강동현 · Infrastructure / Platform Engineer

장애를 직접 재현하고, 무엇이 끊기고 무엇이 남는지 숫자로 확인했습니다

> K3s home lab에서 요청 경로와 상태가 남는 위치를 먼저 찾고, DB 지연·rolling restart·인증서 갱신·동시 refresh 같은 조건을 직접 만들어 관측했습니다. 아래 숫자는 모두 그 실험에서 측정한 값입니다.

66 → 1,872 ms
DB에 200 ms 지연을 넣었을 때 login 응답

151 → 151
rolling restart 전후 DB session row

2,305 s → 1–2 s
인증서 갱신이 실제 serving에 반영되는 시간

K3s · Nginx · Traefik · Keycloak · PostgreSQL · Redis · Vault · Prometheus · Grafana · Loki · Tempo

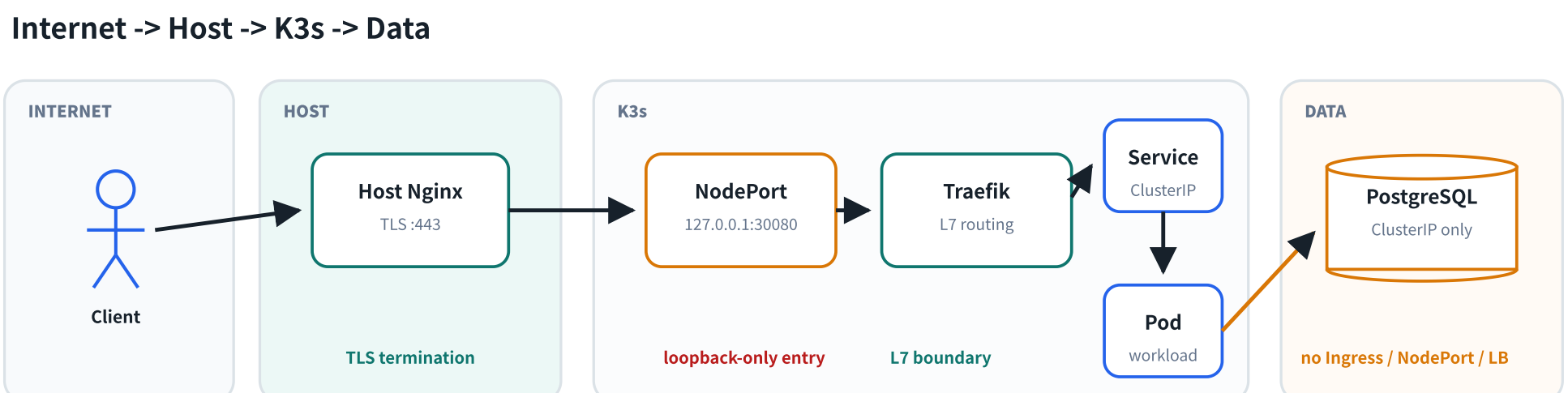
GitHub	**github.com/donghyeon-ka**
TechLog	hyeonworks.com
Email	donghyeonka68@gmail.com

01 · Request Path

외부에서 cluster로 들어오는 문을 Host Nginx 하나로 줄였습니다

요청이 실제로 어느 순서로 들어오는가? · 개인 home lab / dev · Host Nginx · K3s · Traefik · CloudNativePG

TLS는 Host Nginx :443에서 끝나고, 그 뒤 hop은 모두 loopback 안에서만 일어납니다. Traefik NodePort를 `127.0.0.1`에만 bind해 LAN에서 NodePort로 직접 들어오는 경로를 없앴고, PostgreSQL Service에는 Ingress / NodePort / LoadBalancer를 두지 않았습니다.



들어오는 문이 하나면 인증서, forwarded header, 차단 규칙을 한 곳에서만 맞추면 됩니다. 대신 그 한 곳이 단일 실패 지점이므로, 아래 Edge Trust에서 무엇까지 Host Nginx가 책임지는지 정리했습니다.

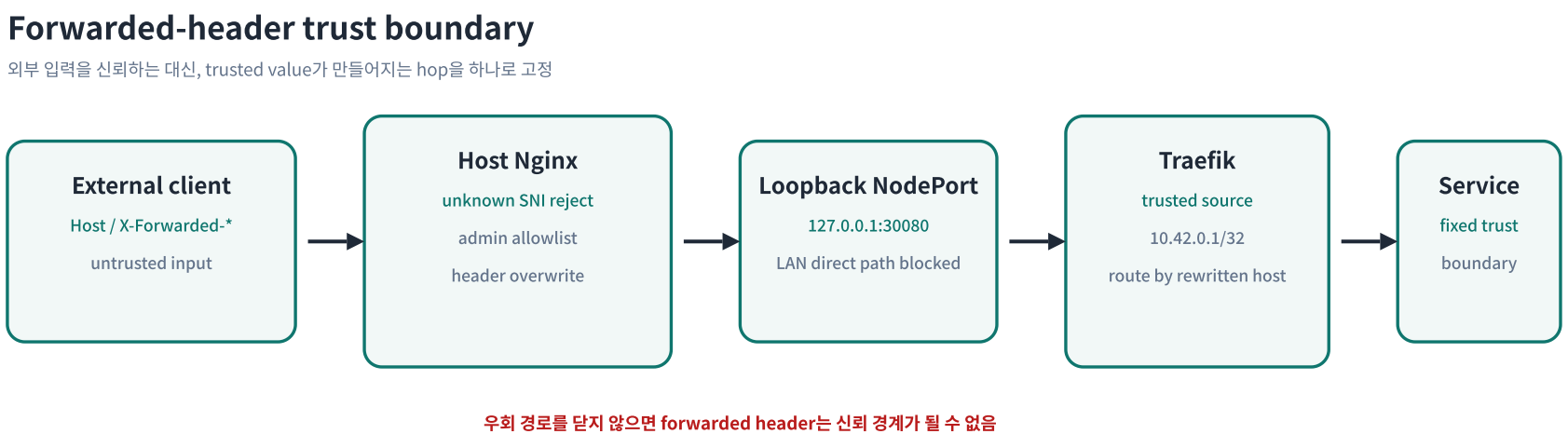
Scope — 단일 물리 서버 home lab입니다. 1/1 Ready나 Local PV Retain을 HA나 backup으로 표현하지 않습니다.

02 · Edge Trust

forwarded header를 믿기 전에 누가 만들 수 있는지부터 막았습니다

클라이언트가 보낸 값과 edge가 확정한 값을 어디서 구분하는가? · Host Nginx · Traefik

외부 요청이 보낸 `X-Forwarded-*`는 신뢰하지 않고 Host Nginx가 다시 만듭니다. Traefik은 그 Host Nginx만 trusted proxy로 인정하므로, 내부에서 보이는 hostname과 client IP는 edge가 확정한 값입니다.



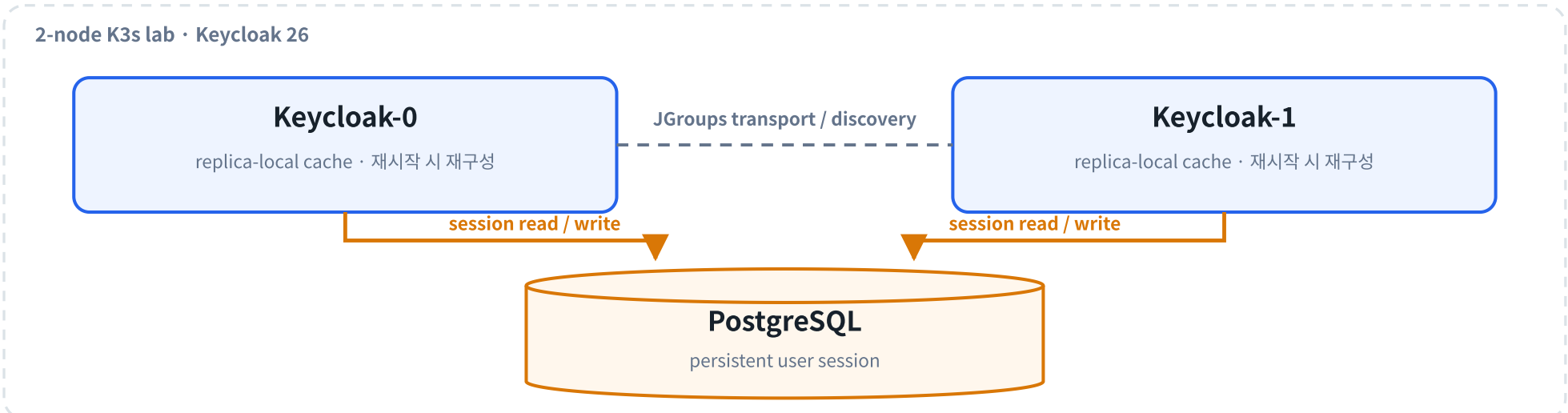
- Host Nginx upstream: `127.0.0.1:30080`
- K3s NodePort bind: `127.0.0.0/8`
- Traefik trusted proxy: `10.42.0.1/32`
- 관리 hostname: LAN `192.168.0.0/24` + Tailscale `100.64.0.0/10` allowlist
- Gitea / Grafana / metrics의 외부 우회 경로는 404

03 · Keycloak · Session State

Persistent user session은 PostgreSQL에, cache는 각 replica에 남습니다

replica를 늘리면 세션은 정말 유지되는가? · VM 2대 · 2-node K3s · Keycloak 26 · JGroups

`persistent-user-sessions`를 켜면 user session의 durable state는 PostgreSQL에 남고, 각 replica는 그와 별개로 local cache를 가집니다. JGroups는 node discovery와 cluster transport를 담당할 뿐 durable state를 소유하지 않습니다.



replica가 죽어도 PostgreSQL의 session row는 남고, local cache만 다시 만들어집니다

replica가 2개라는 사실만으로 세션이 유지되는 것이 아니라, 유지되는 이유가 외부 PostgreSQL에 있습니다. 다음 페이지에서 실제로 replica를 재시작해 어느 쪽이 남는지 확인했습니다.

04 · Keycloak · Rolling Restart

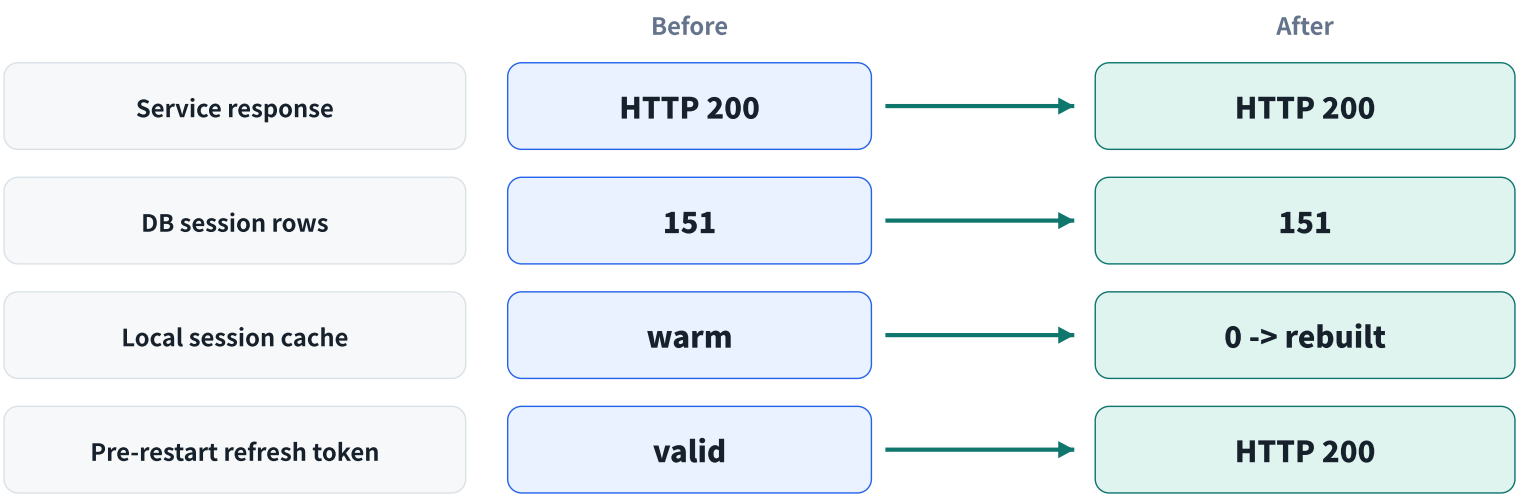
restart 중에도 DB session은 151 → 151로 그대로였습니다

재시작하면 무엇이 끊기고 무엇이 남는가? · rolling restart · 2-node K3s

재시작하는 동안 service 응답은 계속 HTTP 200이었고, PostgreSQL의 session row는 **151 → 151**로 유지됐습니다. 재시작한 replica의 local session cache만 0에서 다시 만들어졌고, 재시작 전에 발급한 refresh token도 이후 요청에서 HTTP 200을 받았습니다.

Rolling restart 전후 state 변화

service continuity / persistent session / local cache를 각각 따로 측정



Scope — 한 물리 서버 안의 2 VM lab 결과입니다. 물리 장애를 포함한 production HA 결과가 아닙니다.

05 · Keycloak · Latency Fault

DB에 넣은 200 ms가 login 1,872 ms와 readiness 실패까지 커졌습니다

한 곳의 지연은 어디까지 번지는가? · PostgreSQL 앞단에 200 ms 지연 주입 · 동시 20건

200 ms는 login 한 번에 여러 번 일어나는 DB round trip마다 붙었고, 느려진 요청이 connection을 오래 쥐면서 뒤 요청의 pool wait까지 끌어올렸습니다. 마지막에는 readiness probe가 timeout되어 느린 Pod가 load-balancing 대상에서 빠졌습니다.

200 ms DB delay propagation

66 -> 1,872 ms	22.2 s	20 s	timeout
login response	20 concurrent max	connection wait max	readiness



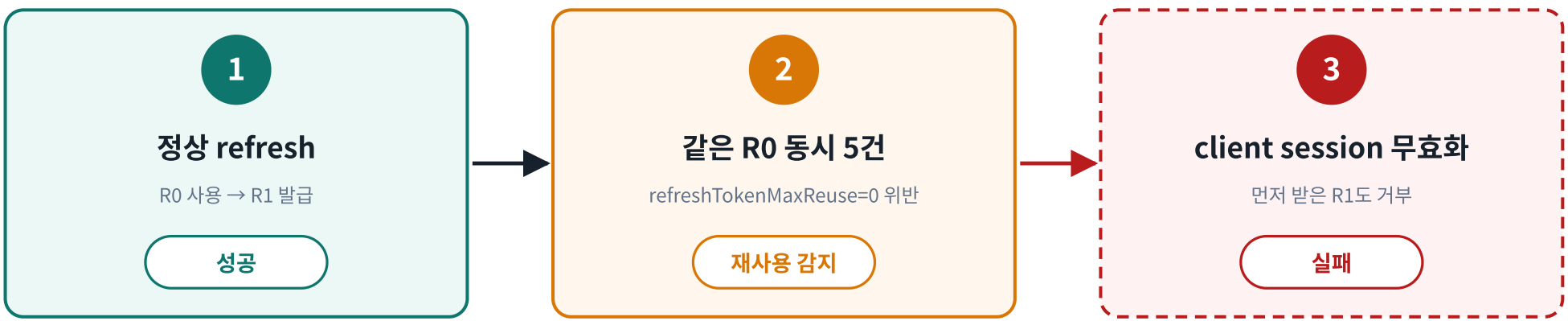
주입한 지연 자체보다, 그 지연이 connection pool과 readiness를 거치며 한 자릿수 배가 아니라 20배 이상으로 커진 점이 이 실험의 결과입니다.

06 · Keycloak · Refresh Rotation

같은 refresh token을 동시에 쓰자 client session이 끊겼습니다

rotation 중에 요청이 겹치면 어느 쪽이 실패하는가? · `revokeRefreshToken=true` · `refreshTokenMaxReuse=0` · 동일 token 5건 동시 요청

같은 refresh token으로 5건을 동시에 보냈습니다. 재사용이 감지된 뒤 client session이 무효화됐고, 그중 먼저 성공 응답으로 받아 둔 **새 refresh token도 이후에는 쓸 수 없었습니다.**



순차 요청에서는 재현되지 않았고, 동시 재사용일 때만 client session이 끊겼습니다

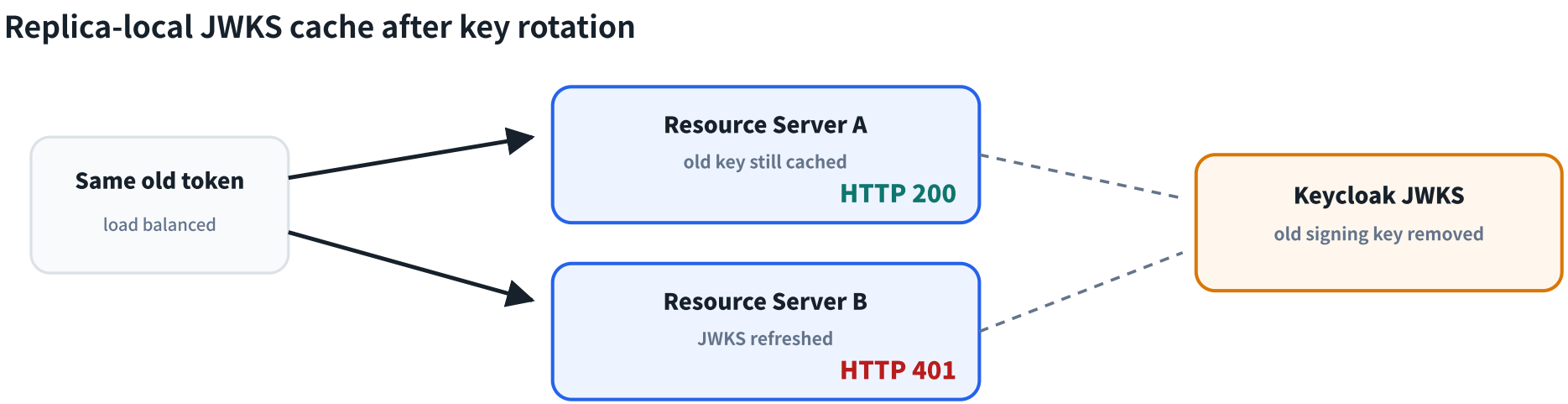
순차로 보냈을 때는 같은 현상이 나오지 않았습니다. 클라이언트 입장에서는 "refresh 한 번 실패"와 "세션 자체가 끊김"이 다른 상황이므로, error handling에서 이 둘을 구분해 재로그인으로 넘길지 판단해야 합니다.

07 · Keycloak · JWKS Cache

같은 token이 replica에 따라 200과 401로 갈렸습니다

서명 키를 지우면 각 replica는 언제 그것을 아는가? · Resource Server 2 replica · Nimbus JWKS cache

realm에서 old signing key를 지운 직후 같은 old access token으로 반복 요청했더니 응답이 **401 → 200 → 401 → 200 → 401 → 200 → 401 → 200**으로 번갈아 나왔습니다.



검증은 Keycloak이 매 요청마다 해 주는 것이 아니라 각 Resource Server replica가 자기 JWKS cache로 합니다. 두 replica의 cache 갱신 시점이 달라서, 한쪽은 아직 old key를 들고 200을, 다른 쪽은 새 JWKS를 반영해 401을 반환했습니다. key를 지운 시점만으로는 이 교차를 설명할 수 없었고, replica별 cache 갱신 시점까지 봐야 설명됐습니다.

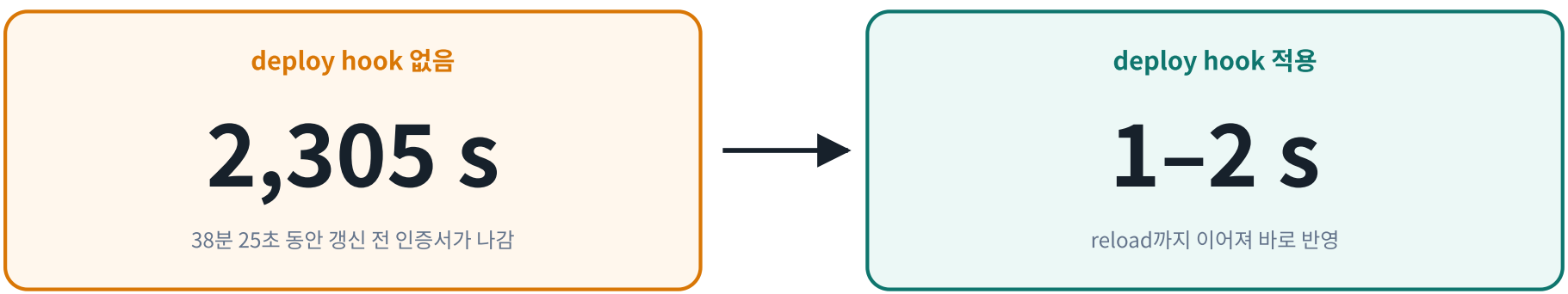
08 · TLS Renewal

인증서 파일이 갱신된 뒤 실제 serving까지 38분 25초가 비어 있었습니다

갱신은 성공했는데 왜 옛 인증서가 나가는가? · certbot timer · Host Nginx

certbot timer는 성공하고 있었지만 deploy hook이 없었습니다. 새 certificate 파일이 생긴 시점과 Nginx worker가 그 파일을 다시 읽는 시점이 분리돼 있어서, 그 사이에는 갱신 전 인증서가 계속 나갔습니다.

certificate 파일이 갱신된 시점 → 그 certificate가 실제로 serving되는 시점



certbot timer는 성공하고 있었고, 빠져 있던 것은 reload를 거는 deploy hook이었습니다

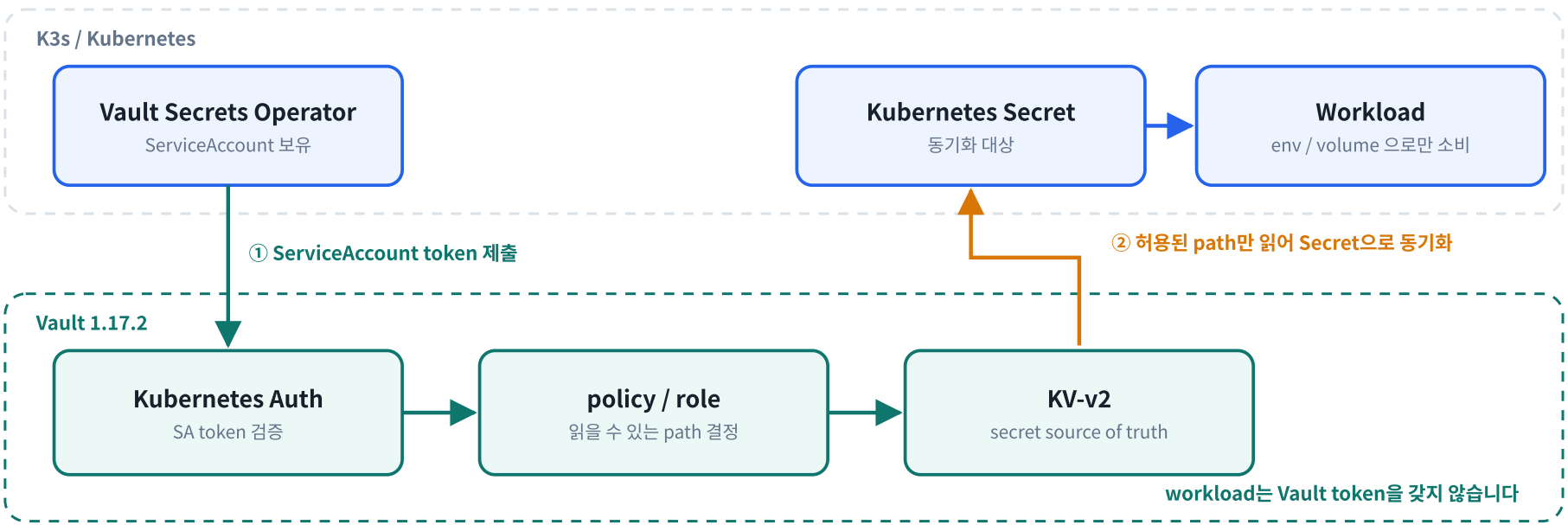
- renewal → serving gap: **2,305 s → 1–2 s** (deploy hook 적용 후)
- 별도의 manual reload 부하 실험: **8,856 / 8,856 HTTP 200**, p95 **205.7 ms → 204.3 ms**
- reload 중이던 845 KB 응답도 **845,361 bytes 전부 수신**

09 · Vault · Secret Delivery

workload는 Vault token 없이 Kubernetes Secret만 소비합니다

누가 인증되고 누가 secret을 읽는가? · Vault 1.17.2 · Vault Secrets Operator · Kubernetes Auth · KV-v2

secret의 source of truth는 Vault KV-v2에 두되, Vault와 직접 이야기하는 주체는 Vault Secrets Operator 하나로 제한했습니다. VSO가 자기 ServiceAccount token으로 Kubernetes Auth를 통과하면, policy와 role이 읽을 수 있는 path를 정하고, 그 결과만 Kubernetes Secret으로 동기화됩니다.



workload는 Vault API도 Vault token도 다루지 않습니다. secret을 바꿀 때 rotation이 무중단으로 반영되는지는 따로 측정하지 않았으므로 여기서는 전달 경로까지만 이야기합니다.

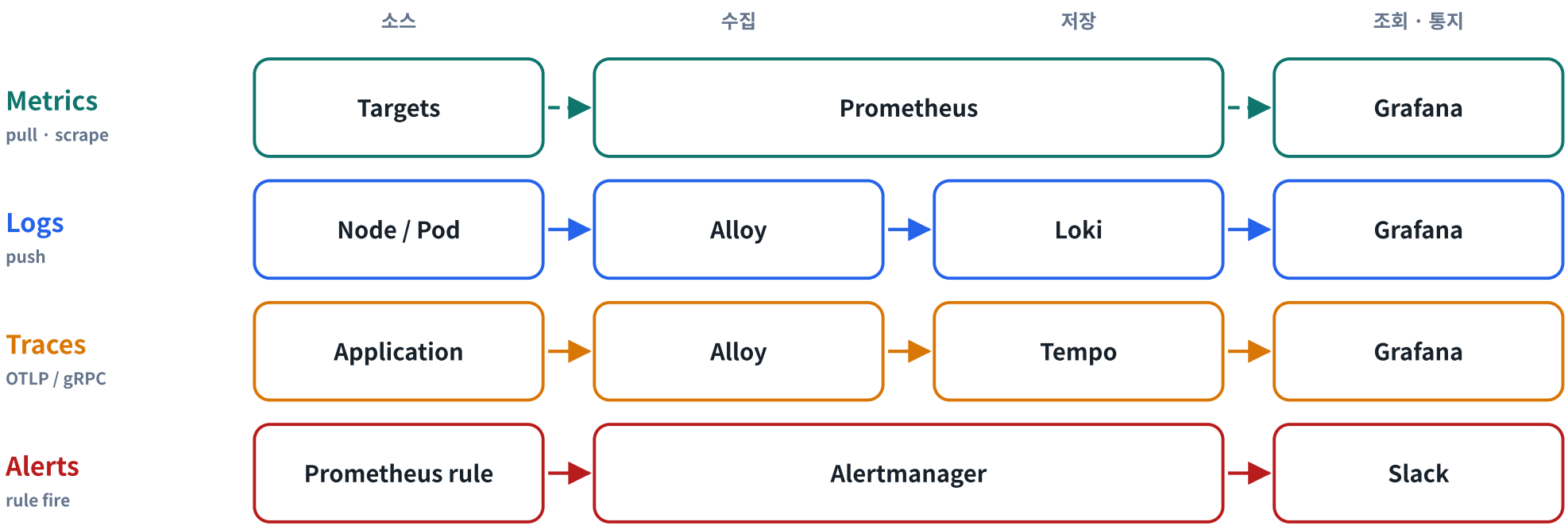
Scope — Vault 1 replica · file storage · local-path PVC · manual init/unseal. 3-node Raft, auto-unseal, full Vault TLS 구성으로 확대하지 않습니다.

10 · Observability

metric · log · trace · alert는 보내는 방식부터 다릅니다

어떤 신호가 어디에 저장되고, 무엇으로 이어 보는가? · Prometheus · Loki · Tempo · Alertmanager

네 신호를 한 덩어리로 다루면 "어디를 봐야 하는가"에 답할 수 없습니다. 수집 방식부터 다릅니다. Prometheus는 target을 직접 scrape하고(pull), log와 trace는 소스가 Alloy로 보냅니다(push).



Grafana는 세 저장소를 같이 조회하는 지점이지 저장소가 아니고, alert는 Prometheus rule에서 갈라져 Slack으로 나갑니다. 한 요청을 metric·log·trace에서 이어 보려면 저장소가 아니라 request id 같은 공통 key를 각 신호에 같이 실어야 합니다.