AWFFull in the Cloud: Shipping Logs to Object Storage and Running in Containers

2026-03-12

A practical guide to shipping web server access logs to AWS S3 or Azure Blob Storage using Fluent Bit, Vector, or Filebeat/Logstash — including persistent buffering, retry configuration, concurrent write safety, and log loss prevention. Covers running AWFFull as a stateless container against object storage logs and serving the generated reports as a static website.

This is a companion post to AWFFull 4.0.0 — Modernising a 2008 Web Log Analyser. It covers the operational side: how to get web server access logs into S3 or Azure Blob Storage reliably, and how to run AWFFull inside a container to generate reports from those logs.

Shipping Access Logs to Object Storage

Before AWFFull can process logs from the cloud, something needs to put them there. The most common approach is to run a log shipper alongside your web server that tails the access log and uploads it to S3 or Azure Blob in date-partitioned batches.

AWFFull Log Flow — Client ⟶ Web Server ⟶ Shipper ⟶ Object Storage

Figure 1: AWFFull Log Flow — Client ⟶ Web Server ⟶ Shipper ⟶ Object Storage

> **⚠ Critical requirement for AWFFull compatibility**
>
> The shipper must preserve the raw log lines unchanged — AWFFull needs to see the original Combined/CLF format. Do not parse, transform, or convert the lines to JSON.

Fluent Bit → S3

Fluent Bit is lightweight, written in C, and has native S3 and Azure Blob output plugins.

```
# /etc/fluent-bit/fluent-bit.conf

[INPUT]
    Name            tail
    Path            /var/log/nginx/access.log
    Tag             nginx.access
    Read_from_Head  false
    Refresh_Interval 5
```

```
[OUTPUT]
    Name            s3
    Match           nginx.access
    bucket          my-logs
    region          eu-central-1
    # Date-partitioned path — one directory per day.
    # $UUID is a Fluent Bit built-in variable; it ensures unique
filenames
    # when multiple instances run concurrently (e.g. multiple Pods).
    s3_key_format   /nginx/%Y/%m/%d/access_%H%M%S_$UUID.log.gz
    compression     gzip
    total_file_size 50M          # flush when file reaches 50 MB
    upload_timeout  10m          # or after 10 minutes, whichever comes
first
    use_put_object  On
```

For **Azure Blob** replace the output section:

```
[OUTPUT]
    Name              azure_blob
    Match             nginx.access
    account_name      mystorageaccount
    shared_key        ${AZURE_STORAGE_KEY}
    container_name    nginx-logs
    # The azure_blob plugin does not support $UUID in the path parameter.
    # Uniqueness is guaranteed because Fluent Bit appends an internal
    # timestamp + sequence counter to each blob name automatically.
    path              nginx/%Y/%m/%d
    auto_create_container On
    tls               On
    compress          gzip
```

Vector → S3 / Azure Blob

Vector offers more flexibility for transformations and supports both sinks natively:

```
# vector.yaml

sources:
  nginx_access:
    type: file
    include:
      - /var/log/nginx/access.log
    read_from: end      # only new lines; use 'beginning' for backfill

sinks:
```

```
  s3_logs:
    type: aws_s3
    inputs: [nginx_access]
    bucket: my-logs
    region: eu-central-1
    # Vector automatically appends a UUID to key_prefix — no explicit
$UUID needed.
    # Multiple instances writing concurrently will never collide.
    key_prefix: "nginx/{{ now() | strftime('%Y/%m/%d') }}/"
    compression: gzip
    encoding:
      codec: text       # raw lines — no JSON wrapping
    batch:
      max_bytes: 52428800   # 50 MB
      timeout_secs: 600     # or 10 minutes
```

For Azure Blob, replace the sink:

```
  azure_blob_logs:
    type: azure_blob
    inputs: [nginx_access]
    connection_string: "${AZURE_STORAGE_CONNECTION_STRING}"
    container_name: nginx-logs
    # Same as S3: Vector appends a UUID automatically — uniqueness is
built-in.
    blob_prefix: "nginx/{{ now() | strftime('%Y/%m/%d') }}/"
    encoding:
      codec: text
    compression: gzip
```

Filebeat → Logstash → S3

Filebeat is the standard shipper in the Elastic Stack. It has an nginx module that parses, enriches and structures the access log — but **do not use the nginx module for AWFFull**. The module converts log lines to JSON with split fields; AWFFull needs the raw Combined/CLF format.

Instead, use the generic `filestream` input with no parsing:

```
# /etc/filebeat/filebeat.yml

filebeat.inputs:
  - type: filestream
    id: nginx-access
    paths:
      - /var/log/nginx/access.log
    # No parsers, no module — raw lines only
    parsers: []
```

```
output.logstash:
  hosts: ["logstash:5044"]
```

Filebeat has no native S3 output. Use **Logstash** as the intermediary with the `logstash-output-s3` plugin:

```
# /etc/logstash/conf.d/nginx-s3.conf

input {
  beats { port => 5044 }
}

filter {
  # No grok, no parsing — pass the raw message through unchanged
}

output {
  s3 {
    bucket         => "my-logs"
    region         => "eu-central-1"
    prefix         => "nginx/%{+YYYY}/%{+MM}/%{+dd}/"
    codec          => line          # raw lines, one per file
    encoding       => "gzip"
    size_file      => 52428800      # 50 MB per file
    time_file      => 10            # or rotate after 10 minutes
    # Credentials from IAM role / environment
  }
}
```

The `codec => line` setting is critical — it writes one log line per line to the file, which is exactly what AWFFull expects. The default `json_lines` codec would break compatibility.

Note: Filebeat also supports shipping to Azure via Logstash with the logstash-output-azure_blob_storage community plugin, using the same pattern.

Concurrent Write Safety

When multiple shipper instances write to the same bucket simultaneously (e.g. Kubernetes Pods with horizontal scaling, or multiple servers sending to one central bucket), the first question is: **can this corrupt the data?**

Short answer: no — but data loss is possible if you don't use unique keys.

S3 and Azure Blob both guarantee **atomic PUT semantics** at the object level. A reader will always see either the complete previous version or the complete new version of an object — never a torn or half-written file. Byte-level corruption from concurrent writes is not possible.

The real risk is **silent overwrite**: two instances that generate the same object key will race, and the last writer wins. The first writer's data is permanently lost.

Here is how the tools described above handle this:

Tool	Mechanism	Multiple instances
Fluent Bit → S3	`$UUID` in `s3_key_format`	Safe — each instance generates a unique key
Vector → S3	UUID appended automatically to `key_prefix`	Safe — built-in
Vector → Azure Blob	UUID appended automatically to `blob_prefix`	Safe — built-in
Logstash → S3	`hostname` + `timestamp` in the generated filename	Mostly safe — risk only if two instances share an identical hostname and flush within the same second
Fluent Bit → Azure Blob (append blob)	Append blob type with atomic server-side appends	Safe — Azure Blob Storage serialises concurrent appends to the same blob at the service level
Fluent Bit → Azure Blob (block blob)	Path parameter sets the blob name	Unsafe if multiple instances use the same `path` without a unique suffix

Practical rule: always include a UUID or a unique hostname token in your object key / blob path. All three tools support this out of the box; just avoid overriding their defaults with a static, shared key.

Log Loss at High-Traffic Sites

Concurrent write collisions are not the only way to lose log data. At high request rates — or under network pressure — the shipper itself can become a bottleneck. The common failure modes are:

1. Memory buffer overflow Every shipper buffers incoming lines in memory while waiting to flush to S3/Azure. If the upload is slower than the ingestion rate (network hiccup, S3 throttle, slow endpoint), the buffer fills up. Once full, depending on configuration, the shipper either **drops new lines silently** or applies **backpressure** upstream — which can slow down the web server itself.

2. Network outage or S3/Azure throttling Object storage APIs return `503 Slow Down` (S3) or `429 Too Many Requests` (Azure) under load. Without a persistent buffer and retry logic the shipper will drop records it cannot upload.

3. Shipper crash without offset tracking If the shipper process dies with data still in its memory buffer, those records are gone. On restart it may resume from the wrong position: too early (duplicate records) or too late (gap in coverage).

4. Log rotation race `logrotate` (or nginx's own rotation) can rename or truncate the access log before the shipper has finished reading it. The unread tail of the old file is silently lost.

Each tool has a different answer to these problems:

Tool	Persistence	At-least-once guarantee
Fluent Bit	Filesystem buffer (`storage.type filesystem`)	Yes — survives restarts and network outages if filesystem buffering is enabled
Vector	Disk buffer (`type: disk` in buffer config)	Yes — survives restarts; default is in-memory only
Logstash	Persistent queue (`queue.type: persisted`)	Yes — survives restarts; default is in-memory only
Filebeat	Registry file tracks read offset per file	Yes — resumes exactly where it left off after restart

Filebeat is the most robust for at-least-once delivery: it writes a registry file that records the exact byte offset and inode for every tailed file. After a crash or restart it picks up from that position. The one gap: if the log file is rotated **and deleted** before Filebeat reads to the end, the unread bytes are lost.

Recommendations for high-traffic sites:

- Enable filesystem / disk / persistent buffering in your shipper — never rely solely on in-memory buffers in production
- Set a generous retry policy and back-off interval for S3/Azure upload errors
- Configure log rotation with a `delaycompress` / `rotate_wait` window long enough for the shipper to finish reading the old file before it is compressed or deleted
- Monitor the shipper's own metrics (dropped records, buffer utilisation, upload errors) — these are separate from the web server metrics

Below are complete production-ready configurations with persistent buffering and retry logic enabled for each tool.

Fluent Bit — Filesystem Buffer + Retry

```
# /etc/fluent-bit/fluent-bit.conf

[SERVICE]
    Flush             5
    Daemon            Off
    Log_Level         info

    # Filesystem buffering — chunks are written to disk before upload.
    # Data survives Fluent Bit restarts and network outages up to the
limit.
    storage.path              /var/lib/fluent-bit/buf/
    storage.sync              normal      # fsync after each chunk write
    storage.checksum          off
    storage.max_chunks_up     128         # max chunks in memory at once

[INPUT]
```

```
    Name              tail
    Path              /var/log/nginx/access.log
    Tag               nginx.access
    Read_from_Head    false
    Refresh_Interval  5
    # Offset database — Fluent Bit remembers the last read position.
    # Without this, every restart re-reads the file from the beginning.
    DB                /var/lib/fluent-bit/nginx-access.db
    # Use filesystem storage for this input's chunks (not memory)
    storage.type      filesystem

[OUTPUT]
    Name              s3
    Match             nginx.access
    bucket            my-logs
    region            eu-central-1
    s3_key_format     /nginx/%Y/%m/%d/access_%H%M%S_$UUID.log.gz
    compression       gzip
    total_file_size   50M
    upload_timeout    10m
    use_put_object    On

    # Retry: up to 5 attempts with exponential back-off (max ~160 s
between tries).
    # Set False for unlimited retries (use with caution — can block the
pipeline).
    Retry_Limit       5

    # Cap the on-disk buffer for this output at 1 GB.
    # When full, oldest chunks are dropped to make room.
    storage.total_limit_size  1G
```

The `DB` file (SQLite) is the key to surviving log rotation: Fluent Bit stores the inode + offset, so even if `logrotate` renames `access.log` to `access.log.1`, Fluent Bit will finish reading the old inode before switching to the new file.

Vector — Disk Buffer + Retry

```
# /etc/vector/vector.yaml

sources:
  nginx_access:
    type: file
    include:
      - /var/log/nginx/access.log
```

```
    read_from: end
    # Vector also writes a checkpoint file (similar to Filebeat's
registry)
    # to /var/lib/vector/ by default — survives restarts automatically.

sinks:
  s3_logs:
    type: aws_s3
    inputs: [nginx_access]
    bucket: my-logs
    region: eu-central-1
    key_prefix: "nginx/{{ now() | strftime('%Y/%m/%d') }}/"
    compression: gzip
    encoding:
      codec: text
    batch:
      max_bytes: 52428800   # 50 MB per object
      timeout_secs: 600     # or 10 minutes, whichever comes first

    # Persistent disk buffer — events are written to disk before upload.
    # Survives Vector restarts; default is type: memory (lost on crash).
    buffer:
      type: disk
      max_size: 1073741824  # 1 GB
      when_full: block       # apply backpressure instead of dropping
events

    # Retry policy for S3 upload failures (503, 429, network errors).
    request:
      retry_attempts: 10               # max upload attempts per batch
      retry_initial_backoff_secs: 1    # first retry after 1 s
      retry_max_duration_secs: 300     # cap back-off at 5 minutes
      timeout_secs: 60                 # per-request timeout
      rate_limit_num: 100              # max concurrent requests
```

`when_full: block` is the safe choice for log data — it slows the pipeline rather than silently dropping events. Use `when_full: drop_newest` only if you explicitly accept loss under extreme backlog.

Filebeat + Logstash — Registry + Persistent Queue

Filebeat (`/etc/filebeat/filebeat.yml`):

```
filebeat.inputs:
  - type: filestream
    id: nginx-access
```

```
    paths:
      - /var/log/nginx/access.log
    parsers: []   # no parsing — raw lines only

# Filebeat writes a registry to /var/lib/filebeat/registry/ that records
# the inode, byte offset, and identifier for every tracked file.
# On restart it resumes exactly where it stopped — no duplicates, no
gaps.
filebeat.registry.path: /var/lib/filebeat/registry

# Internal memory queue — tune for throughput vs. latency.
# These events are lost on crash; durability comes from the registry,
# not from this queue.
queue.mem:
  events: 4096
  flush.min_events: 512
  flush.timeout: 5s

output.logstash:
  hosts: ["logstash:5044"]
  # Retry connecting to Logstash — Filebeat will keep trying
indefinitely.
  # Events accumulate in the queue while Logstash is unreachable.
  timeout: 30
  max_retries: 3      # per batch; 0 = retry forever
  bulk_max_size: 2048
```

Logstash (`/etc/logstash/logstash.yml`):

```
# Persistent queue — events are written to disk between the input and
filter/
# output stages. Logstash survives restarts without losing in-flight
events.
queue.type: persisted
queue.max_bytes: 1gb
queue.path: /var/lib/logstash/queue

# Checkpoint frequency — lower values = more durable, more I/O.
queue.checkpoint.acks: 1024
queue.checkpoint.writes: 1024

# Dead-letter queue — events that fail after all retries go here instead
# of being silently dropped. Inspect with the dead_letter_queue input
plugin.
dead_letter_queue.enable: true
```

```
dead_letter_queue.max_bytes: 1gb
path.dead_letter_queue: /var/lib/logstash/dead_letter_queue
```

Logstash pipeline (`/etc/logstash/conf.d/nginx-s3.conf`):

```
input {
  beats { port => 5044 }
}

filter {
  # No grok, no parsing — pass the raw message through unchanged
}

output {
  s3 {
    bucket              => "my-logs"
    region              => "eu-central-1"
    prefix              => "nginx/%{+YYYY}/%{+MM}/%{+dd}/"
    codec               => line          # one raw log line per line
    encoding            => "gzip"
    size_file           => 52428800      # rotate object at 50 MB
    time_file           => 10            # or after 10 minutes
    # Local staging directory — files are assembled here before upload.
    # Use a path on a real disk, not tmpfs, for crash safety.
    temporary_directory => "/var/lib/logstash/s3-tmp"
    # Retry failed uploads before giving up and sending to the DLQ.
    retry_count         => 10
    retry_delay         => 1             # seconds between retries
  }
}
```

With `queue.type: persisted` and the dead-letter queue enabled, Logstash provides end-to-end at-least-once delivery: events survive a Logstash restart, and events that cannot be delivered after all retries land in the DLQ for manual inspection rather than being silently dropped.

Running AWFFull Against Object Storage Logs

Figure 2: AWFFull Report Pipeline — Object Storage ⟶ AWFFull ⟶ Object Storage

i Info

The AWFFull container downloads the log files and the previous `awffull.hist` state file from object storage, runs the analysis, then uploads the generated HTML + PNG reports and the updated history file back — no persistent volumes needed.

Once logs are stored with date-based prefixes, fetching a specific day is a single command:

From S3:

```
# Download all log files for 12 March 2026
aws s3 sync s3://my-logs/nginx/2026/03/12/ /tmp/logs/2026-03-12/

# Process with AWFFull (pass files in chronological order)
awffull -c /etc/awffull.conf $(ls /tmp/logs/2026-03-12/*.log.gz | sort)
```

From Azure Blob:

```
azcopy copy \
    "https://mystorageaccount.blob.core.windows.net/nginx-logs/nginx/
2026/03/12/*" \
    /tmp/logs/2026-03-12/ \
    --recursive

awffull -c /etc/awffull.conf $(ls /tmp/logs/2026-03-12/*.log.gz | sort)
```

Downloading a full month (common for monthly reporting):

```
aws s3 sync s3://my-logs/nginx/2026/03/ /tmp/logs/2026-03/ --include
"*.gz"
awffull -c /etc/awffull.conf $(find /tmp/logs/2026-03/ -name "*.log.gz" |
sort)
```

The `sort` is important — AWFFull processes log records in the order it receives them. Files uploaded with timestamps in their names (e.g. `access_0230_abc123.log.gz`) will sort correctly into chronological order automatically.

Running AWFFull in a Container World

AWFFull works perfectly well inside a container — the challenge is purely about where the log files come from and where the generated HTML goes. In a container-native environment there are several proven patterns:

Option 1 — Persistent Volume (classical approach)

The simplest lift-and-shift. Mount a persistent volume into the container for both the log input and the report output, exactly as you would with a local disk. Works identically to the bare-metal setup.

```
# docker-compose example
services:
  awffull:
    image: registry.gitlab.com/aleks001/awfull:latest
    volumes:
      - /var/log/nginx:/logs:ro          # log files (read-only)
      - /srv/awffull/reports:/reports    # generated HTML + PNG output
```

```
      - /etc/awffull.conf:/etc/awffull.conf:ro
    command: ["-c", "/etc/awffull.conf", "/logs/access.log.gz"]
```

In Kubernetes, replace the host-path mounts with a `PersistentVolumeClaim`. A CronJob running once a night to process the previous day's log rotation works very well here.

```
# Kubernetes CronJob skeleton
apiVersion: batch/v1
kind: CronJob
metadata:
  name: awffull-monthly
spec:
  schedule: "0 2 1 * *"     # 02:00 on the 1st of each month
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: awffull
              image: registry.gitlab.com/aleks001/awfull:latest
              args: ["-c", "/etc/awffull.conf", "/logs/access.log.gz"]
              volumeMounts:
                - name: logs    mountPath: /logs
                - name: reports mountPath: /reports
                - name: config  mountPath: /etc/awffull.conf
                                subPath: awffull.conf
          volumes:
            - name: logs    persistentVolumeClaim: { claimName: nginx-
logs }
            - name: reports persistentVolumeClaim: { claimName: awffull-
reports }
            - name: config  configMap: { name: awffull-config }
          restartPolicy: OnFailure
```

Trade-off: Simple and reliable, but you need to manage the PVC and expose the report directory via a web server separately (e.g. an nginx sidecar or a separate static-files pod).

Option 2 — S3 / S3-compatible Object Storage

This option covers two related questions: 1. Where do the generated reports go? 2. **Can logs that were already shipped to S3 be fed back into AWFFull?**

The answer to both is yes — and combining them gives a fully cloud-native, serverless-friendly pipeline.

Shipping logs to S3 is standard practice. Tools like Fluent Bit, Vector, or even a plain `logrotate` + `aws s3 cp` cron job send compressed log files to a bucket with a path like `s3://my-logs/nginx/2026/03/01/access.log.gz`.

Reading them back for AWFFull is equally straightforward:

```
#!/bin/sh
# 1. Download history file (needed for incremental processing)
aws s3 cp s3://my-bucket/awffull/awffull.hist /var/lib/awffull/ 2>/dev/
null || true

# 2. Download log files for the period to process
aws s3 sync s3://my-logs/nginx/2026/03/ /tmp/logs/ --include "*.gz"

# 3. Run AWFFull (reads logs, updates history, writes HTML)
awffull -c /etc/awffull.conf /tmp/logs/*.gz

# 4. Persist the updated history file back to S3
aws s3 cp /var/lib/awffull/awffull.hist s3://my-bucket/awffull/

# 5. Sync reports to the reports bucket / static website
aws s3 sync /reports/ s3://my-reports-bucket/awffull/ --delete
```

The `awffull.hist` step is important: it is the only piece of persistent state AWFFull needs between runs. Without it, incremental processing (`-p`) won't know what was already counted. Storing it alongside the reports in the same bucket works well.

In Kubernetes, this entire sequence becomes a CronJob that downloads, processes, and re-uploads — no persistent volumes needed at all:

```
S3 (logs)  ──►  AWFFull container  ──►  S3 (reports + awffull.hist)
                      ▲
                      └── IAM role / workload identity
```

Generate the report locally inside the container, then push the entire output directory to an S3 bucket. The bucket can be served directly as a static website (S3 static website hosting, CloudFront, MinIO, etc.).

```
#!/bin/sh
# entrypoint wrapper
awffull -c /etc/awffull.conf /logs/access.log.gz

# Sync output to S3
aws s3 sync /reports/ s3://my-bucket/awffull/ \
    --delete \
    --content-type "text/html" \
    --exclude "*.png" \
    --include "*.html"

aws s3 sync /reports/ s3://my-bucket/awffull/ \
    --exclude "*.html" \
    --include "*.png"
```

In a container, add `awscli` to the image or use a two-stage approach with an init container for awffull and a sidecar for the S3 sync. The AWS credentials come from an IAM role (EKS), a Kubernetes secret, or environment variables.

Trade-off: Zero storage management, instant global CDN distribution, built-in versioning. Requires network access from the runner and appropriate IAM/bucket permissions.

Option 3 — Azure Blob Storage

Same pattern as S3, using the Azure CLI (`az storage blob`) or `azcopy`. The output directory is synced to a container within a Storage Account, which can be configured as a static website endpoint.

Just like S3, logs that were already shipped to Azure Blob Storage (e.g. via Fluent Bit's Azure Blob output, Diagnostic Settings, or a custom pipeline) can be downloaded and fed directly to AWFFull:

```
#!/bin/sh
STORAGE="https://<storage-account>.blob.core.windows.net"

# 1. Download history file
azcopy copy "${STORAGE}/awffull/awffull.hist" /var/lib/awffull/ 2>/dev/
null || true

# 2. Download log files for the period
azcopy copy "${STORAGE}/nginx-logs/2026/03/*" /tmp/logs/ --recursive

# 3. Run AWFFull
awffull -c /etc/awffull.conf /tmp/logs/*.gz

# 4. Persist history back to Blob
azcopy copy /var/lib/awffull/awffull.hist "${STORAGE}/awffull/
awffull.hist"

# 5. Sync reports to static website container ($web)
azcopy sync /reports/ "${STORAGE}/\$web/awffull/" \
    --recursive \
    --delete-destination=true
```

Authentication uses a managed identity (AKS workload identity), a SAS token, or a service principal — no credentials baked into the image:

```
# With workload identity (AKS)
az login --identity
az storage blob sync \
    --source /reports/ \
    --container '$web' \
    --account-name <storage-account> \
    --destination awffull/
```

Enable the **Static website** feature on the Storage Account and the reports are immediately reachable at `https://<storage-account>.z6.web.core.windows.net/awffull/index.html`.

Trade-off: Native Azure integration, no separate web server needed, HTTPS included. Requires AKS workload identity or explicit credentials.

Choosing a Pattern

Scenario	Recommended approach
Simple on-prem Kubernetes	Persistent Volume + nginx sidecar
AWS / EKS	S3 + CloudFront static website
Azure / AKS	Azure Blob static website + workload identity
Self-hosted, no cloud	S3 sync
CI/CD pipeline artifact	Upload as GitLab/GitHub artifact, expire in 90 days

In all cases the AWFFull container itself remains stateless and ephemeral — it runs, generates output, uploads it, and exits. The history file (`awffull.hist`) is the only piece of persistent state needed for incremental processing; store it on the same volume or object storage path.

AWFFull is free software licensed under GPL v3 or later. The project is at https://gitlab.com/aleks001/awffull.