haproxy-spoe-rs: Deployment

2026-04-12

Deploying the haproxy-spoe-rs SPOA agent in production — container image, podman-compose, Kubernetes, HAProxy configuration, health checking, logging, and systemd.

This post covers deploying the haproxy-spoe-rs SPOA agent in production. It is a companion to the architecture post which explains the library design.

The agent and HAProxy run as **separate services** — HAProxy connects to the agent over TCP. This keeps their lifecycles independent and allows scaling each side separately.

Building the container image

The repository includes a two-stage `Containerfile`: the builder stage compiles the `ip_reputation` example with `--release`, the runtime stage copies the binary into a minimal `debian:bookworm-slim` image.

```
podman build -f Containerfile -t spoe-agent:latest .
```

The image runs as any UID — no `USER` directive is set, which makes it compatible with OpenShift's default restricted security context.

Quick smoke test:

```
podman run --rm -e SPOE_ADDR=0.0.0.0:9000 -p 9000:9000 spoe-agent:latest
```

Expected output:

```
spoe agent listening on 0.0.0.0:9000
  TOKIO_WORKER_THREADS : 4
  RUST_LOG             : off
```

podman-compose

The following setup runs HAProxy and the SPOE agent as separate services. HAProxy connects to the agent by service name (`spoe-agent:9000`) over the compose network.

Config files

`haproxy.cfg`

```
global
    log stdout format raw local0

defaults
```

```
    log     global
    mode    http
    option  httplog
    timeout connect   5s
    timeout client   30s
    timeout server   30s

frontend http-in
    bind *:80
    filter spoe engine rate-limit config /etc/haproxy/rate-limit.conf
    http-request deny deny_status 429 if { var(sess.rl.ip_score) -m int
ge 80 }
    default_backend web

backend web
    server s1 web:80

backend spoe-backend
    mode tcp
    server spoa spoe-agent:9000 check inter 5s

listen stats
    bind *:8404
    stats enable
    stats uri /stats
    stats refresh 10s
```

rate-limit.conf

```
[rate-limit]
spoe-agent rate-limit
    messages         check-rate
    option           var-prefix rl
    option           pipelining
    timeout hello    100ms
    timeout idle     30s
    timeout processing 15ms
    use-backend      spoe-backend

spoe-message check-rate
    args ip_fwd=req.hdr(Forwarded),rfc7239_field(for),rfc7239_n2nn
    args ip_xff=req.hdr_ip(X-Forwarded-For,1)
    args ip_src=src
    event on-frontend-http-request
```

SPOE filter ordering: HAProxy runs SPOE filters *before* `http-request` rules — this is hard-coded in the stream processing loop and cannot be changed by moving the `filter` directive in the config. Variables set with `http-request set-var` are **not yet available** when the SPOE message is assembled; they arrive as `Null` in the agent.

Source proof (HAProxy 3.3.6): The `FLT_ANALYZE` macro in `src/stream.c` always calls `flt_pre_analyze()` before the analyzer function. SPOE registers on `AN_REQ_HTTP_PROCESS_FE` as a pre-analyzer (`src/flt_spoe.c:1353`); `http-request` rules run inside `http_process_req_common` which executes after all pre-analyzers (`src/http_ana.c:414`).

Solution: use sample fetches directly in `args` — they are evaluated at filter fire time. The three args above pass RFC 7239, XFF, and TCP src as separate values; the agent resolves the priority chain (Forwarded → XFF → src) via `find_map`.

compose file

```yaml
services:
  haproxy:
    image: haproxytech/haproxy-ubuntu:3.3
    ports:
      - "80:80"
      - "8404:8404"
    volumes:
      - ./haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro
      - ./rate-limit.conf:/etc/haproxy/rate-limit.conf:ro
    depends_on:
      - spoe-agent

  spoe-agent:
    image: spoe-agent:latest
    environment:
      SPOE_ADDR: "0.0.0.0:9000"
      TOKIO_WORKER_THREADS: "4"
      RUST_LOG: "warn"
```

Start:

```
podman-compose up -d
```

Kubernetes

The agent runs as a `Deployment` with its own `Service`. HAProxy resolves the agent via the cluster DNS name `spoe-agent:9000`. The HAProxy config and SPOE config are provided via `ConfigMap`.

SPOE agent

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spoe-agent
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spoe-agent
  template:
    metadata:
      labels:
        app: spoe-agent
    spec:
      containers:
        - name: spoe-agent
          image: spoe-agent:latest
          ports:
            - containerPort: 9000
          env:
            - name: SPOE_ADDR
              value: "0.0.0.0:9000"
            - name: RUST_LOG
              value: "warn"
            - name: TOKIO_WORKER_THREADS
              value: "4"
          resources:
            requests:
              cpu: "250m"
              memory: "64Mi"
            limits:
              cpu: "1"
              memory: "128Mi"
---
apiVersion: v1
kind: Service
metadata:
  name: spoe-agent
spec:
  selector:
    app: spoe-agent
  ports:
    - port: 9000
      targetPort: 9000
```

The container runs as any UID and does not require elevated privileges — it is compatible with the Kubernetes `restricted` Pod Security Standard and OpenShift's default SCC without any additional configuration.

`TOKIO_WORKER_THREADS` should match the CPU limit to avoid creating more threads than the scheduler will run. With a limit of `1`, set `TOKIO_WORKER_THREADS=1`; with `2`, set it to `2`.

HAProxy

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: haproxy-config
data:
  haproxy.cfg: |
    global
        log stdout format raw local0
    defaults
        log     global
        mode    http
        option  httplog
        timeout connect   5s
        timeout client   30s
        timeout server   30s
    frontend http-in
        bind *:80
        filter spoe engine rate-limit config /etc/haproxy/rate-limit.conf
        http-request deny deny_status 429 if { var(sess.rl.ip_score) -m
int ge 80 }
        default_backend web
    backend web
        server s1 web:80
    backend spoe-backend
        mode tcp
        server spoa spoe-agent:9000 check inter 5s
  rate-limit.conf: |
    [rate-limit]
    spoe-agent rate-limit
        messages         check-rate
        option           var-prefix rl
        option           pipelining
        timeout hello    100ms
        timeout idle     30s
        timeout processing 15ms
        use-backend      spoe-backend
    spoe-message check-rate
        args ip_fwd=req.hdr(Forwarded),rfc7239_field(for),rfc7239_n2nn
```

```
        args ip_xff=req.hdr_ip(X-Forwarded-For,1)
        args ip_src=src
        event on-frontend-http-request
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: haproxy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: haproxy
  template:
    metadata:
      labels:
        app: haproxy
    spec:
      containers:
        - name: haproxy
          image: haproxytech/haproxy-ubuntu:3.3
          ports:
            - containerPort: 80
            - containerPort: 8404
          volumeMounts:
            - name: config
              mountPath: /usr/local/etc/haproxy/haproxy.cfg
              subPath: haproxy.cfg
            - name: config
              mountPath: /etc/haproxy/rate-limit.conf
              subPath: rate-limit.conf
      volumes:
        - name: config
          configMap:
            name: haproxy-config
---
apiVersion: v1
kind: Service
metadata:
  name: haproxy
spec:
  selector:
    app: haproxy
  ports:
    - name: http
      port: 80
```

```
      targetPort: 80
    - name: stats
      port: 8404
      targetPort: 8404
```

HAProxy backend configuration

A few directives on the `spoe-backend` that matter in production:

```
backend spoe-backend
    mode tcp
    server spoa spoe-agent:9000 check inter 5s   # TCP health check every
5 s
    server spoa2 spoe-agent2:9000 check inter 5s # second instance for
redundancy
```

Directive	Where	Effect
`check inter 5s`	`server`	TCP health check interval — removes failed agents automatically
`maxconn N`	`server`	Max simultaneous connections per agent instance
`timeout processing`	SPOE config	Hard deadline per NOTIFY/ACK — tune to your handler latency
`timeout idle`	SPOE config	Keepalive for idle SPOE connections
`option pipelining`	SPOE config	Required for high throughput — sends multiple NOTIFYs without waiting for ACKs

Health checking

The SPOE protocol has a built-in healthcheck: when HAProxy opens a connection for checking, it sends `HAPROXY-HELLO` with `healthcheck=true`. The agent replies with `AGENT-HELLO` and closes the connection immediately. This is handled by the library — no code needed in the handler.

The `check inter 5s` on the `server` line triggers a TCP connection which exercises this path. If the agent does not respond, HAProxy marks the server down and stops sending NOTIFYs to it.

Logging and observability

The agent uses the `log` facade. Enable it by setting `RUST_LOG`:

Value	Effect
`off` (default)	No output
`warn`	Connection errors and non-zero DISCONNECT status
`haproxy_spoe=debug`	Protocol-level events (not recommended in production)

HAProxy's own logging covers request-level events — whether a stream was denied based on the `ip_score` variable set by the agent is visible in the HAProxy access log.

Distributed GCRA rate limiting with Valkey

The basic example above assigns a static score per IP. In production you often need **actual rate limiting** — count real requests, allow controlled bursts, and keep state consistent across all agent replicas.

GCRA (Generic Cell Rate Algorithm) stores a *theoretical arrival time* (TAT) per client and updates it atomically on every request. When the agent runs as multiple replicas, that state must live in a shared external store.

Valkey (Linux Foundation Redis fork, BSD-3-Clause) fills that role. Valkey Cluster distributes hash slots across nodes — active-active — and AOF persistence ensures state survives pod restarts.

> A fully deployable example — standalone Rust crate, Kustomize manifests for Kubernetes and OpenShift, and ArgoCD `Application` objects — lives in `examples/GCRA-Rate-Limiting/` in the repository.

Topology

Component	Replicas	State	Persistence
HAProxy	3	Stateless	—
spoe-agent	3	Stateless	—
Valkey Cluster	3	Stateful (TAT per IP)	AOF (`everysec`)

All three tiers are stateless except Valkey. A given client IP's TAT key maps to exactly one Valkey primary — GCRA correctness is guaranteed regardless of which agent or HAProxy instance handles the request.

{{ bounded_image(src="/img/haproxy-gcra-rate-limit-topology.drawio.png", alt="GCRA distributed rate limiting topology: Client ⟶ HAProxy ×3 (stateless) ⟶ spoe-agent ×3 (stateless) ⟶ Valkey Cluster ×3 (stateful, AOF). ALLOW forwards to backend; DENY returns 429.", max_width=800) }}

The request flows left to right. HAProxy holds the HTTP stream and sends a SPOE NOTIFY with three IP args (`ip_fwd`, `ip_xff`, `ip_src`) — evaluated directly at filter fire time. The agent resolves the priority chain (RFC 7239 Forwarded ⟶ X-Forwarded-For ⟶ TCP src) and runs the GCRA Lua script against Valkey. Valkey returns `[1, 0]` (allow) or `[0, retry_after_µs]` (deny); the agent writes `sess.rl.ip_score = 0` or `100` into the ACK. HAProxy resumes the stream: `ip_score < 80` forwards to the backend, `ip_score ≥ 80` returns 429.

Only Valkey carries state — HAProxy and the spoe-agent are fully stateless and scale horizontally without coordination. A given IP's GCRA key always maps to the same Valkey primary via hash-slot routing, so GCRA correctness is guaranteed across all replicas.

Valkey Cluster

`valkey-cluster.yaml`

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: valkey-config
data:
  valkey.conf: |
    cluster-enabled yes
    cluster-config-file /data/nodes.conf
    cluster-node-timeout 5000
    appendonly yes
    appendfsync everysec
    protected-mode no
    bind 0.0.0.0
---
apiVersion: v1
kind: Service
metadata:
  name: valkey-headless
spec:
  clusterIP: None
  selector:
    app: valkey
  ports:
    - name: client
      port: 6379
    - name: gossip
      port: 16379
---
apiVersion: v1
kind: Service
metadata:
  name: valkey
spec:
  selector:
    app: valkey
  ports:
    - port: 6379
      targetPort: 6379
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: valkey
spec:
  serviceName: valkey-headless
  replicas: 3
```

```
  selector:
    matchLabels:
      app: valkey
  template:
    metadata:
      labels:
        app: valkey
    spec:
      containers:
        - name: valkey
          image: valkey/valkey:8.1
          command:
            - valkey-server
            - /etc/valkey/valkey.conf
            - --cluster-announce-hostname
            - $(POD_NAME).valkey-headless.$(NAMESPACE).svc.cluster.local
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          ports:
            - containerPort: 6379
            - containerPort: 16379
          volumeMounts:
            - name: config
              mountPath: /etc/valkey
            - name: data
              mountPath: /data
          resources:
            requests:
              cpu: "250m"
              memory: "128Mi"
            limits:
              cpu: "1"
              memory: "256Mi"
      volumes:
        - name: config
          configMap:
            name: valkey-config
  volumeClaimTemplates:
    - metadata:
```

```
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

After all three pods are `Running`, initialize the cluster once:

```
kubectl exec valkey-0 -- valkey-cli --cluster create \
  valkey-0.valkey-headless:6379 \
  valkey-1.valkey-headless:6379 \
  valkey-2.valkey-headless:6379 \
  --cluster-replicas 0 --cluster-yes
```

> **OpenShift note:** `valkey/valkey:8.1` runs as UID 999. OpenShift's restricted SCC assigns a random UID from the namespace range. Add `securityContext: { runAsUser: null }` to the container spec to let OpenShift assign the UID.

GCRA handler

Add the `redis` and `r2d2` crates to `Cargo.toml`:

```
redis = { version = "0.27", features = ["cluster"] }
r2d2  = "0.8"
```

`r2d2_redis` only wraps the single-node client, so we implement the pool manager for `ClusterClient` ourselves — it is a small struct.

The GCRA algorithm runs as a Lua script inside Valkey rather than in Rust for one fundamental reason: **atomicity**. A read-modify-write sequence in Rust — GET the TAT, compute the new TAT, SET it back — would require a `WATCH`/`MULTI`/`EXEC` transaction or a Redlock-style distributed lock to be safe under concurrent access from multiple agent replicas. Both approaches add round-trips and complexity.

A Lua script uploaded to Valkey executes as a single atomic unit: no other client can observe or modify the key between the GET and the SET. This is a Redis/Valkey guarantee: scripts run under the single-threaded command processor, so the entire GCRA decision is serialized per key without any locking overhead on the client side.

Performance is not a concern. Valkey (like Redis) embeds LuaJIT — the script is compiled to native code on first load and cached by its SHA1 digest. Subsequent calls via `redis::Script` skip the compilation step entirely. The script itself is five arithmetic operations and two conditional `SET`/`PEXPIRE` calls — it completes in microseconds, well within the `timeout processing 15ms` budget configured in the SPOE agent section.

The script executes atomically, so no two agent instances can race on the same key:

```
use haproxy_spoe::{Agent, Scope, TypedData};
use redis::cluster::{ClusterClient, ClusterConnection};
use std::sync::Arc;
use std::time::{SystemTime, UNIX_EPOCH};
use tokio::net::TcpListener;

// r2d2 connection pool manager for Valkey Cluster.
struct ClusterManager(ClusterClient);

impl r2d2::ManageConnection for ClusterManager {
    type Connection = ClusterConnection;
    type Error = redis::RedisError;

    fn connect(&self) -> Result<Self::Connection, Self::Error> {
        self.0.get_connection()
    }

    fn is_valid(&self, conn: &mut Self::Connection) -> Result<(),
Self::Error> {
        redis::cmd("PING").query(conn)
    }

    fn has_broken(&self, _: &mut Self::Connection) -> bool {
        false
    }
}

// GCRA via atomic Lua script.
// KEYS[1] = TAT key  ARGV[1] = now (µs)  ARGV[2] = emission_interval
(µs)
//           ARGV[3] = burst_tolerance (µs)  ARGV[4] = ttl (ms)
// Returns [1, 0] = allowed, [0, retry_after_µs] = denied.
const GCRA_SCRIPT: &str = r#"
local tat = tonumber(redis.call('GET', KEYS[1]))
local now = tonumber(ARGV[1])
local ei  = tonumber(ARGV[2])
local bt  = tonumber(ARGV[3])
local ttl = tonumber(ARGV[4])
if tat == nil or tat < now then tat = now end
local new_tat = tat + ei
if new_tat - now > bt then
    -- Refresh TTL on denial so the key does not expire while the client
    -- is actively being rate-limited (prevents a free burst after
expiry).
    redis.call('PEXPIRE', KEYS[1], ttl)
    return {0, new_tat - now}
```

```
end
redis.call('SET', KEYS[1], new_tat, 'PX', ttl)
return {1, 0}
"#;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    env_logger::Builder::from_default_env()
        .target(env_logger::Target::Stdout)
        .init();

    let nodes: Vec<String> = std::env::var("VALKEY_NODES")
        .unwrap_or_else(|_| "redis://valkey-0.valkey-headless:
6379".into())
        .split(',')
        .map(|s| s.trim().to_string())
        .collect();

    let pool_size = std::env::var("TOKIO_WORKER_THREADS")
        .ok()
        .and_then(|v| v.parse::<u32>().ok())
        .unwrap_or(16);

    // build_unchecked: no connections opened at startup — first attempt
happens
    // on the first incoming request. Avoids noisy r2d2 ERROR logs when
Valkey
    // is not yet reachable during pod startup.
    let pool = Arc::new(
        r2d2::Pool::builder()
            .max_size(pool_size)
            .min_idle(Some(0))
            .build_unchecked(ClusterManager(ClusterClient::new(nodes)?)),
    );

    let emission_interval_us: i64 = std::env::var("SPOA_RATE_INTERVAL")
        .ok()
        .and_then(|v| v.parse::<i64>().ok())
        .unwrap_or(10_000); // 10 ms = 1 / 100 req/s

    let burst_tolerance_us: i64 = std::env::var("SPOA_RATE_BURST")
        .ok()
        .and_then(|v| v.parse::<i64>().ok())
        .unwrap_or(200_000); // 200 ms = 20 burst requests

    let ttl_ms: i64 = std::env::var("SPOA_RATE_TTL")
```

```
        .ok()
        .and_then(|v| v.parse::<i64>().ok())
        .unwrap_or(1800) * 1000;

    let addr = std::env::var("SPOE_ADDR").unwrap_or_else(|_|
"0.0.0.0:9000".into());
    let listener = TcpListener::bind(&addr).await?;

    let workers = std::env::var("TOKIO_WORKER_THREADS")
        .ok()
        .and_then(|v| v.parse::<usize>().ok())
        .unwrap_or_else(|| std::thread::available_parallelism().map(|n|
n.get()).unwrap_or(1));
    println!("gcra-rate-limit listening on {addr}");
    println!("  TOKIO_WORKER_THREADS  : {workers}");
    println!("  SPOA_RATE_TTL         : {}s", ttl_ms / 1000);
    println!("  SPOA_RATE_INTERVAL    : {}µs (= {:.0} req/s)",
emission_interval_us, 1_000_000.0 / emission_interval_us as f64);
    println!("  SPOA_RATE_BURST       : {}µs (= {} req burst)",
burst_tolerance_us, burst_tolerance_us / emission_interval_us);

    let agent = Agent::new(move |req| {
        let Some(msg) = req.get_message("check-rate") else { return };
        // Priority: RFC 7239 Forwarded > X-Forwarded-For > TCP src.
        // SPOE filters run before http-request rules, so all three
sources
        // are passed as separate args and resolved here.
        let ip = ["ip_fwd", "ip_xff", "ip_src"].iter().find_map(|arg| {
            match msg.get(arg) {
                Some(TypedData::IPv4(ip)) => Some(ip.to_string()),
                Some(TypedData::IPv6(ip)) => Some(ip.to_string()),
                _ => None,
            }
        });
        let Some(ip) = ip else { return };
        let sid = req.stream_id;

        let key = format!("gcra:{ip}");
        let now = SystemTime::now()
            .duration_since(UNIX_EPOCH)
            .unwrap()
            .as_micros() as i64;

        let mut conn = match pool.get() {
            Ok(c) => c,
            Err(e) => {
```

```
                log::warn!("valkey pool unavailable, failing open: {e}");
                return;
            }
        };
        let result: Option<Vec<i64>> = redis::Script::new(GCRA_SCRIPT)
            .key(&key)
            .arg(now)
            .arg(emission_interval_us)
            .arg(burst_tolerance_us)
            .arg(ttl_ms)
            .invoke(&mut *conn)
            .ok();

        let denied = result.as_ref().map(|v| v[0] == 0).unwrap_or(false);

        if denied {
            let retry_us = result.as_ref().map(|v| v[1]).unwrap_or(0);
            log::debug!("stream={sid} DENY  {ip}  retry_after={:.1}ms",
retry_us as f64 / 1000.0);
        } else {
            log::debug!("stream={sid} ALLOW {ip}");
        }

        req.set_var(
            Scope::Session,
            "ip_score",
            TypedData::Int32(if denied { 100 } else { 0 }),
        );
    });

    let mut sigterm =
tokio::signal::unix::signal(tokio::signal::unix::SignalKind::terminate())
        .expect("failed to register SIGTERM handler");

    tokio::select! {
        res = agent.serve(listener) => res?,
        _ = tokio::signal::ctrl_c() => {}
        _ = sigterm.recv() => {}
    }

    Ok(())
}
```

Rate parameters — all configurable via environment variables:

Env var	Default	Meaning
`SPOA_RATE_INTERVAL`	`10000` µs	Emission interval: `1 000 000 / rate_per_second`
`SPOA_RATE_BURST`	`200000` µs	Burst tolerance: `burst_size × SPOA_RATE_INTERVAL`
`SPOA_RATE_TTL`	`1800` s	How long an IP is remembered in Valkey after last request

Example: 50 req/s sustained, burst of 10:

```
SPOA_RATE_INTERVAL=20000   # 1 000 000 / 50 = 20 ms
SPOA_RATE_BURST=200000     # 10 × 20 ms = 200 ms
```

`pool_size` is read from `TOKIO_WORKER_THREADS` so each worker thread can hold its own connection without waiting. Each `pool.get()` call checks out a connection for the duration of the handler invocation and returns it automatically when it goes out of scope.

SPOA Deployment (3 replicas)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spoe-agent
spec:
  replicas: 3
  selector:
    matchLabels:
      app: spoe-agent
  template:
    metadata:
      labels:
        app: spoe-agent
    spec:
      containers:
        - name: spoe-agent
          image: spoe-agent:latest
          ports:
            - containerPort: 9000
          env:
            - name: SPOE_ADDR
              value: "0.0.0.0:9000"
            - name: RUST_LOG
              value: "warn"
            - name: TOKIO_WORKER_THREADS
              value: "2"
            - name: VALKEY_NODES
              value: "redis://valkey-0.valkey-headless:6379,redis://
```

```
valkey-1.valkey-headless:6379,redis://valkey-2.valkey-headless:6379"
            - name: SPOA_RATE_TTL
              value: "1800"
            - name: SPOA_RATE_INTERVAL
              value: "10000"
            - name: SPOA_RATE_BURST
              value: "200000"
          resources:
            requests:
              cpu: "250m"
              memory: "64Mi"
            limits:
              cpu: "1"
              memory: "128Mi"
```

HAProxy load-balances the three agents via the `spoe-agent` ClusterIP Service. No sticky routing is needed.

HAProxy (3 replicas)

Set `replicas: 3` in the HAProxy Deployment from the Kubernetes section. No other changes are needed — each HAProxy instance connects to the same `spoe-agent` Service.

GitOps deployment with ArgoCD

The `examples/GCRA-Rate-Limiting/` directory contains everything needed to deploy the full stack with ArgoCD:

```
examples/GCRA-Rate-Limiting/
├── Cargo.toml                          standalone crate (path dep on
library)
├── Containerfile                       two-stage build, build context =
repo root
├── src/main.rs                         GCRA handler with r2d2, IPv4+IPv6,
signals
├── kubernetes/
│   ├── base/                           namespace-agnostic manifests +
kustomization.yaml
│   ├── overlays/
│   │   ├── kubernetes/                 sets namespace, image tag
│   │   └── openshift/                  + Route + runAsUser: null patch
for HAProxy
│   └── valkey/values.yaml              Helm values for valkey-io/valkey-
helm
└── argocd/
    ├── application-spoe.yaml           Kustomize app (switch overlay via
comment)
```

```
    └── application-valkey.yaml        multi-source: Helm chart + values
from this repo
```

The Valkey `Application` uses ArgoCD's multi-source pattern: the Helm chart comes from the `valkey-io/valkey-helm` repository while `values.yaml` is read from this repository, pinned to `main`:

```
sources:
  - repoURL: https://valkey-io.github.io/valkey-helm
    chart: valkey
    targetRevision: "1.*"
    helm:
      valueFiles:
        - $values/examples/GCRA-Rate-Limiting/kubernetes/valkey/
values.yaml
  - repoURL: https://github.com/git001/haproxy-spoe-rs.git
    targetRevision: main
    ref: values
```

To deploy on vanilla Kubernetes, apply both `Application` objects:

```
kubectl apply -f examples/GCRA-Rate-Limiting/argocd/
```

For OpenShift, edit `application-spoe.yaml` and switch the path to `overlays/openshift` before applying.