LYFTDATA PRODUCT DOCUMENTATION

Operate and Scale

Daily operations, monitoring, scaling, backup, security, telemetry, worker enrollment, and support signals.

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

Contents

Operate and Scale

Operate & scale the platform

Source: operate/overview.md

LyftData operations focus on keeping the control plane healthy, the job fleet productive, and telemetry flowing to the right places. Use this page as the jumping-off point for your runbooks.

Daily checklist

- Confirm the server is reachable (for example `GET /api/liveness`) and that you can sign in.
- Watch the live job status feed for stalled deploys, long retries, or sudden error spikes.
- Track worker health in the UI; investigate offline workers and growing backlogs quickly.
- Review errors and warnings in Logs & Issues and in your host logging system (systemd journal, Windows Event Log, or your central logging sink).

Runbooks by theme

- **Daily operations** - Daily operations playbook keeps the control plane healthy with checklists and drills.
- **Observability & alerts** - Monitoring LyftData covers metrics, dashboards, and alert wiring.
- **Logs and live events** - Logs & Issues and Messages are your first stop for triage.
- **Resilience & recovery** - Backup & recovery explains snapshot cadence, restores, and disaster recovery tests.
- **Account recovery** - Reset an admin password covers the break-glass host-side flow when the only admin account is locked out.
- **Worker provisioning** - Worker auto enrollment covers shared-secret bootstrap flows and what to disable afterwards.
- **Capacity planning** - Scaling LyftData walks through worker sizing, channel fan-out strategies, and deployment hygiene.
- **Security posture** - Security hardening documents TLS, secret rotation, and RBAC guidance.
- **Telemetry** - Telemetry explains what LyftData collects locally and how to access it.

Releases and change management

- Before upgrades, note your current version (`lyftdata --version`) and review the release notes.
- Use the downloads portal for current builds and checksums.
- Keep a simple change log for your environment (what changed, who approved it, and how to roll back).

Where to go next

- Follow the Daily operations playbook for your everyday checklist and weekly reviews.
- Set up dashboards using the Monitoring guide and plan capacity with the Scaling runbook.
- Harden your deployment via Security guidance and Backup & recovery.
- Track upcoming changes in the release notes and communicate upgrades with stakeholders.

Backup and Recovery

Back up LyftData configurations and data, and recover quickly when something goes wrong

Source: operate/backup.md

A resilient LyftData deployment needs regular backups and rehearsed recovery procedures. This runbook captures what to back up, how often to do it, and how to validate restores.

What to back up

Component	Why it matters	Suggested cadence
Job definitions (export)	Portable backup of pipeline definitions, independent of server state	Nightly
Server staging directory (`LYFTDATA_STAGING_DIR`)	Primary server state (job metadata, deployments state, stored telemetry)	Daily snapshots
Server startup config	How the server is started (systemd unit/launchd, container manifests, env files)	Weekly or whenever changed
Worker startup config + jobs directory (`LYFTDATA_JOBS_DIR`)	Worker identity/credentials and local state	Weekly (or treat as rebuildable)
TLS material	Certificates/keys if using built-in TLS or a reverse proxy	Aligned to rotation schedule
Secrets and master keys	Worker API keys, variables/credential-manager keys, license key	Aligned to rotation schedule
Logs / audit exports	Forensics and compliance outside LyftData retention windows	Daily export with 30–90 day retention

Quick export commands

```
# Export all jobs to a dated directory and compress it
EXPORT_ROOT=backups/jobs-$(date +%Y%m%d) lyftdata jobs export --dir "$EXPORT_ROOT" tar -czf
"${EXPORT_ROOT}.tar.gz" -C "$(dirname "$EXPORT_ROOT")" "$(basename "$EXPORT_ROOT")"
```

```
STAGING_DIR="${LYFTDATA_STAGING_DIR:-/var/lib/lyftdata-server}" sudo systemctl stop lyftdata-
server sudo tar -czf "backups/server-staging-$(date +%Y%m%d).tar.gz" -C "$(dirname
"$STAGING_DIR")" "$(basename "$STAGING_DIR")" sudo systemctl start lyftdata-server
```

```
sudo tar -czf "backups/server-service-config-$(date +%Y%m%d).tar.gz" \
/etc/systemd/system/lyftdata-server.service \ /etc/default/lyftdata-server
```

Store backups in two places: fast local storage for quick restores and offsite object storage for disasters. Encrypt sensitive archives before upload.

Automate daily configuration backup

```
#!/usr/bin/env bash
set -euo pipefail BACKUP_DIR="/var/backups/lyftdata" DATE=$(date +%Y%m%d-%H%M%S) mkdir -p
"$BACKUP_DIR"

EXPORT_DIR="$BACKUP_DIR/jobs-$DATE" lyftdata jobs export --dir "$EXPORT_DIR" tar -czf
"$BACKUP_DIR/jobs-$DATE.tar.gz" -C "$BACKUP_DIR" "jobs-$DATE" rm -rf "$EXPORT_DIR"
```

```
STAGING_DIR="${LYFTDATA_STAGING_DIR:-/var/lib/lyftdata-server}" systemctl stop lyftdata-
server tar -czf "$BACKUP_DIR/server-staging-$DATE.tar.gz" -C "$(dirname "$STAGING_DIR")"
"$(basename "$STAGING_DIR")" systemctl start lyftdata-server tar -czf "$BACKUP_DIR/server-
config-$DATE.tar.gz" /etc/systemd/system/lyftdata-server.service /etc/default/lyftdata-server

find "$BACKUP_DIR" -type f -mtime +30 -delete
```

Validate backups

Run automated checks to ensure archives are usable:

```
#!/usr/bin/env bash
BACKUP="$1" TEMP_DIR=$(mktemp -d) trap 'rm -rf "$TEMP_DIR"' EXIT
```

```
tar -xzf "$BACKUP" -C "$TEMP_DIR" EXPORT_DIR=$(find "$TEMP_DIR" -maxdepth 1 -type d -name
'jobs-*' -print -quit) if [ -z "$EXPORT_DIR" ]; then echo "could not locate exported jobs
directory" >&2 exit 1 fi
```

```
lyftdata jobs import --dry-run --dir "$EXPORT_DIR"
```

Schedule validation weekly; surface failures in monitoring.

Recovery steps

1. Restore the server:

- Rebuild host/container and reinstall the same (or compatible) LyftData version.
- Restore your startup config (systemd unit/launchd, env file, container manifests, TLS material).
- Restore the server staging directory (`LYFTDATA_STAGING_DIR`) and restart the service.

2. Re-register workers:

- Reinstall `lyftdata-worker` on each worker host.
- Restore each worker's config and jobs directory (`LYFTDATA_JOBS_DIR`) so it retains its identity and credentials.
- Confirm workers appear in the Workers UI or via `lyftdata workers list`.

3. Redeploy jobs:

- Extract the latest job archive and run `lyftdata jobs import --dir <path> --update`
- Confirm job state matches expectations in the UI

4. Validate:

- Confirm liveness (`GET /api/liveness`) and sign-in.
- Run canary jobs or sample pipelines.
- Watch the first hour of Logs & Issues for new errors.

Disaster recovery tips

- Keep infrastructure-as-code scripts handy to recreate servers and workers in new regions.
- Document RPO/RTO expectations (e.g., 1 hour of data loss max, four-hour recovery window).
- Test restore procedures quarterly to ensure runbooks stay current.

See also: Monitoring LyftData for detection signals and the troubleshooting guide for incident triage.

Daily Operations

Routine health checks, weekly reviews, and incident drills for LyftData

Source: operate/daily-operations.mdx

This playbook is aimed at operators and SREs who need a repeatable cadence for keeping LyftData healthy. Use it alongside the detailed runbooks in **Operate & Scale**.

Daily checklist (15 minutes)

- [ ] Review the dashboard for worker status, queue depth, and recent alerts.
- [ ] Scan the job status feed for stalled deployments or retry storms.
- [ ] Check licensing state in the UI or via `lyftdata license show` so Community Edition limits or expiring keys are flagged early.
- [ ] Spot-check server and worker logs for new error signatures (`journalctl -u lyftdata-server`, worker logs).

> tip

Automate these checks where possible—scripts that call `GET /api/liveness` and `GET /api/health` (with an admin bearer token, plus worker/job queries) reduce manual toil. Prefer `https://…` by default; add `-k` only for evaluation/self-signed environments.

> Note

Weekly tasks

- [ ] Review worker utilization trends in the monitoring guide and plan scale-up if CPU or queue depth is trending high.
- [ ] Validate backups and retention using the backup & recovery checklist.
- [ ] Audit user accounts and API keys (rotate stale credentials, remove unused workers).
- [ ] Capture notable changes (new connectors, job migrations) in your team runbook.

Before deploying changes

- [ ] Stage jobs and verify via **Run & Trace** in lower environments.
- [ ] Check the release notes for upgrade guidance or known issues.
- [ ] Ensure alerting/metrics dashboards reflect any new jobs or channels.

Incident drills & readiness

- [ ] Rehearse the escalation path for worker failures (who owns remediation?).
- [ ] Test the process for draining jobs and restarting workers safely.
- [ ] Validate that error budgets or SLIs are defined and monitored (pair with scaling guidance).

Resources

- Monitoring runbook: /operate/monitoring

- Scaling playbook: /operate/scaling
- Backup & recovery: /operate/backup
- Security hardening: /operate/security

Keep this page bookmarked as the starting point for day-to-day operations and link it in your incident response handbook.

Job Status Feed

Legacy job status feed and telemetry integration

Source: operate/job-status-feed.md

LyftData keeps job and worker status **live** in the UI so you can see when a job is staged, deployed, running, and healthy without refreshing the page.

Where status appears

- **Jobs list**: quick status badges and last activity.
- **Job detail**: live run history, recent issues, and per-worker deployment state.
- **Workers**: which jobs are running on each worker and whether the worker is healthy.

Troubleshooting

If status looks stale or inconsistent:

1. **Refresh the page** and confirm you are still signed in. 2. **Check connectivity**: the UI relies on long-lived HTTP connections for live updates. Reverse proxies, VPNs, and corporate gateways can break streaming connections. 3. **Check permissions**: if you are using an API token, make sure it has read access to jobs/workers and live status. 4. **Check the server logs** for API errors during status updates.

If you are building external monitoring, prefer the metrics and logs guidance in Monitoring and Logs and issues rather than depending on UI-oriented status streams.

Logs and Issues

Understand how runtime warnings flow into the UI surfaces you use to triage problems.

Source: operate/logs-and-issues.md

Use **Logs** for historical queries, **Problems/Issues** for “what’s broken right now”, and **Messages** when you need a live view. This page describes how these UI surfaces work together and how to troubleshoot common gaps. For job-driven notification patterns (including `message` actions with `log-event: true`), see Notifications & Alerts.

Logs

Open **Observe** → **Logs** (or visit `/logs`) to query historical log records emitted by the server and workers.

Common filters:

- **Time range**: widen it first if you see “no logs”.
- **Severity**: focus on Warning/Error when triaging failures.
- **Worker** and **Job**: narrow to the component you’re investigating.

You can also fetch worker logs via the CLI for quick spot-checks:

```
lyftdata workers logs <worker-id> --limit 200
```

Problems / Issues

Open **Observe** → **Problems** (or use the Issues panel on a job) to see actionable warnings and errors that need attention. Problems/Issues are designed for triage:

- Start with the most recent Error entries.
- Use the job/worker links on each issue to jump directly to context.
- Pivot into **Logs** to see surrounding events and repeated failures.

Messages (live)

Open **Observe** → **Messages** for a live stream of job and worker activity. It’s most useful when:

- you are deploying or stopping jobs and want immediate feedback,
- you are debugging intermittent warnings,
- you need to correlate worker status changes with job failures.

See Messages for usage and filtering tips.

Troubleshooting

- **No logs or issues show up**: widen the time range, confirm the worker is online, and verify your token has read access.

- **Live views look stale**: long-lived HTTP connections can be interrupted by reverse proxies, VPNs, and corporate gateways; try a direct connection to the server or adjust proxy settings.

Related reading

- Monitoring
- Notifications & Alerts
- Telemetry
- Troubleshooting

Message Bus (Aggregator)

How internal messages flow between server, workers, and jobs, and how to use them for coordination and observability.

Source: operate/message-bus.md

LyftData has an internal message bus (the **aggregator**) used for coordination and observability across the server, workers, and jobs. It is separate from channel-based **data flow** (events moving over workflow channels).

Use messages when you need a **wake-up signal**, a **runtime notification**, or a **control-plane side channel**. Use workflow channels/transports for **bulk data movement**.

What A Message Contains

Messages carry:

- **Kind**: `system` vs `user`.
- **Source**: which process generated it (`server`, `worker`, `job`).
- **Type**: a stable message type (for example `job-run-started`, `job-run-ended`, `user-generated`, `user-alert`).
- **Optional tag**: a user-defined string used for selective wake-ups and routing. Tags apply to user-generated messages.
- **Optional payload**: a "publicly exposed" message payload. Some internal message types intentionally do not expose full payloads.

In the UI, many of these show up in **Observe → Messages** and **Observe → Logs / Issues**. In jobs, you consume them via the `internal-messages` input.

Emit Messages From A Job

There are two different DSL primitives that emit messages.

`message` Action (Human-Facing Notifications)

Use the `message` action when you want a runtime **alert/info notification** (optionally also written into job logs).

```
actions:

  • message:


condition: "true" notification-type: alert message-content: "Open listening port detected"
log-event: true
```

This is the right choice for operator-facing warnings and breadcrumbs.

`message` Output (Machine-Facing Coordination)

Use the `message` output when you want to emit a **structured payload** that other jobs can subscribe to.

```
output:
message: tag: stage1-done
```

By default, the output emits a `user-generated` message whose payload is the job's output event JSON.

> note

The message output only emits JSON. If the current event is not JSON, the published `job_event` becomes `null`. If you use `set-variable-name`, the variable value is the JSON event stringified (for example `{"ok":true}`), not a structured object.

> Note

If you set `set-variable-name`, the output emits a variable update instead of a user-generated message:

```
output:
message: set-variable-name: MY_DYNAMIC_VALUE
```

Consume Messages In A Job (`internal-messages`)

The `internal-messages` input is the "subscribe to the bus" primitive. It is event-driven: as matching messages arrive, they become input events for the job to process.

Example: wake up only on a tagged user-generated message from a specific upstream job.

```
input:
internal-messages: filter-kind: user filter-job: stage1_job filter-type: [user-generated]
filter-tag: stage1-done
```

Input Event Schema

Each consumed message becomes one JSON event with a stable top-level shape:

- `id`: message ID
- `version`: message-attributes schema version (currently `0.1`)
- `nanoseconds_since_epoch`: message timestamp
- `kind`: `system` or `user`
- `source`: `server`, `worker`, or `job`
- `message_type`: a kebab-case type (for example `user-generated`, `job-run-started`)
- `job_name` (optional): originating job name
- `worker_id` (optional): originating worker ID
- `message_data` (optional): a public payload, when the message type exposes one

For `user-generated` messages, `message_data` includes the tag and the original payload:

```
{
"id": "…", "version": "0.1", "nanoseconds_since_epoch": 1739370000000000000, "kind": "user",
"source": "job", "message_type": "user-generated", "job_name": "stage1_job", "worker_id":
"worker-1", "message_data": { "type": "UserGeneratedMessage", "tag": "stage1-done",
"job_event": { "ok": true, "count": 42 } } }
```

> note

`filter-tag` only matches user-generated messages, but many other message types are still useful (job lifecycle, warnings, trace export, variable changes). Use `filter-type`, `filter-source`, `filter-job`, and `filter-worker` to keep subscriptions tight.

> Note

Message Payloads vs `internal-messages` Events

Jobs see message information in two different ways, and they use different expansion paths:

- **Message triggers** (`input.<x>.trigger.message`) expose the triggering message's **public payload** via `${msg|...}`.
- The **`internal-messages` input** delivers a full message wrapper as the event (including `message_type`, `kind`, `source`, and `message_data`). In this mode, `${msg|...}` is not set; read from the event fields instead.

Example: reading a Trigger invocation ID.

```
# Message trigger (use ${msg|...})

  • add:


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

```
# internal-messages input (use normal event fields)

  • add:


output-fields: invocation_id: "${message_data.job_event.invocation_id}" message_type:
"${message_type}"
```

Message Types Worth Knowing

The bus includes many message types, but teams usually build around a small, practical set.

Type (`message_type`)	Typical source	What it's good for	Notes

`user-generated`	job	Job-to-job coordination	Taggable + filterable (`filter-tag`). Payload is the emitting job's event (`message_data.job_event`).
`user-alert` / `user-notification`	job	Operator-facing warnings and breadcrumbs	Usually produced by the `message` action. Useful for humans; not usually for automation.
`job-run-started` / `job-run-ended`	job	Run boundaries	Useful for watchers or to correlate other telemetry.
`job-runtime-error` / `job-errors`	job	Error evidence	Useful for alerting and auto-triage.
`update-variable`	job	Variable updates	Produced by `output.message.set-variable-name` .
`trigger-response`	job	Completing a Trigger invocation	See Triggers for the response envelope and how it shapes invocation results.
`trigger-proxy-response`	job	Capturing a proxied HTTP response	Typically auto-emitted by `http-post` when a `trigger_proxy` context exists; recorded into the Trigger invocation result.
`deployment-phase`	server	Deployment progress	Useful for "what's happening right now?" operational dashboards.

Example: subscribe to job runtime errors for one job:

```
input:
internal-messages: filter-kind: system filter-job: important_job filter-type: [job-runtime-
error, job-errors]
```

Patterns That Build On Messages

"Tagged wake-ups" (workflow message edges)

Message edges in workflows are control-flow wiring ("wake up when signalled"), not data transport. Conceptually:

- Upstream step emits a tagged user-generated message (`output.message.tag`).
- Downstream step subscribes with `input.internal-messages.filter-tag` .

This is one way to build "run B after A has committed its durable output" without polling.

Fan-in joins (OR vs AND)

If you need "run when *all* of these upstream steps finished", model it explicitly:

- Subscribe to the relevant message types/tags.
- Track prerequisites in Worker KV.
- Emit a single tagged message when the join is satisfied.

This keeps coordination state explicit and avoids accidental re-triggers.

Triggers (UI + MCP)

The Triggers registry (UI + MCP dynamic tools) dispatches user-generated messages to a target job. For the operator-facing surface and MCP behavior, see Triggers.

Where To Go Next

- Messages (UI) for the live feed and filtering ergonomics.
- Notifications & Alerts for `message` action patterns.
- Deployments Core Concepts for message edges vs channel edges.
- Triggers for curated invocation surfaces that dispatch messages.

Messages

Stream live job, worker, and warning events from the runtime.

Source: operate/messages.md

The **Messages** view gives you a live feed of everything the runtime is broadcasting over the Messages stream. Use it to watch job progress, confirm workers are healthy, or capture warnings the moment they are emitted. You can pause the feed, filter it, or export the captured data for later analysis.

> tip

If you're trying to build message-driven coordination (emit/subscribe, tags, `internal-messages` input), start with Message Bus.

> Note

Subscribe to live channels

The panel on the left lists the available channels. Toggle a channel to begin streaming its messages:

- **Job Status** – deployment and lifecycle updates for every job (`Deployed`, `Shutting down`, `Failed`, etc.).
- **Worker Status** – heartbeats and state transitions for connected workers.
- **Job Messages** – application logs that a job emits. Enter one or more job names to subscribe; each job appears as its own channel so you can follow multiple pipelines at once.
- **Errors & Warnings** – the same warnings surfaced in Logs & Issues, delivered as soon as the runtime emits them.
- **Job Execution Tracking** – per-run execution milestones (queued, running, completed) for detailed diagnostics.
- **Debug: All Messages** – a firehose feed intended for short-term debugging. Enable it only when you need full visibility, because it can grow noisy in busy environments.

Active channels stay selected until you disable them. The page remembers your choices for the current browser tab, so you can reload the view without losing subscriptions. This state is not shared across new tabs/windows. If you prefer to start with a clean slate, click **Dismiss** on the restore prompt or use **Clear** to wipe the buffer.

Filter and search

- Use the **Filter by channel** checkboxes to focus on a subset of streams while keeping others running in the background.
- The search box matches channel names, message types, and the text within each payload. Searches apply instantly and leave the underlying subscriptions untouched.
- The counters above the message list show how many messages are currently buffered and the per-minute rate so you can spot bursts at a glance.

Inspect message details

Click any message to open the detail drawer. The UI formats common payloads so you can read them quickly:

- Job and worker status messages are summarised with badges for worker, job, and state transitions.
- Errors & warnings are grouped into dedicated lists with badges for the affected job or worker and a direct **View Logs** shortcut.
- Job messages show the structured fields that the job emitted; open **Raw JSON** to copy the original payload.

Every badge in the drawer links back into the rest of the UI. Use **Job** to jump to the job definition, **Worker** to open the worker detail page, or **View Logs** to pivot into the Logs viewer with the worker or job filters pre-filled.

Debugging Triggers (message-driven workflows)

When a Trigger invocation looks stuck in `dispatched`, or you are validating response-slot/proxy result shaping, use **Debug: All Messages** to confirm the correlated message traffic.

1. Enable **Debug: All Messages** for a short window. 2. Search for the `invocation_id` (from the Trigger invocation record). 3. Look for the dispatch `user-generated` message whose payload includes `job_event.type: "trigger_invoke"`. 4. If the Trigger uses `response_slot`, look for a `trigger-response` message with matching `invocation_id` and `response_slot`. 5. If the Trigger uses proxy mode, look for a `trigger-proxy-response` message with matching `invocation_id` and `response_slot`.

If you see `trigger_invoke` but no response messages, the most common causes are: the target job is not message-triggered on the right tag/type, or it emitted a malformed response envelope (extra top-level keys, wrong `tenant_id`, or wrong slot). See Triggers for the strict response envelope rules.

Pause, export, and restore

- Hit **Pause** to stop collecting new entries while you investigate the current buffer. Resume when you want to catch up.
- **Clear** wipes the buffer immediately and resets the persisted state. Choose **Export JSON** to download everything currently in the buffer.
- The viewer keeps the most recent 500 messages in browser storage. When you reload or return to the page in the same tab, you can restore previous subscriptions and continue from where you left off.

Deep links and pivots

Share links to specific channels using query parameters:

- `?job=<name>` subscribes to that job's messages.
- `?worker=<id>` enables the worker status channel.

You can combine both parameters to prime the page for a particular investigation. The buttons in each message detail also use these parameters when they open the Logs view, so you always land on the relevant subset.

Troubleshooting

- If you do not see new messages, confirm the channel toggle is enabled and the **Pause** button is not active.
- Large exports or the Debug feed can fill the buffer quickly. Clear the buffer or turn off unused channels to keep the stream readable.
- For end-to-end context on how warnings flow into Logs & Issues, see Logs & Issues.

Monitoring LyftData

Monitor servers, workers, and jobs with built-in dashboards, CLI tooling, and external integrations

Source: operate/monitoring.md

Keeping LyftData healthy in production means watching the control plane, the workers, and the jobs they execute. Use the practices below to get fast feedback when something drifts from normal.

Built-in observability (UI)

- **Dashboard**: high-level health, recent job activity, and key charts.
- **Jobs**: per-job deploy state, run history, and issues.
- **Workers**: which workers are online, what they're running, and their current limits.
- **Metrics Explorer**: query stored metrics by job/worker and time range.
- **Observe → Logs / Problems / Messages**: historical logs, "what's broken right now", and a live event stream.

Quick checks from the shell

```
# Liveness (unauthenticated). Add `-k` if you are using the default self-signed cert.
curl -fsS https://<server>:3000/api/liveness
```

```
curl -fsS -H "Authorization: Bearer <admin-token>" \ https://<server>:3000/api/health | jq
'{status: .status, version: .version}'
```

If you have the CLI configured (see CLI reference), you can also run:

```
lyftdata doctor
lyftdata workers list lyftdata jobs list
```

Key signals to alert on

Signal	Investigate when	Typical response
Workers offline	Any production worker is unexpectedly offline	Check connectivity/TLS, restart worker, or replace host
Backlog growing	Queue depth or "pending work" trends up over several minutes	Add workers, reduce job cadence, or tune expensive steps

Deployments stuck	Jobs sit in staged/deploying states unusually long	Check worker availability, review Issues, redeploy
Error spikes	Errors or retries rise suddenly	Investigate downstream systems before scaling
Disk pressure	Staging/log storage approaches your limits	Increase disk, tighten retention, or move staging to a larger volume
License risk	License nearing expiry or limits being hit	Resolve licensing before it blocks production runs

Most teams start with the UI dashboards and add alerts as the “normal” baseline becomes clear for their workloads.

Alerting and integrations

- Forward host-level logs to your central platform (Loki/ELK/Splunk/etc.) for long-term retention and correlation.
- For LyftData telemetry (logs/issues/metrics), prefer the UI surfaces and CLI (`lyftdata workers logs <worker-id>` , `lyftdata workers metrics <worker-id>`).
- If you run collectors from fixed IPs, consider the server allowlist (`--whitelist` / `LYFTDATA_API_WHITELIST`) which enables read-only access to selected worker telemetry endpoints when the collector sends `Authorization: WHITELIST` (see Security hardening).
- Notify on-call channels when pipelines stall, workers flap, or error budgets are exceeded.

Health checks and diagnostics

- **HTTP probes**: `GET /api/liveness` for basic reachability; `GET /api/health` for a richer status summary.
- **CLI**: `lyftdata doctor` and `lyftdata server health` for guided checks.
- **Synthetic jobs**: schedule a tiny canary job that runs every few minutes and alerts if it fails.

Troubleshooting signals

- Jobs stuck in *Staged*: verify target workers are online and the scheduling queue is clear.
- Rising retries on a connector: inspect connector logs and downstream APIs for throttling or auth errors.
- High backlog with low CPU: scale out workers or increase job concurrency; see the Scaling guide.

What's next

- Hardening operations continues in the rest of the Operate section.
- For immediate triage, pair this guide with the Troubleshooting reference.

Notifications and Alerts

Configure runtime notifications, route security alerts into Issues/Logs, and consume notification streams from MCP clients.

Source: operate/notifications-and-alerts.md

LyftData supports general-purpose runtime notifications that can be surfaced in three places:

- **Dashboard toast** for immediate operator feedback.
- **Issues/Problems** when the notification is warning/error severity.
- **Logs** when a job emits the notification with `log-event: true`.

Use this page for practical setup and troubleshooting.

> tip

Need the broader assistant story, including how MCP sessions authenticate, connect to remote LyftData servers, and expose live activity? Start with AI and MCP Overview.

> Note

1. Emit notifications from a job

Use the `message` action:

```
actions:

  - message:


notification-type: alert message-content: "Open listening port detected: {{ bind_address }}:
{{ port }}" log-event: true
```

Guidance:

- `notification-type: alert` raises warning-style notifications.
- `notification-type: info` raises informational notifications.
- `log-event: true` is required when you want the same signal in **Issues** and **Logs**.

2. Deploy the security alert catalog job

The local catalog includes **`security-open-port-alert`** (library job + blueprint) for open-port detection.

Typical flow:

1. Open **Deployments** → **Catalog**. 2. Select **Jobs** and search for `security-open-port-alert` (or choose the matching security blueprint). 3. Click **Plan & register** and configure:

- `poll_interval` (for example `60s`)
- `allowed_ports_regex` (for example `22|80|443`)
- `tenant_label`

- `site_name`

4. Apply the deployment (or create a job from the library job template).

When a non-allowlisted port is detected, the job emits an alert notification and logs the event for Issues/Logs workflows.

3. MCP notification consumption (pull + push)

Pull (polling fallback)

Use `job_notifications_recent` when your MCP client cannot open websocket side channels:

```
{
"job_names": ["security-open-port-alert"], "window_minutes": 30, "limit": 200 }
```

Push (dedicated websocket)

Use `mcp_notifications_transport_get` to discover runtime endpoints:

- `transport.recent_http` → `/api/mcp/notifications/recent`
- `transport.subscribe_websocket` → `/api/mcp/notifications/subscribe`

Payload fields:

- `message_type` (`UserNotification` or `UserAlert`)
- `severity` (`info` or `warning`)
- `message`
- `timestamp` / `timestamp_ns`
- optional `job_name`, `worker_id`, `worker_name`

4. Troubleshooting

- **Toast appears, but no Issue/Log entry**: ensure `log-event: true` is set on the `message` action.
- **No dashboard toast**: verify realtime updates are enabled and websocket traffic to `/api/notifications/subscribe` is not blocked by proxy/network policy.
- **No MCP push events**: confirm admin-scoped auth for `/api/mcp/notifications/*` and verify websocket-capable MCP client behavior; otherwise use `job_notifications_recent` polling.
- **Unexpected alert volume**: tighten `allowed_ports_regex` and/or increase `poll_interval`.

Reset an Admin Password

Recover access when the LyftData admin password is lost

Source: operate/reset-admin-password.mdx

Use this runbook when the `admin` password is lost and you cannot sign in through the normal login flow.

> Scope

This runbook applies to Linux deployments managed by `systemd`.

If you run LyftData under another service manager, use the same reset flow but adapt the environment file, service override, and restart steps to your platform.

> Note

What this guide does

LyftData does not currently provide a self-service "forgot password" flow on the login page. Password reset is an operator task.

- If another admin can still sign in, use the UI to set a new password for the affected user.
- If no admin session remains, reset the built-in `admin` account from the server host by restarting the service once with `--reset-admin-password` and a new `LYFTDATA_ADMIN_INIT_PASSWORD`.

> caution

This procedure assumes host access to the server that runs LyftData. It is a break-glass workflow for operators, not an end-user action.

> Note

Before you start

- Choose a strong replacement password.
- Admin passwords must be at least 12 characters and include:
- one uppercase character
- one lowercase character
- one digit
- Identify your server host, service name, environment file, and binary path.

Two common layouts are:

- Documentation install example:
- service: `lyftdata-server`
- env file: `/etc/default/lyftdata-server`
- binary: `/usr/sbin/lyftdata`
- Reverse-proxied production install:
- service: `lyftdata`
- env file: `/etc/lyftdata/lyftdata.env`

- binary: `/opt/lyftdata/current/bin/lyftdata`

The steps below use the second layout. Substitute your own paths if your installation differs.

Fast path when another admin can still sign in

If another admin session is available:

1. Sign in to the UI with that working admin account. 2. Open **Settings > Users**. 3. Edit the affected user and set a new password. 4. Sign out and confirm the updated password works from a fresh login screen.

Use the host-side reset only when no working admin session remains.

Break-glass reset on Linux (`systemd`)

1. Connect to the host and stage a new password

```
ssh <server-host>
sudoedit /etc/lyftdata/lyftdata.env
```

Add or update:

```
LYFTDATA_ADMIN_INIT_PASSWORD=YourNewStrongPass123
```

> note

Keep this value only as long as needed for the reset window. Remove it again after access is restored.

> Note

2. Add a one-restart systemd override

Create a runtime-only override for the service:

```
sudo systemctl edit --runtime lyftdata
```

Paste:

```
[Service]
ExecStart= ExecStart=/opt/lyftdata/current/bin/lyftdata run server --bind-address
127.0.0.1:3000 --disable-tls --reset-admin-password
```

This keeps the normal service definition intact and adds the reset flag for the next restart only.

3. Restart the service

```
sudo systemctl restart lyftdata
sudo systemctl status lyftdata --no-pager
```

At startup, LyftData deletes and recreates the built-in `admin` user using the password from `LYFTDATA_ADMIN_INIT_PASSWORD`.

4. Sign in with the new password

Open the normal server URL and log in again as `admin`.

For example, if your installation is reverse-proxied behind a public hostname, log in at that normal server URL.

Examples:

- `https://lyftdata.example.com`
- `https://localhost:3000`

If sign-in still fails, inspect the service logs:

```
sudo journalctl -u lyftdata -n 100 --no-pager
```

Common causes are:

- the new password does not meet the admin password policy
- the service override was not applied
- the environment file was edited incorrectly

5. Remove the temporary reset configuration

Once you have confirmed the new password works:

```
sudo systemctl revert lyftdata
sudoedit /etc/lyftdata/lyftdata.env
```

Remove the temporary `LYFTDATA_ADMIN_INIT_PASSWORD` value unless you intentionally keep it in your service configuration.

Then restart back onto the normal service definition:

```
sudo systemctl restart lyftdata
```

Adapt this to another install

If you followed the generic Linux install guide instead of the reverse-proxied production layout above, swap the service/env/binary values:

- service: `lyftdata-server`

- env file: `/etc/default/lyftdata-server`
- binary: `/usr/sbin/lyftdata`

The reset pattern stays the same:

1. Set `LYFTDATA_ADMIN_INIT_PASSWORD` in the service environment file. 2. Restart once with `--reset-admin-password`. 3. Verify login. 4. Remove the temporary reset settings.

Related guides

- Troubleshooting
- Server Installation on Linux
- Security Hardening

Scaling LyftData

Scale servers and workers so pipelines stay fast and reliable

Source: operate/scaling.md

Use this playbook when jobs start to backlog, worker utilisation spikes, or new workloads arrive. LyftData scales horizontally by adding workers and vertically by increasing resources per node.

> note

External workers require a license. Community Edition includes only the built-in worker, so scaling is primarily vertical (bigger box) unless you upgrade.

> Note

Know when to scale

Monitor these signals (see Monitoring LyftData for how to collect them):

Metric	Action threshold	Typical response
Worker CPU usage	> 80% sustained	Add workers or increase CPU
Worker memory usage	> 85% sustained	Increase memory or adjust batch size
Job queue length	> 100 pending	Add workers / tune scheduling
Job runtime	2× baseline	Profile actions, add capacity
Error/retry rate	> 5%	Investigate downstream systems before scaling

Horizontal scaling (add workers)

- Provision new worker hosts or containers as close to your data sources as possible to minimise latency.
- For licensed deployments, create a worker ID in **Workers** and an API key in **Settings → API Keys** (see Workers).
- Point the worker at the control plane and supply credentials:

```
  lyftdata-worker \
--url https://lyftdata.example.com \ --worker-id worker-02 \ --worker-api-key
"$LYFTDATA_WORKER_API_KEY"
```

- For fleets that rely on auto enrollment, configure auto-enrollment on the server, then start each worker with `LYFTDATA_AUTO_ENROLLMENT_KEY` instead of pre-issuing API keys.
- Verify the worker registers successfully via the UI or `lyftdata workers list`.

Worker configuration checklist

- `LYFTDATA_URL` – HTTPS URL for the control plane.

- Identity – either `LYFTDATA_WORKER_ID` + `LYFTDATA_WORKER_API_KEY`, or `LYFTDATA_AUTO_ENROLLMENT_KEY` with an optional `LYFTDATA_WORKER_NAME`.
- `LYFTDATA_WORKER_LABELS` – optional comma-separated descriptors surfaced in the UI for filtering and documentation.
- `--limit-job-capabilities` – start the worker with this flag when it should only accept basic connectors (handy for hardened ingress zones).
- Systemd or your orchestrator should manage the worker process so it restarts automatically after upgrades or crashes.

Workers advertise their capabilities back to the server. The UI shows each worker's advertised limits (including maximum concurrent jobs). In Community Edition, scaling remains constrained by built-in-worker-only operation, licensed-feature gating, and the daily processing cap.

Vertical scaling (bigger boxes)

When horizontal scale is not practical:

- Increase CPU/memory allocations for the server or existing workers.
- Ensure disks that host the staging directory have headroom; adjust `--db-retention-days` or `--disk-usage-max-percentage` if cleanup is too aggressive.
- Revisit batch sizes and action efficiency so additional resources translate into throughput.

Automate worker scaling

- Poll server health and worker status on a schedule (via the UI, CLI, or API) and persist readings so you can compare moving averages instead of reacting to single spikes.
- Feed those metrics into your autoscaling tool (Kubernetes HPA, AWS ASG, Nomad autoscaler, etc.). A typical policy adds a worker when queue depth stays above your threshold for several minutes.
- During scale-in, stop workers gradually and watch for new backlog or error spikes.
- Alert whenever automation fails (for example, API calls start returning errors or workers flap repeatedly) so operators can intervene manually.

Placement strategies

- **Region-aware**: co-locate workers with data sources to reduce egress and latency.
- **Capability-aware**: dedicate workers to specialised connectors or sensitive networks and label them so runbooks show where to deploy specific jobs.
- **Redundancy**: keep spare capacity in another availability zone for failover scenarios.

Keep jobs efficient

Scaling is easier when jobs are lean:

- Split large multi-purpose jobs into smaller stages connected by worker channels.
- Use `filter` early to drop unnecessary records.
- Profile Lua scripts and external lookups; cache results where possible.
- Adjust scheduling cadence (see Advanced scheduling) to smooth bursty workloads.

Review after scaling

- Validate that job runtimes and queue lengths return to normal.
- Update documentation/runbooks with the new topology.
- Revisit resource budgets monthly; scale down when demand shrinks to save cost.

Pair this guide with the backup and security runbooks to keep operations predictable as the platform grows.

Security Hardening

Lock down your LyftData installation with network, access, and data protections

Source: operate/security.md

These practices harden a LyftData deployment so only trusted users and systems can interact with it.

TLS (HTTPS)

LyftData serves HTTPS by default (self-signed if you do not provide TLS material). For production, install a trusted certificate (or terminate TLS behind a reverse proxy).

- **Built-in TLS**: run the server with `--tls-cert` and `--tls-key` (or `LYFTDATA_TLS_CERT` / `LYFTDATA_TLS_KEY`).
- **Reverse proxy**: terminate TLS in a proxy and run the server with `--disable-tls`.
- **Do not use insecure TLS in production**: `--tls-insecure` / `LYFTDATA_TLS_INSECURE` applies to CLI and worker-to-server clients and is intended for evaluation/self-signed environments only.

See Networking & TLS for end-to-end verification checks.

Network boundaries

- Keep the server bound to `127.0.0.1:3000` unless you explicitly need remote access.
- If you expose the server port, restrict it to management CIDRs (VPN/bastion) and block public networks.
- Segment traffic where possible: separate control plane and worker hosts and allow only required ports.

Example (UFW):

```
sudo ufw default deny incoming
sudo ufw allow 22/tcp sudo ufw allow from 10.0.0.0/8 to any port 3000 proto tcp sudo ufw
enable
```

Accounts and secrets

- Run server and workers as dedicated service accounts with isolated data directories (see Prerequisites).
- Store secrets (admin bootstrap password, worker API keys) in service environment files with tight permissions.
- Encrypt disks/volumes that store staging data, logs, and backups where possible.

For headless Linux servers, prefer explicit env-backed master keys for the variables store and credential manager instead of relying on an interactive OS keyring:

- `LYFTDATA_VARIABLES_MASTER_KEY_SOURCE=env`
- `LYFTDATA_CREDENTIAL_MANAGER_MASTER_KEY_SOURCE=env`

Keep the corresponding key values only in a root-owned env file with `0600` permissions.

`systemd` hardening (Linux)

For production Linux hosts, prefer a reverse-proxied `systemd` unit that:

- runs as a dedicated non-login account
- keeps the service bound to `127.0.0.1:3000`
- uses a root-owned environment file for secrets
- sets `UMask=0077`
- enables `NoNewPrivileges=true`, `PrivateTmp=true`, `PrivateDevices=true`, `ProtectSystem=strict`, and `ProtectHome=true`
- leaves `CapabilityBoundingSet=` empty and narrows `ReadWritePaths` to the service state directory

After changing the unit, validate both the service and the sandbox posture:

```
systemctl status lyftdata --no-pager
curl http://127.0.0.1:3000/api/liveness systemd-analyze security lyftdata.service
```

See Server Installation on Linux for a concrete hardened unit example.

Workers and enrollment

- Prefer unique API keys per worker and rotate them on a schedule.
- Use auto-enrollment (`LYFTDATA_AUTO_ENROLLMENT_KEY`) only on trusted networks; rotate or disable the shared secret after provisioning.
- Keep the server URL scheme consistent (`https://…` by default; `http://…` only when running the server with `--disable-tls`).

See Worker Authentication for the tradeoffs.

Optional: allowlisted collectors

If you need a trusted collector to scrape worker logs/metrics without an admin token, configure the server whitelist (`--whitelist` / `LYFTDATA_API_WHITELIST`) and keep the allowlist tight (specific source IPs only). This bypass applies only to selected worker read endpoints and still requires the `Authorization: WHITELIST` header.

Incident response checklist

1. Isolate impacted workers/servers from the network. 2. Rotate worker API keys, auto-enrollment secrets, and TLS certificates. 3. Restore from a known-good backup (see Backup & recovery). 4. Review logs and recent configuration changes to determine cause and scope.

Security is ongoing; integrate these hardening steps into your deployment automation and revisit them regularly.

Telemetry

Understand and manage the platform telemetry pipeline

Source: operate/telemetry.md

LyftData emits telemetry so you can understand what the server and workers are doing without SSH'ing into hosts. Telemetry includes job lifecycle events, worker heartbeats, logs and issues, and metrics suitable for dashboards and alerting.

What telemetry includes

- **Job lifecycle and health**: staged/deployed/running outcomes, failures, retries, and warnings surfaced in the UI.
- **Worker heartbeats and system status**: online/offline, basic host metrics, and runtime coordination signals.
- **Metrics**: counters and gauges for server and worker behavior (for example throughput, error rates, backpressure).
- **Logs and issues**: actionable warnings/errors you can use to triage failed runs or misconfiguration.

Where to view it

- **UI**:
- **Dashboard** for quick health and issues.
- **Jobs** and **Workers** for run history and per-worker state.
- **Monitoring** and **Metrics Explorer** for stored metrics.
- **Endpoints (advanced)**:
- Worker telemetry surfaces under `/api/workers` (logs and metrics).
- Stored metrics query surfaces under `/api/metrics` (used by the Metrics Explorer UI).

Access and security

Telemetry endpoints follow the same authentication and RBAC rules as the rest of the LyftData API.

- Use a dedicated **read-only** token for dashboards and scrapers.
- If you run Prometheus or log shippers from fixed IPs, the server supports IP allowlisting for selected read endpoints

via `--whitelist` / `LYFTDATA_API_WHITELIST`. This bypass is limited and still requires the `Authorization: WHITELIST` header (see Security hardening and Configuration).

Outbound "phone-home" telemetry

LyftData can optionally send anonymized usage/health telemetry outbound when enabled. If you want to disable outbound telemetry entirely, start the server with `--disable-telemetry` or set `LYFTDATA_DISABLE_TELEMETRY=true`.

Operational tips

- Start with Monitoring and Logs and issues to get a solid “day 2”

operational baseline.

- If live UI status appears stale, check that your environment allows long-lived HTTP connections (reverse proxies,

VPNs, and corporate gateways can interrupt streaming updates).

Worker Auto Enrollment

Enroll new workers with shared secrets instead of manual API keys

Source: operate/worker-auto-enrollment.md

Auto enrollment lets you bootstrap external workers without copying API keys by hand. The server issues an API key the first time a worker connects with the shared secret, then the worker caches its credentials locally. Use this flow to scale fleets quickly, then disable it again once provisioning is complete.

Requirements

- A licensed deployment. Community Edition supports only the built-in worker, so external worker enrollment is unavailable.
- Admin access to the server UI.
- A secure channel for distributing the enrollment secret (VPN, secrets manager, or trusted configuration system). Treat it like a root password.

Configure auto-enrollment on the server

1. Sign in to the server UI as an admin. 2. Navigate to **Settings** → **Security**. 3. Under **Auto-enrollment**, enable it and set an enrollment secret (generate a random value of at least 32 characters). 4. Save. The secret is write-only and will not be shown again.

Bootstrap a worker

1. On the worker host, set the URL, a unique name, and the enrollment secret:

```
   export LYFTDATA_URL=https://lyftdata.example.com
export LYFTDATA_WORKER_NAME=ingest-us-east-01 export LYFTDATA_AUTO_ENROLLMENT_KEY=
<enrollment-secret> export LYFTDATA_JOBS_DIR=/var/lib/lyftdata-worker


lyftdata-worker
```

2. Confirm the worker joined the fleet in the **Workers** UI (or via `lyftdata workers list`). 3. After the first successful enrollment, remove `LYFTDATA_AUTO_ENROLLMENT_KEY` from the worker's service configuration. The worker will reuse its cached identity and API key from `LYFTDATA_JOBS_DIR` on restart.

Rotate or disable enrollment

- **Disable** auto-enrollment in **Settings** → **Security** once provisioning is finished to block unexpected workers.
- **Rotate the secret** if you suspect exposure.

- To force a specific worker to enroll again, start it with a fresh `LYFTDATA_JOBS_DIR` (or delete the existing directory after confirming you no longer need the cached identity), then restart it with the current secret.

Troubleshooting

Symptom	Where to look	Likely cause
Worker exits immediately	Worker logs (`journalctl -u lyftdata-worker`)	Secret missing, worker name unset, or cached ID/API key mismatch
Worker shows auth/enrollment errors	Worker + server logs	Auto-enrollment disabled, secret mismatch, or the worker cannot reach the server URL
External workers don't work	License screen / server logs	Instance is running Community Edition; external workers require a license
Worker never appears in the UI	Worker + server logs	Wrong `LYFTDATA_URL` scheme/host, TLS trust issues, or firewall rules blocking access

Best practices

- Use auto enrollment only on trusted networks and only for the time it takes to bring new workers online. Disable it afterwards.
- Store the secret in a secrets manager and inject it as an environment variable rather than baking it into images.
- Monitor `/api/workers` for unexpected entries and alert when workers appear with unfamiliar names.
- Combine enrollment with automation: Terraform/Ansible can start new nodes and immediately disable auto-enrollment once the fleet is provisioned.

With these safeguards in place, auto enrollment speeds up provisioning without sacrificing traceability or control.