or job level. Jobs access context values with the `{{CONTEXT_KEY}}` syntax.

`Variable expansion`

Variable expansion resolves runtime values (event fields, context keys, job metadata) inside job definitions.

Job Language

Job Language

Source: reference/language.md

> Essential content

- What are jobs, technically?
- Difference between visual editor and YAML editor

> Note

Licensing

Community limits, evaluation licenses, and activation workflow

Source: reference/licensing.mdx

Community Edition runs without a license; it is ideal for evaluation and small deployments.

> Community Edition limits

- One built-in worker (external workers require a license)
- Up to 15 jobs across the workspace
- Daily processing volume cap of approximately 250 GB/day
- Licensed-only connectors and features remain unavailable in Community Edition

> Note

Use this guide when you are ready to upgrade so you can add external workers, raise limits, and unlock licensed-only features and support.

Request a free evaluation license via email using the license request form.

> Evaluation license limits

Evaluation licenses are time-limited and include a daily processing volume cap and a worker limit. The exact limits depend on your evaluation program; after activation, check **Settings** → **License** for the values shown in your License Summary.

> Note

For production or higher-throughput deployments, contact us to review available licensing tiers and pricing.

Configuring a server license

To configure a license on a LyftData server, follow these steps:

1. Sign in to the UI as `admin` using the one-time password from the server log. 2. Open the main navigation menu and choose **Settings** → **License**. 3. Click **Change license**, paste your key, and select **Activate**.

Alternative: provide license via environment variable

You can provide a license non-interactively on server startup using the LYFTDATA_LICENSE environment variable. This is useful for automation:

```
# example: systemd EnvironmentFile or shell export before starting
export LYFTDATA_LICENSE="<paste-your-license-jwt>"
```

- When set, the server uses this value as a bootstrap license at startup.
- If no license exists yet, it is applied automatically.
- If an existing stored license is invalid, the bootstrap license is used to replace it.
- If a valid stored license already exists, release builds keep the existing license.
- For first-run automation, you can also accept the EULA via `LYFTDATA_LICENSE_EULA_ACCEPT=yes`.

See also: Configuration for details on these environment variables.

Reducing Data Volume

Use filtering, streaming deltas, and compact encodings to lower egress costs

Source: reference/reducing-data-volume.md

Reducing Data Volume

Edge workers can discard noise and compress payloads before data leaves a site. The actions below mirror the Hotrod guidance but use LyftData's DSL and runtime semantics so remote jobs can keep bandwidth and licensing costs in check.

Drop events early

Use the `filter` action to keep only events that match fixed patterns or Lua conditions. Multiple filters can be chained—one to match fields, another to gate by a numeric threshold.

```
actions:
  - filter:

field-pattern-pairs:

  - severity: 'high'
  - source: '^GAUTENG-'
  - filter:

condition: speed > 1
```

If you only want specific keys to survive, switch the filter into schema mode so that any other fields are dropped in-place:

```
actions:
  - filter:

schema:

  - source
  - destination
  - sent_kilobytes_per_sec
```

Combine this with the `remove` and `rename` actions to strip temporary fields or shorten key names before handing events to the next hop.

Forward only changes

The `stream` action keeps a running value and emits deltas. Set `only-changes` so that identical samples disappear, and LyftData will also include an elapsed time field for context.

```
actions:

  • stream:


delta: true watch: throughput only-changes: true output-field: delta elapsed-field: elapsed_ms


  • filter:


condition: delta != 0
```

This pattern is ideal for forwarding counters or gauges that rarely change but must be monitored continuously.

Trim payload fields

After filtering, `remove` drops helper keys (optionally via regular expressions) and `rename` shortens long field names so JSON payloads shrink on the wire:

```
actions:

  • remove:


fields: ["_raw", "debug"]


  • rename:


key-value-pairs:


  • source=s
  • destination=d
  • sent_kilobytes_per_sec=sent
```

Compact payloads for transport

If your downstream systems accept batches, the biggest wins usually come from batching and compressing at the output boundary:

- Use output `batch.wrap-as-json` to send JSON arrays instead of many small JSON documents.
- Use object-store output `preprocessors: [gzip]` to compress the payload before upload.

Example: batch + gzip to S3

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



  - gzip



batch: mode: fixed fixed-size: 500 timeout: 1s wrap-as-json: true retry: timeout: 30s retries: 5
```

With these primitives, edge workers can minimize bandwidth and storage costs while still delivering complete records to central collectors.

Scripting

Use Lua scripting for calculated fields, conditional logic, and runtime-aware helpers

Source: reference/scripting.md

LyftData ships a Lua 5.3 runtime for the `script` action. Use it when you need to derive fields, normalize payloads, or branch on complex logic without shelling out to external tooling. Scripts run on each event the action receives, working in-place on the JSON document.

Core syntax

```
- script:
let:


  - total: amount * tax_rate

  - normalized_status: string.upper(status)


set:


  - site: '{{worker}}'


merge: overwrite condition: amount ~= nil
```

- **`let`** lists `field: expression` pairs whose values are evaluated for every event.
- **`set`** assigns literal values. Context expansions such as `{{job}}` and `{{now}}` are available.
- **`merge`** controls how existing fields are handled:
- `unless-exists` (default) keeps the original value if the field already exists.
- `overwrite` always replaces the value.
- `error` leaves the event untouched and records an attachment when a scripted field already exists.
- **`condition`** guards the entire action. When it evaluates to `false`, none of the `let` or `set` expressions run.

Field names must be valid Lua identifiers (start with a letter, contain letters, numbers, or `_`). Nested fields use dot notation (`http.status`), and arrays are 1-indexed (`hosts[1]`). The pseudo field `_E` exposes the entire event for cloning or inspection.

Runtime helpers

The runtime preloads helpers before your script executes. Selected categories:

- **Core helpers**: `count()` (per-action counter), `round(x)`, `cond(condition, a, b)`, `condn(...)` (multi-branch), `array(...)`, `map(...)`, `len(value)`, `json(value)` (pass-through), and a `NULL` sentinel usable with `is_null(value)`.
- **Randomness**: `rand(n)` returns a random integer between 1 and `n`; `pick_random(...)` selects one argument at random. Because randomness is seeded per action execution, downstream steps behave deterministically within a run.

- **Aggregation**: `sum(accumulator, value, [keep_running])` maintains running totals keyed by the string in `accumulator`. Drop in `keep_running` to reset when a condition flips.
- **Time**: `sec_s()` and `sec_ms()` return the current epoch time in seconds or milliseconds.
- **Network & matching**: `cidr(ip, "10.0.0.0/24")` checks membership in an IPv4 CIDR range.
- **Hashing**: `md5(text)`, `sha1(text)`, `sha256(text)`, `sha512(text)` return lowercase hex digests.
- **Identifiers**: `uuid()` emits a version 4 UUID.
- **Base64**: `encode_base64(text)` and `decode_base64(text)` encode or decode UTF-8 strings.
- **Encryption** (requires the binary to ship with the corresponding features):
- `encrypt(plaintext, key)` and `decrypt(blob, key)` provide backwards-compatible AES-CBC wrappers that now route through the AEAD implementation.
- `encrypt_s(plaintext, key, scheme)` / `decrypt_s(blob, key)` use AEAD (default `chacha20poly1305`, pass `"aes256gcm"` for AES).
- `decrypt_auto(blob, key)` accepts either legacy or AEAD payloads.
- `encrypt_for(plaintext, recipient_pub_b64)` / `decrypt_with(blob, recipient_priv_b64)` expose HPKE X25519 + ChaCha20-Poly1305 when built with the `hpke` feature.
- `encrypt_age(text, recipients_csv)` / `decrypt_age(blob, identities_csv)` integrate with age recipients when the `age` feature is enabled.
- **Job metrics**: scripts can query runtime counters— `error_count()`, `warning_count()`, `input_event_count()`, `output_event_count()`, `run_count()`, `batch_number()`, and related byte counters —for telemetry-aware logic.
- **State store**: `store_set(key, condition, value)` saves a string in a per-job in-memory cache when `condition` is `true`; `store_get(key, default)` retrieves it. Use this for lightweight state between events.

Lua's base libraries (`math`, `string`, `table`, `base`) are available. Sandbox safety removes `require`, `dofile`, `load`, and `collectgarbage`. Referencing a missing field raises an error by default; support can flip the runtime flag to downgrade these to null assignments during troubleshooting.

> **Note** > The older Hotrod pipelines exposed helpers such as `emit()` and `ip2asn()`. LyftData no longer ships those bindings—scripts work on the current event only. Use the `expand-events` action when you need to fan out arrays into multiple events.

Examples

Derived fields and normalization

```
name: normalize-orders
input: text: | {"amount": 125.50, "currency": "usd", "customer": "ALICE"} actions:


  • script:


let:


  • total_cents: round(amount * 100)

  • currency: string.upper(currency)

  • customer: string.lower(customer)

  • observed_at: sec_ms()
```

```
merge: overwrite output: write: console
```

Rolling sums with conditional resets

```
- script:
let:

  - batch_total: sum("orders", revenue, status == 'ok')
  - batch_seq: count()

condition: revenue ~= nil
```

When `status` stops equalling `ok`, the accumulator resets; the next matching event starts a fresh running sum.

State between events

```
- script:
let:

  - last_status: store_get('status', 'unknown')
  - status_changed: last_status ~= status
  - _ignored: store_set('status', status_changed, status)
```

The helper returns the previous status while updating the cache only when it actually changed.

Guarded execution

```
- script:
condition: condn(env == 'prod', true, env == 'staging', run_count() % 10 == 0, false) let:

  - census: map('count', input_event_count(), 'warnings', warning_count())
```

Production runs on every event; staging only every tenth batch; everything else skips the action entirely.

Extending the environment

`init.lua`

If your job package includes an `init.lua` file, LyftData loads it before any script runs. Use it to declare shared functions:

```
-- init.lua
function every(n) return count() % n == 0 end


function normalize_country(code) local normalized = string.upper(code or '') if normalized == 'UK'
then return 'GB' end return normalized end
```

```
- script:
let:



  - counter: count()
  - should_emit: every(5)
  - country: normalize_country(country)



condition: should_emit
```

Bundle `init.lua` under the job's `files:` section so workers download it alongside the spec. Scripts run inside the same interpreter, so keep helper names unique to avoid collisions.

Loading additional Lua modules

Set the `load` attribute to import another Lua file bundled with the job:

```
- script:
load: lib/string_utils.lua let:



  - segments: split_path(url)
  - tenant: segments[2]
```

The referenced file is read from the job package before the action executes. This gives you a place to stage larger helper libraries while keeping `init.lua` for global bootstrap code.

```
-- lib/string_utils.lua
function split_path(url) local segments = {} for segment in string.gmatch(url or '', "[^/]+") do
table.insert(segments, segment) end return segments end
```

Load helpers like this alongside the job so every worker sees the same implementation.

Side-effect scripts with `run`

The `run` option executes a Lua expression for each event without mutating the payload. Use it for callbacks defined in `init.lua` or modules loaded via `load`:

```
- script:
run: > if error_count() > 0 and run_job_errors() % 50 == 0 then store_set('error_alert_marker',
true, tostring(run_job_errors())) return true end return false
```

`run` scripts can still access and modify globals, but because they bypass `let` / `set`, events flow through unchanged.

Troubleshooting tips

- Missing fields or helpers raise runtime errors that appear in the job attachments. When debugging, operations can flip the “suppress script errors” toggle to coerce failures to `null` assignments.
- Remember that Lua arrays start at 1. When you need zero-based math, subtract 1 explicitly.
- Use the `filter` or `assert` actions when you need to drop or block events—scripts only modify the document, they do not control flow.
- Keep cryptographic keys outside the spec; fetch them from the environment or licensing system and inject through context expansions.

With these helpers and patterns, the `script` action remains the workhorse for pipeline-specific business logic in LyftData without sacrificing determinism or sandbox safety.

Troubleshooting

Common issues and quick fixes for Server, Workers, and the CLI

Source: reference/troubleshooting.md

Use this page as a quick playbook when something goes sideways. Start with the symptom, work through the checks, and capture diagnostics before escalating.

How to use this page

1. Find the symptom that best matches what you see. 2. Run the quick checks in order; most issues are solved within the first two steps. 3. Collect the diagnostics listed in the **Capture before escalating** section and attach them to support requests.

Connectivity issues

Cannot access the web UI

- Symptom: Browser cannot reach the UI from another machine or the connection is refused.
- Quick checks:
- Start the server with a public bind address: `lyftdata run server --bind-address 0.0.0.0:3000`.
- Or set `LYFTDATA_BIND_ADDRESS=0.0.0.0:3000` in the environment.
- Confirm firewalls or security groups allow inbound TCP 3000.
- Default bind address is `127.0.0.1:3000` (local only).

"Login failed" or CLI cannot reach the server

- Symptom: `lyftdata login admin` fails or CLI commands time out.
- Quick checks:
- Ensure the server is running and reachable at `LYFTDATA_URL` (default `https://localhost:3000/`).
- If using a non-default port or address, export `LYFTDATA_URL` with the correct scheme, for example `https://server:3000/` (or `http://server:3000/` only when the server runs with `--disable-tls`).
- If the server is using the default self-signed certificate, re-run the CLI command with `--tls-insecure` (or set `LYFTDATA_TLS_INSECURE=true`) until you install a trusted certificate.
- If the password is lost and no admin session remains, follow Reset an admin password.

Worker fails to register or stays offline

- Symptom: Worker shows offline in the dashboard and does not receive jobs.
- Quick checks:
- Ensure `LYFTDATA_URL` points to the server API (include scheme and port).
- Verify API key and worker ID (`LYFTDATA_WORKER_API_KEY`, `LYFTDATA_WORKER_ID`).
- Confirm network reachability from the worker host to the server on TCP 3000.
- Capture before escalating:
- Worker logs (`journalctl -u lyftdata-worker` or console output).
- Recent server logs showing registration attempts.

Licensing and onboarding

EULA prompt blocks automation

- Set `LYFTDATA_LICENSE_EULA_ACCEPT=yes` in the environment for the first run to bypass the interactive prompt.
- Apply the same variable when automating worker or CLI runs that would otherwise prompt.

Evaluation license expired or missing

- Check the License page in the UI for status and expiry.
- Reapply the license through the UI or restart the server with an updated `LYFTDATA_LICENSE` environment variable if you manage licenses non-interactively.

Installation and permissions

Server will not start because of staging directory permissions

- Verify the service account owns the staging directory and it is writable.
- Avoid running the server as root; create a dedicated user and fix ownership (`chown -R lyftdata:lyftdata /var/lib/lyftdata`).
- If running on Windows, run the service as an account with Modify permissions on the staging directory.

Server startup fails in Docker with keyring/DBus errors

- Symptom: startup aborts with errors like `failed to access variables master key` and `Unable to autolaunch a dbus-daemon without a $DISPLAY for X11` .
- Cause: headless containers usually do not have a desktop keyring/DBus session, so keyring-backed master key loading fails.
- Quick checks:
- Configure env-backed master keys for server and workers.
- Use `--variables-master-key-source env` (or `LYFTDATA_VARIABLES_MASTER_KEY_SOURCE=env`) for the server.
- Do not use legacy settings like `LYFTDATA_KEYRING_BACKEND` , `LYFTDATA_MASTER_KEY` , or `--master-key-backend` ; current binaries use scoped master-key variables.
- External workers require a licensed deployment; Community Edition supports only the built-in worker.
- For a full container setup guide, see Docker and Docker Compose.

```
services:
lyft-server: build: . restart: unless-stopped command:


  - run
  - server
  - --disable-tls
  - --bind-address
  - 0.0.0.0:3000
  - --variables-master-key-source
  - env


ports:


  - "3000:3000"
```

```
environment: LYFTDATA_LICENSE_EULA_ACCEPT: "yes" # Recommended for published container ports:
create the first admin explicitly # rather than depending on the local setup-link bootstrap flow.
LYFTDATA_ADMIN_INIT_PASSWORD: "ChangeMeVerySoon1" # Optional but recommended: bootstrap the license
non-interactively (required for external workers on first run) LYFTDATA_LICENSE: "<paste-your-
license-jwt>" LYFTDATA_STAGING_DIR: "/data" LYFTDATA_AUTO_ENROLLMENT_KEY:
"ChangeThisEnrollmentKey!" LYFTDATA_VARIABLES_MASTER_KEY_SOURCE: "env"
LYFTDATA_VARIABLES_MASTER_KEY: "<64-hex-chars>" LYFTDATA_CREDENTIAL_MANAGER_MASTER_KEY_SOURCE:
"env" LYFTDATA_CREDENTIAL_MANAGER_MASTER_KEY: "<64-hex-chars>" # Required in headless containers
because the built-in worker runs inside the server LYFTDATA_SETTINGS_MASTER_KEY_SOURCE: "env"
LYFTDATA_SETTINGS_MASTER_KEY: "<64-hex-chars>" volumes:


  - ./lyft_data/server:/data


worker-alpha: build: . restart: unless-stopped command:


  - run
  - worker
  - --url
  - http://lyft-server:3000
  - --worker-name
  - worker-alpha
  - --worker-jobs-dir
  - /data


depends_on:


  - lyft-server


environment: LYFTDATA_LICENSE_EULA_ACCEPT: "yes" LYFTDATA_AUTO_ENROLLMENT_KEY:
"ChangeThisEnrollmentKey!" LYFTDATA_SETTINGS_MASTER_KEY_SOURCE: "env" LYFTDATA_SETTINGS_MASTER_KEY:
"<64-hex-chars>" volumes:


  - ./lyft_data/worker-alpha:/data


worker-beta: build: . restart: unless-stopped command:


  - run
  - worker
  - --url
```

```
    - http://lyft-server:3000
    - --worker-name
    - worker-beta
    - --worker-jobs-dir
    - /data

depends_on:

    - lyft-server

environment: LYFTDATA_LICENSE_EULA_ACCEPT: "yes" LYFTDATA_AUTO_ENROLLMENT_KEY:
"ChangeThisEnrollmentKey!" LYFTDATA_SETTINGS_MASTER_KEY_SOURCE: "env" LYFTDATA_SETTINGS_MASTER_KEY:
"<64-hex-chars>" volumes:

    - ./lyft_data/worker-beta:/data
```

- Note: if your server is running with TLS enabled, switch worker URL values to `https://lyft-server:3000` and set `LYFTDATA_TLS_INSECURE=true` for local/self-signed evaluation.

Worker exits immediately on startup

- Confirm the binary matches the host architecture.
- Check that required environment variables are set (`LYFTDATA_URL` , `LYFTDATA_WORKER_API_KEY`).
- Review stdout/stderr for errors; if possible, run the worker in the foreground once to capture the full startup output (add `-v` for more logs).

Worker local state is corrupted or auth lease is stuck

- Symptom: startup errors mention malformed sqlite databases, auth lease validation loops, or the worker never recovers after credential-related failures.
- Quick checks:
- Stop the worker service first (`systemctl stop lyftdata-worker` , `sc stop LyftDataWorker` , or equivalent).
- Confirm the worker state directory (`LYFTDATA_JOBS_DIR` or default worker cache path).
- Start once with one of the reset modes:
- `lyftdata-worker --reset-state auth`
- `lyftdata-worker --reset-state repair`
- `lyftdata-worker --reset-state full`
- Restart normally without `--reset-state` after the one-time repair/reset.
- Reset mode guidance:
- `auth` : clears cached worker auth lease material so the worker re-fetches fresh auth state from the server.
- `repair` : runs sqlite integrity checks and rebuilds unhealthy worker sqlite files from a logical dump.
- `full` : wipes local worker runtime state and recreates a minimal baseline, retaining worker identity/API key where possible.
- Notes:
- `repair` requires `sqlite3` to be available on the worker host.
- `full` may still require `--worker-api-key` or auto-enrollment if no reusable local credentials exist.

- You can use `lyftdata run worker --reset-state <mode>` instead of `lyftdata-worker` if you run the combined binary.

Jobs and pipeline execution

No data flowing through a job

- Confirm the job is staged and deployed to an online worker.
- Review **Run Output** and **Logs** for validation errors or connector failures.
- Verify connector credentials and destination permissions.
- Use the Issues pane to resolve schema or validation problems before redeploying.

UI actions do not take effect after staging or deploying

- Check that the job version shows as `Staged` and the intended worker is connected.
- For external workers, validate worker enrollment, API key, and `LYFTDATA_URL`.
- Ensure the browser session is using the latest admin password; re-login if prompted.

Validation errors referencing missing fields

- Ensure upstream jobs emit the fields referenced by Convert/Filter actions.
- Use the **Preview** tab on each action to inspect incoming sample events.
- When splitting pipelines, document channel schemas so downstream jobs expect the correct shape.

Performance and scaling

Backpressure or slow throughput

- Monitor worker CPU and memory utilisation; scale horizontally by adding workers.
- Use worker channels to fan out workloads and avoid single-job hotspots.
- Apply rate limits on inputs where bursts cause congestion.
- Inspect the worker backlog charts (UI) for long-running actions.

Disk usage high or logs purged unexpectedly

- The server cleans up data when disk thresholds are exceeded.
- Adjust retention: `--db-retention-days` or `LYFTDATA_LOG_RETENTION_DAYS` (default 30 days).
- Adjust disk usage threshold: `--disk-usage-max-percentage` or `LYFTDATA_DB_DISK_USE_MAX_PERCENT` (default 80%).
- Provision more disk for the staging directory volume.

Diagnostics and logs

- Server staging directory: defaults to a per-user data path, configurable via `--staging-dir` or `LYFTDATA_STAGING_DIR`.
- Worker job/data directory: `LYFTDATA_JOBS_DIR` or the worker cache directory if unset.
- Collect server and worker logs, job run result JSON, and Issues list from the UI when escalating.
- Review the Logs & Issues guide if warnings are not appearing where you expect them.
- Record exact versions, host platform details, and recent configuration changes.

Additional resources

- Configuration reference: reference/configuration

- Getting started guide: Getting started
- Worker installation (Linux): Install on Linux
- Multi-job orchestration concepts: Build overview

Web Services And Client

Handle inbound webhooks, scheduled API calls, and outbound HTTP delivery with LyftData

Source: reference/web-services-and-client.md

Web Services and Client

LyftData ships HTTP-focused connectors so a pipeline can accept inbound webhooks, poll remote APIs, and push results to downstream services—all without custom glue. This page highlights the core pieces and when to use them.

For operator recipes (auth patterns, pagination, RSS/Atom, webhooks), see HTTP endpoints.

Receive webhooks with `http-server`

Use the `http-server` input when external systems need to post data into LyftData. The runtime spins up a listener on the address (and optional path) you configure, captures the HTTP method, URL, decoded query parameters, headers, and body into a JSON event, and can enrich the event with the caller IP.

Key capabilities:

- Serve plain HTTP or enable TLS by supplying `tls.cert` and `tls.key`.
- Shape responses with `custom-response` and `content-type`, or narrow the capture to a single query parameter via `query`.
- Switch to body-only mode (`only-body`) when you only care about the payload.

```
input:
http-server: address: 0.0.0.0:8443 path: /webhook/ingest tls: cert: /etc/lyftdata/certs/server.crt
key: /etc/lyftdata/certs/server.key source-ip-field: client_ip custom-response:
'{"status":"accepted"}' content-type: application/json
```

Every request produces an event containing the method, URL, decoded query map, headers, body, and (if configured) `client_ip` field so downstream actions can fan out or persist the exact webhook call.

Poll APIs with `http-poll`

`http-poll` is the scheduled HTTP client. It supports interval or cron triggers, arbitrary methods (GET/POST/PUT/DELETE and more), headers, query parameters, basic auth, request bodies, and form URL-encoded payloads. The `response` section lets you opt in to status codes, response headers, or rename the body field, while `json`, `ignore-line-breaks`, `document-mode`, and `events-field` control how the payload is emitted.

```
input:
http-poll: url: https://status.example.com/api/incidents trigger: interval: duration: 60s method:
get headers: Authorization: Bearer ${secret|status/token} Accept: application/json response:
status-field: http_status headers-field: response_headers response-field: incident_body json: true
ignore-line-breaks: true
```

When the endpoint answers, LyftData records the status and headers (if requested) alongside the parsed JSON body, so later actions can branch on `http_status` or pick fields out of `incident_body` without losing context.

Pagination strategies

`http-poll` can paginate across multiple HTTP requests within a single run:

- **Cursor** (`pagination.strategy: cursor`): extracts a cursor and/or next-page URL from a JSON response body and persists the cursor in Worker KV so the next run can resume.
- **Link header** (`pagination.strategy: link-header`): follows RFC8288 `Link: <...>; rel="next"` headers within a run (stateless; no checkpoint).
- **Query param** (`pagination.strategy: query-param`): increments a numeric query parameter (for example `?page=2` , `?paged=3`) and persists the “next value to request” (defaults to runtime artifacts; Worker KV is an explicit opt-in).

For RSS/Atom feeds that paginate via a `paged` query parameter (common on WordPress), pair `http-poll` with the `xml` and `expand` actions:

```
name: rss-wordpress-news-poll
description: "Poll an RSS feed via http-poll and paginate via a page query-param."

input: http-poll: trigger: interval: duration: 300s url: https://wordpress.org/news/feed/ method:
get headers: Accept: application/rss+xml ignore-line-breaks: true document-mode: true payload-mode:
raw

pagination: strategy: query-param param: paged start: 2 step: 1 max-pages-per-run: 10 # checkpoint
defaults to runtime-artifacts. # terminal-status-codes defaults to [404]. continue-if-body-regex: "
<item>"

actions:

  - xml:

input-field: _raw remove: true

  - expand:

mode: events: skip-list:

  - "^/rss/channel/item/\\d+/"

exclude-non-empty-arrays: false
```

Send events with `http-post` or `http-get`

Use the `http-post` output to deliver processed events to remote services. URLs support `${}` field expansions, and `body` controls how the request payload is built (entire event, a specific field, or a literal/template string). You can also add headers, retries, batching, and method overrides—including turning the output into a GET or DELETE call. LyftData uses the same engine for the `http-get` DSL shortcut.

```
output:
http-post: url: https://collector.example.com/events/acme-retail method: post headers: Content-
Type: application/json X-Request-ID: "{{context.request_id}}" body: field: field: payload retry:
timeout: 30s retries: 5 batch: mode: fixed fixed-size: 100 timeout: 250ms wrap-as-json: true
```

Each batch is posted with the selected HTTP verb; setting `method: delete` or choosing the `http-get` variant causes LyftData to reuse the same client with the alternate verb. The optional `insecure: true` flag allows self-signed endpoints during development (use with care).

Feature notes and compatibility

- **HTTP server output**: Hotrod exposed an HTTP server output that returned the last event; LyftData currently omits a runtime implementation. To expose live results over HTTP, connect the job to a downstream service (for example with `http-post`) or build a small façade that reads from the destination store.
- **Authentication helpers**: `http-poll` supports basic auth directly; for other schemes inject headers or tokens via context/secrets.
- **TLS**: Inbound TLS termination happens inside the `http-server` input. Outbound calls honor system root certificates and can skip verification only when `insecure` is set explicitly.

Worker Authentication

Worker bootstrap and runtime authorization

Source: reference/worker-authentication.md

Worker authentication has two layers:

1. **Bootstrap authentication** (how a worker first proves identity to the server). 2. **Runtime authorization** (how external workers are continuously authorized to execute jobs).

For bootstrap authentication, two strategies are available:

Strategy	Features
API key	Pre-create each worker and provide a per-worker API key
Auto-Enrollment	Bootstrap with a shared secret; server issues and the worker caches an API key

> note

The server can support both bootstrap strategies at once, but each worker should use one strategy during initial bootstrap.

> Note

API key authentication

API key bootstrap is best when you need explicit, per-worker control. It is especially useful for workers on less-trusted networks.

External workers require a licensed deployment. Community Edition supports only the built-in worker.

Adding a new Worker involves the following two steps, as well as specific configuration:

Server Configuration

1. Create a new Worker with a name. As an option, customize the Worker ID. 2. Create a new API key on the Server, or re-use an existing API key.

Worker Configuration

Under the Worker startup settings:

3. Configure the Worker ID (`LYFTDATA_WORKER_ID`) and API key (`LYFTDATA_WORKER_API_KEY`) as per the Server configuration above.

> note

The Worker name is optional when using API key authentication.

> Note

Auto-enrollment authentication

Auto-enrollment uses a server-side shared secret to bootstrap new workers quickly. The server then issues an API key, and the worker caches that credential for subsequent restarts.

Unlike the API key strategy, you do not need to pre-create each worker or issue per-worker keys manually. Because this uses a shared secret, use it only on trusted networks.

Auto-enrollment is disabled by default.

Server (licensed deployments)

1. Sign in as an admin and enable auto-enrollment in **Settings** → **Security**. 2. Set an enrollment secret (random 32+ characters) and save. The secret is write-only.

Worker

3. Configure a worker with a name (`LYFTDATA_WORKER_NAME`) and the shared secret (`LYFTDATA_AUTO_ENROLLMENT_KEY`). 4. In most cases leave pre-issued worker identity settings unset (`LYFTDATA_WORKER_ID` , `LYFTDATA_WORKER_API_KEY`). If you need deterministic IDs, you can provide `LYFTDATA_WORKER_ID` during enrollment.

The Server will automatically create Worker entries for any connecting Workers using the Server Auto-Enrollment secret.

After the first successful enrollment, remove `LYFTDATA_AUTO_ENROLLMENT_KEY` from the worker service configuration. The worker will continue using cached credentials.

> Security

Use a randomly generated value of at least 32 characters for the shared secret, for example: `c3s3R0s1T5QAQr7lz1KsT00pKh3adnma` .

> Note

> danger

If the Auto-Enrollment secret is known, anyone with network access to the Server can add new Workers, without having access to the Server itself.

> Note

See also: Worker Auto Enrollment for provisioning and rotation runbooks.

Runtime authorization (external workers)

After bootstrap, external workers are authorized at runtime using signed, time-bounded auth leases issued by the server and refreshed periodically.

- Built-in workers do not use auth leases.
- External workers that cannot obtain/refresh valid authorization stop running jobs.
- Community Edition does not authorize external workers.

Working With Data

Parse raw text, convert types, and reshape LyftData events before handing them to outputs

Source: reference/working-with-data.md

LyftData treats each event as a JSON object. Text-first inputs typically wrap payloads in `_raw` (or another configured field), and actions mutate the event in place. This guide shows common patterns for turning raw streams into structured fields and preparing payloads for downstream systems.

JSON as the baseline

- For text inputs, the raw payload usually lands in `_raw`.
- Prefer `json: true` on inputs that already emit JSON so fields are available without an extra parsing step.
- The `exec` input can place stdout/stderr/exit status into dedicated fields via the `result` block.

Example: capture a command's stdout in `uptime_raw`:

```
input:
exec: command: uptime result: stdout-field: uptime_raw
```

Extract unstructured text with `extract`

Use `extract` when a regular expression is the fastest path to structure. Captured groups become fields (either by name or by `output-fields` order).

```
actions:

  • extract:


input-field: uptime_raw pattern: 'load average: (\S+), (\S+), (\S+)' output-fields:


  • load1

  • load5

  • load15


remove: true drop: false
```

By default, `extract` emits warnings when the pattern does not match. Set `suppress-warnings: true` to silence them, or `drop: true` to stop non-matching events.

Convert values and units

Convert extracted strings into typed values:

```
actions:
  - convert:

conversions:

  - field: load1

conversion: num

  - field: load5

conversion: num

  - field: load15

conversion: num
```

Unit conversions work the same way:

```
actions:
  - convert:

units:

  - field: latency

from: ms to: s
```

If conversions may see empty values, pick a `null-behaviour` (`keep` or `default`) to avoid surprises.

Parse structured text

CSV

Parse CSV text into JSON fields with the `csv` action:

```
actions:
  - csv:
```

```
input-field: _raw header: true delim: ',' fields: port: num throughput: num remove: true
```

Key/value logs

Parse `k=v` style payloads with `key-value`:

```
actions:
  - key-value:

input-field: _raw delim: ' ' key-value-delim: '=' autoconvert: true remove: true
```

If the producer repeats keys, pick a strategy with `multiple: array` (collect all values), `first`, or `last`.

Split one record into many with `expand-events`

Use `expand-events` to fan out embedded arrays into multiple events:

```
actions:
  - expand-events:

input-field: results output-split-field: result skip-list:
  - /metadata

remove: true
```

Parse embedded JSON strings

If a field contains JSON as a string, use the `json` action:

```
actions:
  - json:

input-field: payload remove: true
```

Preparing non-JSON output

Outputs are JSON-first, but you can send custom HTTP bodies with `http-post`:

- Build a payload field using `add` (strings support `${}` runtime expansions).
- Send that field via `http-post.body.field`.

```
actions:

  • add:



output-fields: payload: | token=${secret|pushover/token}&user=${secret|pushover/user_key}
message=Hello from LyftData! host=${host||unknown}


output: http-post: url: https://api.example.com/ingest headers: Content-Type: application/x-www-
form-urlencoded body: field: field: payload
```

Handy toggles and safeguards

- `remove`: delete source fields once parsing succeeds.
- `suppress-warnings`: silence parsing warnings when a data source is noisy.
- `drop` (extract): stop events when validation fails.