slaptain

A Kubernetes operator for multi-site, multi-master OpenLDAP

Dominik Epple · LDAPCon

dominik.epple@gmail.com

slaptain.chuck-chuck-chuck.net · Apache-2.0

For starters: One question

Hands up:

Who in this room has OpenLDAP on Kubernetes **on the table**?

You run it. You would like to run it. Or you suspect somebody will decide it for you.

Hand up? This talk is for you — I have something that might make it better.

Hand down? This talk is for you too — don't tune out. The grass on the Kubernetes side might be surprisingly green.

Agenda

1. Motivation: The case for Kubernetes
2. What slaptain wants to do (and already does)
3. The `slapd` image we ship
4. Single-site and multi-site architecture
5. Backup and restore
6. How this got written

About me

Dominik Epple

DAY JOB

Principal Architect at Open-Xchange

Email, Webmail, Groupware, as Software, and as Managed Service[1]

Kubernetes-first, all in: our own components, and the entire stack (yes, this includes OpenLDAP)

dominik.epple@open-xchange.com

1 — documentation.open-xchange.com/appsuite/operation-guides/

About me

Dominik Epple

SPARE TIME

New hobby: Creating missing Kubernetes operators.

slaptain is one of them.[1]

dominik.epple@gmail.com

Today I am here as this one.

1 — github.com/chuck-chuck-chuck-net/littlered for Redis is the other one

Behind the name

`slapd`

n. stand-alone LDAP daemon

"It listens for LDAP connections on any number of ports (default 389), responding to the LDAP operations it receives over these connections."

— `slapd(8)`

A Kubernetes operator's job is to steer, automate and manage the lifecycle of a workload.

Making it the captain of `slapd` is a perfect conceptual fit.

`slaptain = slapd + captain`

And `chuck-chuck-chuck.net`? That is the little red hen, who said it a lot — github.com/chuck-chuck-chuck-net/littlered#behind-the-name

Motivation

It is worth seeing where this leads.

It works.

A container is process isolation — same kernel, same syscalls

Persistence is the part Kubernetes already solved

> Postgres, Cassandra, Kafka run this way. This is not new ground.

It works **like a charm.**

Compute, storage, network, secrets — declared, and kept that way

Upgrading is a new image tag. Rolling back is the old one.

The same files run in a lab, in a cloud, or for one test run

Profit!

"Do you run on Kubernetes?" is now a procurement question

The directory stops being the exception with its own tooling

> The last workload on VMs does not cost you a VM.
> It costs you everything it takes to keep running VMs.

And my personal motivation?

Learn agentic coding. Agentic methods are disrupting software engineering.

On something real. A gap that is blocking my day job.

In spare time. Because it is not part of that job.

As open source. To give back, and to benefit as well.

> Which is why replication, migration and multi-site were in scope from day one.
> The success criterion was never a nice demo. It was my reality at work.

And what I want this talk to do

Know that slaptain exists. That is most of it already.

About to write your own Helm chart? Try slaptain first.

Need an LDAP lab? A namespace each, a hundred of them, in seconds.

Multi-site, multi-cluster? You need slaptain. Or find something like it — I did not.

Then tell me. Bugs, missing features, bad defaults, bad replication design.

> Use it. Clone it. Fork it. Star it. Break it — and tell me how.
> Everything you report is something I can fix (maybe).

Requirements

Goals. What we set out to achieve.

`slaptain` requirements

THE PLATFORM

Make OpenLDAP meet the expectations people have of a cloud-native, Kubernetes-first application — which, by itself, it does not.

Declarative end to end. Fully gitops-able.

THE DIRECTORY

Production-grade OpenLDAP out of the box.

N-way multi-master delta-syncrepl, plus read-only consumers.

Multi-site and multi-cluster, with whichever cross-site transport that needs.

THE LIFECYCLE

Migration of existing deployments — syncrepl, delta-syncrepl, hot and cold.

Backup, restore, disaster recovery.

THE QUALITY BAR

First-class security by default — where it does not cost functionality.

Tooling worth having: deployment, observability, debugging, bug reports.

Requirements, and where we stand

- ☑ Declarative end-to-end, gitops all the way
- ☑ N-way multi-master delta-syncrepl, plus consumers
- ☑ Multi-site, cluster per site, cross-site transports
- ☑ Migration of existing deployments, hot and cold
- ☐ Production-grade OpenLDAP out of the box
- ☑ Backup, restore, disaster recovery — to S3
- ☑ Databases, schemas, ACLs as custom resources
- ☑ Security defaults that do not cost functionality
- ☑ Tooling: `slctl`, conditions, e2e across three sites

That last one is not mine to tick.

Our purpose-built image

What runs in the pod, before anything orchestrates it.

The `slapd` container image

Purpose-built container image: Control and trust; requirements and best practices.

base	`gcr.io/distroless/base-debian13`
built from	`debian:trixie-slim` builder — Debian's own OpenLDAP packaging[1]
runs as	uid/gid 1024 · ports 1024 / 1025 · no capabilities
root fs	read-only
config	`cn=config`, live-writable — no `slapd.conf` at runtime
volumes	`config` · `data` · `accesslog`
bootstrap	init container, same package set — no entrypoint script

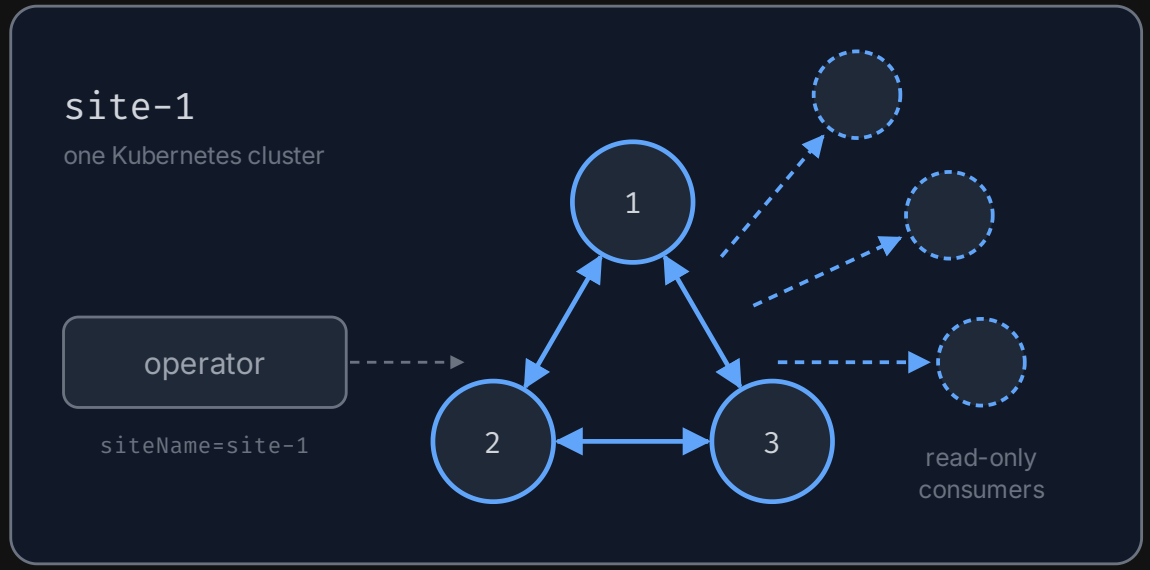
1 — Currently: 2.7.1 built by us with Debian's own packaging scripts. Revert to Debian upstream once it ships 2.7.

First, one site

What it is, what you write, and how you check it worked.

One site

Three slapd pods, all read-write, all equal. One operator, and nothing else in the picture.



Every pod replicates from every other — N-way multi-master, no primary. Their `olcServerID`s are **1, 2, 3**: derived from the pod's ordinal, never configured by hand.

The dashed ones are **consumers**: they pull from the mesh and take reads. Nothing replicates from them, and they have no `olcServerID` at all.

Installing the operator

```
% helm install -n slaptain-system slaptain \
> oci://ghcr.io/chuck-chuck-chuck-net/charts/slaptain
```

Seven new kinds

The API server can now store, validate and serve `SlapdMesh`, `SlapdCluster`, `SlapdDatabase` and `SlapdSchema` — plus three more for backup and restore.

Until now it knew generic terms for compute, storage, network. Now it has words for a *directory* too.

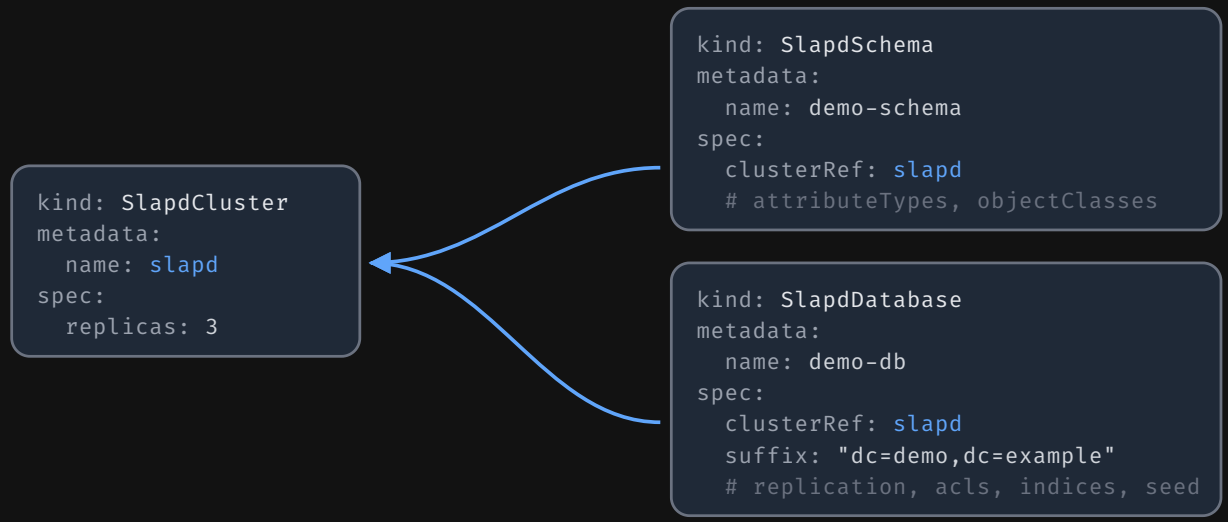
One pod

The controller. It watches those kinds and does nothing whatsoever until one of them appears.

Everything for the rest of this talk is its reaction to an object somebody applied.

`kubectl apply` gives you a noun and a verb. Everything between that and a running, correctly configured, replicating directory is knowledge — and it lives in that pod.

`slaptain` Custom Resources



The cluster carries the running things

pods, volumes, services, TLS, how many replicas and how many consumers. Nothing about what is *in* the directory.

The database and the schema carry the directory

a suffix, its ACLs, its indices, its replication — and schema separately, because schema outlives any one database.

One cluster can hold several databases. That is why they are separate objects and not fields on one.

How boring can a cluster be?

Replicas, TLS, replication on — and defaults for the rest. Nothing here carries meaning: the suffix, the ACLs, the indices, the schema all live in the other objects.

```yaml
# slapdcluster.yaml
apiVersion: ldap.chuck-chuck-chuck.net/v1alpha1
kind: SlapdCluster
metadata:
  name: slapd
  namespace: slaptain
spec:
  ldap:
    tls:
      enabled: true
      secretName: slapd-tls
  replicas: 3
  readReplicas: 1
  replication:
    enabled: true
```

```
% kubectl apply -f slapdcluster.yaml
```

What that produces

Two StatefulSets — `slapd` 3/3 and `slapd-readonly` 1/1.

Volumes per pod — read-write: `config`, `data`, `accesslog`. Read-only: `config`, `data`.

Four Services — `slapd` and `slapd-readonly` for clients, a headless pair for the pods themselves.

The Services answer on `389` and `636`. The pods listen unprivileged on `1024` and `1025` — clients never find out.

And now the considerably less boring part

```
# kubectl apply -f slapdschema.yaml
kind: SlapdSchema
metadata:
  name: demo-schema
spec:
  clusterRef: slapd
  priority: 100
  attributeTypes:
    - >-
      ( 1.3.6.1.4.1.99999.1.1.3
        NAME 'demoQuota'
        EQUALITY integerMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        SINGLE-VALUE )
  objectClasses:
    - >-
      ( 1.3.6.1.4.1.99999.1.2.1
        NAME 'demoUser'
        SUP inetOrgPerson STRUCTURAL
        MAY ( demoContextId $ demoQuota ) )
```

```
# kubectl apply -f slapddatabase.yaml
kind: SlapdDatabase
metadata:
  name: demo-db
spec:
  clusterRef: slapd
  suffix: "dc=demo,dc=example"
  acls:
    - >-
      to attrs=userPassword
      by self write
      by anonymous auth
      by * none                  # …and two more
  indices:
    - "objectClass eq"           # …and four more
  replication:
    ridSegment: 0
    deltaSync: true
  seed:
    entries:                     # …and four more
      - |
        dn: dc=demo,dc=example
        objectClass: organization
  cleanupPolicy: Retain
```

What differs between the pods

```
slctl ldapsearch --as config --pod slapd-N -- -b cn=config '(|(olcServerID=*)(olcSyncrepl=*))'
```

slapd-0

```
dn: cn=config
olcServerID: 1

dn: olcDatabase={2}mdb,cn=config
olcSyncrepl: {0}rid=002
  provider=ldaps://slapd-1…
olcSyncrepl: {1}rid=003
  provider=ldaps://slapd-2…
```

slapd-1

```
dn: cn=config
olcServerID: 2

dn: olcDatabase={2}mdb,cn=config
olcSyncrepl: {0}rid=001
  provider=ldaps://slapd-0…
olcSyncrepl: {1}rid=003
  provider=ldaps://slapd-2…
```

slapd-2

```
dn: cn=config
olcServerID: 3

dn: olcDatabase={2}mdb,cn=config
olcSyncrepl: {0}rid=001
  provider=ldaps://slapd-0…
olcSyncrepl: {1}rid=002
  provider=ldaps://slapd-1…
```

Same image, same chart, same CRs.
Two facts differ per pod — and the operator derived both.

Everything else in every stanza is identical: `searchbase`, `syncdata=accesslog`, `logbase`, `logfilter`, TLS, `retry`. The `rid` is the database's segment plus the provider's pod ordinal — this database sits in segment 0, which starts at 1 — so a stanza names the peer it points at, and no pod carries one for itself.

Is it actually replicating?

`ldapsearch -b dc=demo,dc=example -s base contextCSN` — anonymous, on every pod.

Before — all four pods

```
contextCSN: 2026…173503.489521Z#000000#001#000000
```

After — all four pods

```
contextCSN: 2026…173503.489521Z#000000#001#000000
contextCSN: 2026…180925.496164Z#000000#003#000000
```

In between, one write — `uid=alice,ou=People,dc=demo,dc=example`, added on *one* pod and read back from another.

`contextCSN` is a vector: one value per server that has ever written.

`#001#` is the founder's seed. `#003#` is slapd-2 taking Alice — the same `sid` as its `olcServerID`. Four pods, one vector, including the read-only replica.

LIVE run from `../slaptain-demo/`

Operator installation

```
helm upgrade --install slaptain \
  oci://ghcr.io/chuck-chuck-chuck-net/charts/slaptain \
  -n slaptain-system --create-namespace
```

RESULT

```
kubectl -n slaptain-system get deploy,pod -l control-plane=controller-manager
```

```
kubectl api-resources --api-group=ldap.chuck-chuck-chuck.net
```

LIVE run from `../slaptain-demo/`

One cluster

```
kubectl create namespace slaptain-demo
```

```
../slaptain/tests/gencert.sh -n slaptain-demo \
  -s slapd -H slapd-headless slapd-tls
```

```
kubectl apply -f manifests/01-cluster.yaml
```

MANIFESTS/01-CLUSTER.YAML

```yaml
apiVersion: ldap.chuck-chuck-chuck.net/v1alpha1
kind: SlapdCluster
metadata:
  name: slapd
  namespace: slaptain-demo
spec:
  ldap:
    tls:
      enabled: true
      secretName: slapd-tls
  logLevel: 16640
  service:
    type: LoadBalancer
```

LIVE run from `../slaptain-demo/`

What the operator made

Thirteen lines of CR, and everything Kubernetes was told to create

```
kubectl -n slaptain-demo get slapdcluster,statefulset,svc,secret,pvc
```

What the operator believes about it — status and conditions

```
kubectl -n slaptain-demo get slapdcluster slapd
```

And the configuration itself, answering an LDAP query

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 \
  -b 'cn=config' -s base '(objectClass=*)'
```

Every database in the server, read out of the server

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 \
  -b cn=config -s sub '(objectClass=olcDatabaseConfig)' olcSuffix olcDbDirectory
```

LIVE run from `../slaptain-demo/`

A schema

Attribute types and object classes, into a running server

```
kubectl apply -f manifests/02-schema.yaml
```

MANIFESTS/02-SCHEMA.YAML

```yaml
kind: SlapdSchema
spec:
  clusterRef: slapd
  priority: 100
  attributeTypes:
    - >-
      ( 1.3.6.1.4.1.99999.1.1.1 NAME 'demoContextId'
        EQUALITY integerMatch SINGLE-VALUE
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )
    - >-
      ( 1.3.6.1.4.1.99999.1.1.3 NAME 'demoQuota'
        EQUALITY integerMatch SINGLE-VALUE
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )
  objectClasses:
    - >-
      ( 1.3.6.1.4.1.99999.1.2.1 NAME 'demoUser'
        SUP inetOrgPerson STRUCTURAL
        MAY ( demoContextId $ demoQuota ) )
```

LIVE run from `../slaptain-demo/`

A database

A suffix, its ACLs, its indices, its replication settings — and seed entries

```
kubectl apply -f manifests/03-database.yaml
```

What came of it

```
kubectl -n slaptain-demo get slapddatabase demo-db
```

MANIFESTS/03-DATABASE.YAML

```yaml
kind: SlapdDatabase
spec:
  clusterRef: slapd
  suffix: "dc=demo,dc=example"
  acls:
    - >-
      to attrs=userPassword
      by self write
      by anonymous auth
      by * none
    - >-
      to * by * read
  replication:
    ridBase: 100
    deltaSync: true
  indices:
    - "uid eq,sub"
    - "entryCSN eq"
  seed:
    entries:   # dc=demo, ou=People, ou=Groups, ...
```

LIVE run from `../slaptain-demo/`

Did it land?

The schema entries, now in `cn=schema,cn=config`

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 \
  -b cn=schema,cn=config -s one dn
```

The subschema, as any client sees it — ours among the standard ones

```
slctl ldapsearch -n slaptain-demo --anonymous -o ldif-wrap=no \
  -b cn=subschema -s base objectClasses attributeTypes | grep -i demo
```

And the database the CR produced, as `slapd` has it

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 -o ldif-wrap=no \
  -b cn=config '(olcSuffix=dc=demo,dc=example)'
```

LIVE run from `../slaptain-demo/`

Alice

An entry that could not exist if the schema had not landed

```
slctl ldapadd -n slaptain-demo <<'LDIF'
dn: uid=alice,ou=People,dc=demo,dc=example
objectClass: demoUser
cn: Alice Example
sn: Example
uid: alice
mail: alice@demo.example
userPassword: alicePassw0rd
demoContextId: 42
demoQuota: 1048576
LDIF
```

Read back, custom attributes and all

```
slctl ldapsearch -n slaptain-demo -b "uid=alice,ou=People,dc=demo,dc=example" \
  cn mail demoContextId demoQuota
```

LIVE run from `../slaptain-demo/`

Three masters, and a replica

What actually differs between the two files

```
diff -u manifests/01-cluster.yaml \
  manifests/04-cluster-ha.yaml
```

Apply it and watch it converge — one paste, so the watch is running before the pods are

```
kubectl apply -f manifests/04-cluster-ha.yaml
kubectl get slapdcluster -n slaptain-demo slapd -w
```

```
NAME    PHASE       READY   REPLICAS   AGE
slapd   Degraded    1       2          20m
slapd   Degraded    2       3          20m
slapd   Degraded    2       3          20m
slapd   Running     3       3          20m
```

`Degraded` is the honest answer while a pod is still joining, not an error. The read-only pod has columns of its own, under `-o wide`.

No dump. No restore.
The new pods sync from `slapd-0`.

THE WHOLE CHANGE

`replicas` default `1` → **`3`**
N-way multi-master

`readReplicas` default `0` → **`1`**
one pure read-only consumer

`replication.enabled` default `false` → **`true`**
delta-syncrepl, with its accesslog database

Everything else in the file is unchanged: same suffix, same TLS Secret, same PVC defaults, same Service type.

LIVE run from `../slaptain-demo/`

One site, checked

One binary, from your laptop. Nothing installed next to `slapd`.

```
slctl inspect -n slaptain-demo slapd
```

WHAT TO LOOK AT WHILE IT SCROLLS

`contextCSN`	identical on all four pods, down to the digit — Alice is everywhere
`syncRepl`	two stanzas on each read-write pod, three on the read-only one. No pod replicates from itself
`rid=021 022 023`	derived from the pod ordinals. Nobody allocated them, and nothing in the YAML mentions them
`namingContexts`	the accesslog database is on the read-write pods only — the read-only one has no use for it

14 passed, 0 warnings, 0 failed.

LIVE run from `../slaptain-demo/`

Write here, read there

Write it on `slapd-0`

```
slctl ldapadd -n slaptain-demo --pod 0 <<'LDIF'
dn: uid=bob,ou=People,dc=demo,dc=example
objectClass: inetOrgPerson
cn: Bob Example
sn: Example
uid: bob
LDIF
```

Read it back from `slapd-2`

```
slctl ldapsearch -n slaptain-demo --pod 2 \
  -b "uid=bob,ou=People,dc=demo,dc=example" cn
```

And ask all four where they think they are

```
for p in 0 1 2 readonly-0; do
  echo "-- pod $p"
  slctl ldapsearch -n slaptain-demo --anonymous --pod "$p" \
    -b dc=demo,dc=example -s base contextCSN | grep '^contex
done
```

```
contextCSN: 20261007061055.403193Z#001#
```

The same value on every pod, to the microsecond. One serverID in it — `slapd-0` is the only pod that has taken a write.

LIVE run from `../slaptain-demo/`

What you wrote, and what was derived

The same query as before — the answer is longer now

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 -o ldif-wrap=no \
  -b cn=config '(olcSuffix=dc=demo,dc=example)'
```

And the modules the server has loaded — not the list it booted with

```
slctl ldapsearch -n slaptain-demo --as config --pod 0 \
  -b cn=config -s sub '(objectClass=olcModuleList)' olcModulePath olcModuleLoad
```

Four lines of YAML went in

a suffix, its ACLs, its indices. Kept verbatim.

The rest is arithmetic

rids, peers, the accesslog wiring, the replication identity and the ACL that lets it read, the indices replication needs. None of it is in any file.

The image boots with `back_mdb` and nothing else. Turning replication on added `syncprov` and `accesslog` to `cn=module{0},cn=config` over `ldaps://` — no restart, no second image, no `slapd.conf`.

Then, the mesh

The same three questions, at three sites.

First things first: a site identity

Going multi-cluster, every site — or rather, every site's *operator* — has to be told which site it is on.

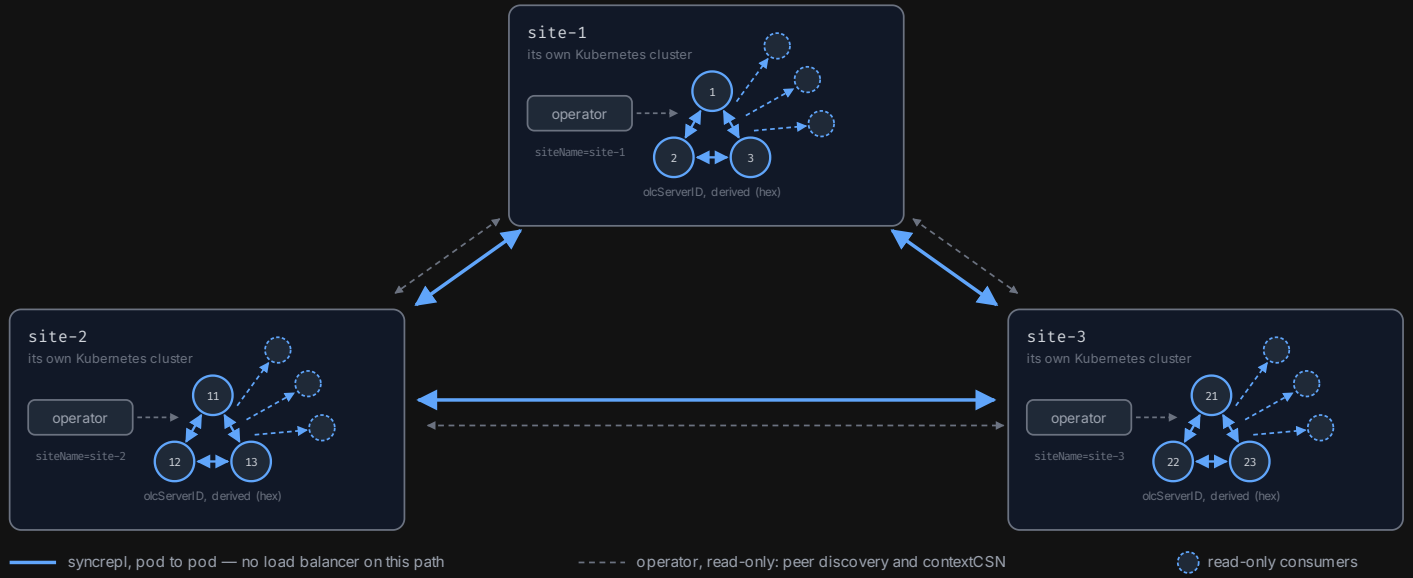
```
% helm install -n slaptain-system slaptain \
> oci://ghcr.io/chuck-chuck-chuck-net/charts/slaptain \
> --set siteName=site-2
```

Exactly what it needs that for comes later in this section. For now it is enough to picture the alternative: an operator asked to orchestrate a mesh, with no idea which of the sites in that mesh is the one it is standing in.

One flag, at install time, outside every custom resource. Everything else the site needs, it derives.

Three sites, all of them peers

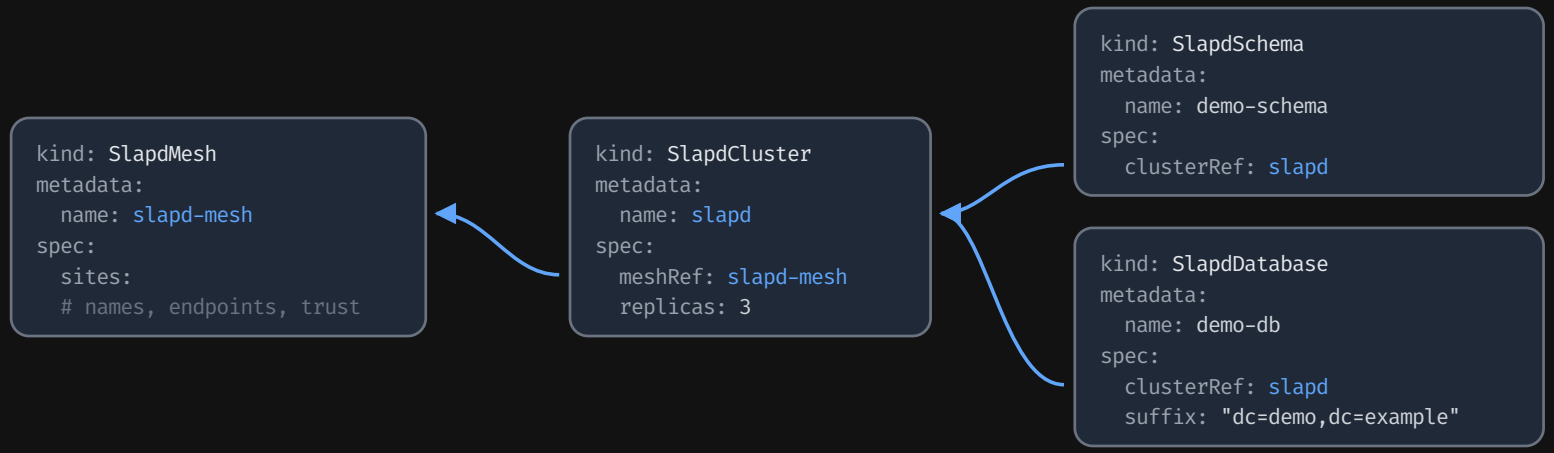
No hub, no meta-cluster, no shared control plane. Three Kubernetes clusters, each with its own operator, each of which keeps working while the other two are unreachable.



An operator only ever **reads** across sites — peer discovery from the remote API server, `contextCSN` from the remote pods. It writes configuration to its own pods and nowhere else.

No primary, no leader election. Every pod shown is an equal read-write peer, and `olcServerID` is derived from `siteName` — never configured.

A fourth kind: `SlapdMesh`



What the mesh carries

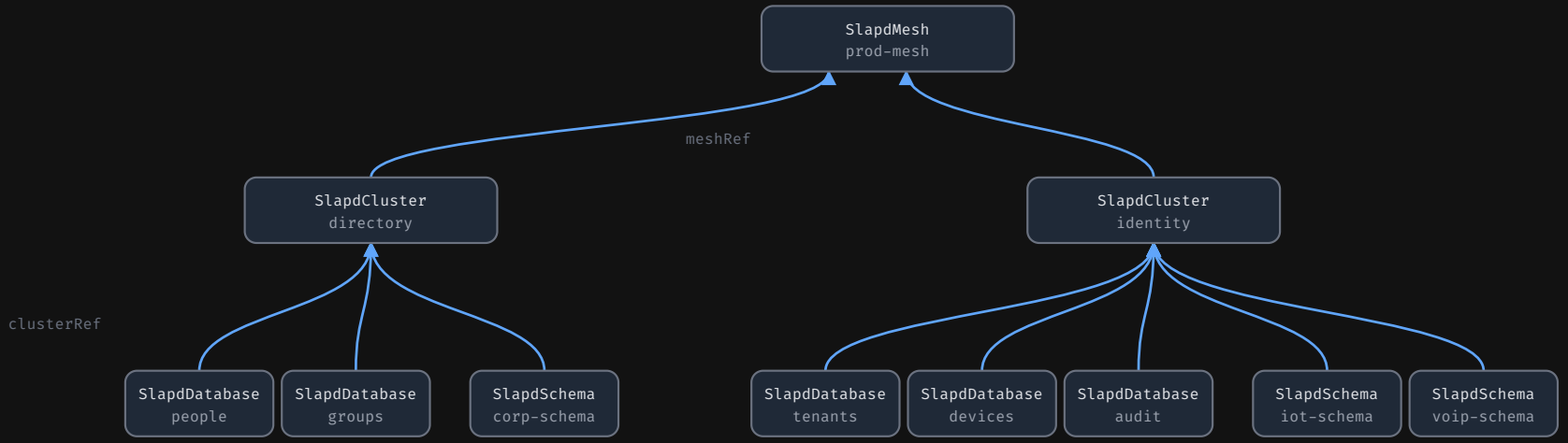
The sites, and the fabric between them: names, endpoints, network mode, trust. Nothing else.

Why the mesh is its own object

The site and topology facts that every cluster would otherwise repeat, written once and shared. On its own it reconciles nothing — it waits, inert, until a cluster references it.

It composes

One mesh. Two LDAP clusters that share its sites. Eight more objects.



The mesh is written once. Both clusters point at it, and neither one owns it.

No fact is written twice. Each one has a single authoritative home and everything else points at it — nothing to keep in sync, nothing that can drift.

All about topology

```yaml
apiVersion: ldap.chuck-chuck-chuck.net/v1alpha1
kind: SlapdMesh
metadata:
  name: slapd-mesh
  namespace: slaptain
spec:
  network:
    mode: pod-routed
    useForInCluster: false
  sites:
  - endpoint: https://10.2.1.10:6443
    name: site-1
    serverIDIndex: 0
  - endpoint: https://10.2.2.10:6443
    name: site-2
    serverIDIndex: 1
  - endpoint: https://10.2.3.10:6443
    name: site-3
    serverIDIndex: 2
```

Pods count up from `serverIDIndex × 0×10 + 1`:

```
site-1:  0×1,  0×2, …
site-2: 0×11, 0×12, …
```

Which is exactly how a `contextCSN` shows them:

```
…Z#000000#001#000000
…Z#000000#011#000000
```

The `#sid#` field is hex. So the stride is too.

Stable `olcServerID`s — the `sid` in every CSN — need stable `serverIDIndex` fields.

So they are explicit, not positional. Reordering or removing a `sites:` entry must never renumber a neighbour.

A `meshRef` makes it multi-site

Back to the `SlapdCluster` from the single-site section. It does not change — it gains one field.

One line added

```yaml
# slapdcluster.yaml — unchanged from earlier
kind: SlapdCluster
metadata:
  name: slapd
spec:
  meshRef: slapd-mesh      # ← that is all
  ldap:
    tls: { enabled: true, secretName: slapd-tls }
  replicas: 3
  readReplicas: 1
  replication:
    enabled: true
```

what it replaces — still available

```yaml
  replication:
    enabled: true
    serverIDBase: 32
    externalPeers:
      - name: site-1
        siteIndex: 0
        discovery:
          kubeconfigSecret:
            name: site-1-kubeconfig
      - name: site-2
        siteIndex: 1
        discovery: …
```

And the database's `rid`s follow from the same reference. Each database owns a band — twenty wide: eight local pod ordinals, then twelve peer-site slots — and a `rid` is a coordinate in it. The pods you saw earlier carry `002` / `003` for their neighbours here, `010` / `011` for the two sites across the wire.

`SlapdMesh` is autoconfiguration: one reference turns a single-site cluster into one site of a multi-site, N-way multi-master, delta-syncrepl directory.

The two objects, together

Both are applied at every site, byte-identical. Between them they define the whole mesh — and the operator derives the rest from `siteName`.

The mesh — who exists

```yaml
kind: SlapdMesh
spec:
  network:
    mode: pod-routed
  sites:
  - name: site-1
    endpoint: https://10.2.1.10:6443
    serverIDIndex: 0
  - name: site-2
    endpoint: https://10.2.2.10:6443
    serverIDIndex: 1
  - name: site-3
    endpoint: https://10.2.3.10:6443
    serverIDIndex: 2
```

The cluster — one line added

```yaml
kind: SlapdCluster
spec:
  meshRef: slapd-mesh
  replicas: 3
  readReplicas: 1
  replication:
    enabled: true
```

Neither object names *which* site it is on. That comes from the operator's own `--set siteName`, and it is the only per-site input anywhere.

find myself
`siteName` → my entry in `sites:` → my `serverIDIndex`

my identity
`index × 0×10 + ordinal + 1` → each pod's `olcServerID`

my peers
the same list minus me → the cross-site syncrepl stanzas

What every site applies

The same four objects, byte-identical at every site. None of them names a site.

SlapdMesh

The sites, the fabric between them, the trust. **The *only* object that knows a mesh exists at all.**

SlapdCluster

The single-site object, **plus one line.** Every tunable and sizing choice it carried, it keeps — and that one line is the *entire* replication topology.

SlapdSchema

Mesh-agnostic — byte-for-byte the object from the single-site section. It has no idea any of this is happening.

SlapdDatabase

Also mesh-agnostic, and also unchanged — suffix, ACLs, indices, seed, and its `ridSegment`, still declared by hand.

> One site-specific input, and it is not in any of them: `siteName`, in the *operator's* own config. From it the operator filters itself out of the mesh's site list, takes its own serverIDs, and builds its peer set.

No custom resource knows which site it is on.

A Helm chart ships them together, but that is packaging. The mechanism is the objects — and `ridSegment` is a reminder that not everything is derived: it is declared, identical everywhere, and the operator refuses two databases that claim the same one.

The whole mesh, checked

One binary. Your laptop. No agent on the server, nothing installed next to slapd.

```
% slctl inspect -n slaptain slapd --short
```

```
[OK ] mesh-resolution            2 external peer(s) derived from SlapdMesh "slapd-mesh"
[OK ] csn-convergence            2 database(s): all 4 pods report identical contextCSN for each
[OK ] suffix-uuid-agreement      2 database suffix(es): all pods agree on the base entry's entryUUID
[OK ] accesslog-consistency      8 database/pod pair(s): logbase = olcAccessLogDB = log olcSuffix
[OK ] syncrepl-stanza-count      2 replicated database(s): RW 4 stanzas each, RO 3 each
[OK ] syncrepl-skip-self         no RW pod replicates from itself on any database
[OK ] rid-uniqueness             all RIDs unique within each pod, across its databases
[OK ] external-syncrepl          every replicated database on every RW pod has stanzas for all 2 external peers
[OK ] cross-site-csn             all external peers report Synced
… and nine more
```

18 passed, 0 warnings, 0 failed.

Three sites, two databases, four pods per site. In the replicated database the `contextCSN` carries three serverIDs — `#001#`, `#011#`, `#021#`. The field is hex: that is site 0, site 1 and site 2, each on its pod 0. Nobody configured a serverID anywhere in this deck.

The same command, without `-short`

```
Pod: slapd-0 [Running, ready]
contextCSN (dc=example,dc=org):
  20261005213026.317713Z#000000#001#000000
  20261005213422.575587Z#000000#011#000000
  20261005213432.756583Z#000000#021#000000
syncRepl (dc=example,dc=org, in-cluster):
  {0}rid=002 provider=ldaps://slapd-1.slapd-headless…:1025 refreshAndPersist
  {1}rid=003 provider=ldaps://slapd-2.slapd-headless…:1025 refreshAndPersist
syncRepl (dc=example,dc=org, cross-site):
  [site-2] {2}rid=010 provider=ldaps://10.233.66.27:1025
  [site-3] {3}rid=011 provider=ldaps://10.233.128.237:1025
syncRepl (dc=second,dc=example,dc=net, in-cluster):
  {0}rid=022 … {1}rid=023
syncRepl (dc=second,dc=example,dc=net, cross-site):
  [site-2] {2}rid=030 … [site-3] {3}rid=031
… then slapd-1, slapd-2, slapd-readonly-0 — and the 18 checks across all of them
```

Two databases, two bands, twenty apart: `002–003` / `010–011` against `022–023` / `030–031`. The `#021#` higher up is a *serverID*, in hex — different thing, different base. Read out of a running server, not out of the spec that produced it.

Backup and restore

A directory is only as good as the copy you can put back.

How a backup is taken

Two steps, and the separation is deliberate.

SPOOL, THEN UPLOAD

1. `slapcat -b <suffix> | gzip` into a scratch directory on the node's disk
2. upload that file to S3

Why not stream straight to the bucket? Because a `slapcat` that dies half way would still leave a complete-looking object behind. Spooling first means a failed dump fails the backup, and no object is ever created.

The price is disk and a little time. The artifact is a plain `slapcat` LDIF — `gzip -dc` and read it.

HOW IT READS THE DATA AT ALL

We ask Kubernetes for volumes with access mode `ReadWriteOnce` — and that means one **node**, not one pod. Two pods on the same node can hold the same volume at once: the node mounts the filesystem once and bind-mounts it into each container.

So the backup is its own pod, pinned to the node `slapd-0` is on, reading the same `/data` while slapd keeps serving. `slapcat` on `back-mdb` is safe online — this is not an outage.

A finished pod still counts as a user of the volume — `kubernetes.io/pvc-protection` does not check the phase — so the operator deletes its own Jobs. Otherwise the first backup quietly prevents the cluster from ever replacing a pod.

One backup, asked for and answered

```yaml
kind: SlapdBackup
metadata:
  name: example-db-backup
spec:
  databaseRef: example-db
  storage:
    bucket: my-ldap-backups
    prefix: prod
    region: eu-central-1
    endpoint: https://s3.example:7443
    credentialsSecretName: my-s3-creds
```

The object key is legible from the bucket alone:
`<prefix>/<cluster>/<database>/<timestamp>.ldif.gz`

```yaml
status:
  phase: Completed
  sourcePod: slapd-0
  sourceContextCSN:
    - …Z#000000#001#000000
    - …Z#000000#002#000000
    - …Z#000000#003#000000
  conditions:
    - type: SourceConverged
      status: "True"
    - type: SourceSuffixHealthy
      status: "True"
```

A backup is **one pod's copy of one database at one instant** — and the CSN vector says which instant, in replication's own units.

Those conditions are not a gate. A backup always takes a backup and records its circumstances.

LIVE run from `../slaptain-demo/`

A place to put it

An S3 endpoint in its own namespace — standing before any of this started, and still standing when the demo namespace goes away

```
kubectl get deployment,service -n slaptain-s3
```

The credentials, though, have to be in the *demo* namespace — `credentialsSecretName` is a `LocalObjectReference`, so a Secret next to the bucket would be unreachable

```
kubectl apply -f manifests/00-s3-credentials.yaml
```

A bucket, a region, an endpoint. Nothing in the backup object knows this is not AWS.

It is `versitygw` in front of a volume, reached by cross-namespace Service DNS at `http://versitygw.slaptain-s3.svc:7480`.

LIVE run from `../slaptain-demo/`

Ask for one

One object. A backup is a thing you ask for, not a command you run

```
kubectl apply -f manifests/05-backup.yaml
```

The only line in it that is about this lab is `endpoint`. Leave it out and the same object goes to real S3.

Six lines of storage, one database.

No pod, no node, no node name.

MANIFESTS/05-BACKUP.YAML

```yaml
kind: SlapdBackup
metadata:
  name: demo-backup-1
  namespace: slaptain-demo
spec:
  databaseRef: demo-db
  storage:
    bucket: slaptain-backups
    prefix: demo
    endpoint: "http://versitygw.slaptain-s3.svc:7480"
    region: us-east-1
    credentialsSecretName: s3-credentials
```

LIVE run from `../slaptain-demo/`

Did it run?

The short answer

```
kubectl -n slaptain-demo get slapdbackup demo-backup-1 -o wide
```

And the long one — `spec` as the operator defaulted it, `status` as the Job reported it

```
kubectl -n slaptain-demo get slapdbackup demo-backup-1 -o yaml
```

WHAT TO LOOK AT WHILE IT SCROLLS

`phase`	`Completed`, `sourcePod: slapd-0` — one pod's copy of one database, and the status says whose
`path`	prefix, cluster, database, timestamp — legible from the bucket alone, with no cluster to ask
`sourceContextCSN`	which instant this is, in replication's own units rather than a wall clock
`conditions`	`SourceConverged`, `SourceSuffixHealthy`, `Complete` — circumstances recorded, nothing gated
`spec`	`compression: gzip`, `podAffinity: true` — in the object, and in no file

`sizeBytes: 634` — and a CSN vector saying exactly which instant those bytes are.

LIVE run from `../slaptain-demo/`

Is it really in the bucket?

`slctl aws` reads the endpoint and the Secret out of the backup object, port-forwards to the bucket, and hands the rest to the AWS CLI

```
slctl aws -n slaptain-demo --from sb/demo-backup-1 \
  s3 ls s3://slaptain-backups/ --recursive --human-readable
```

```
→ aws via http://localhost:40231 (SlapdBackup/demo-backup-1, bucket slaptain-backups)
2026-10-07 09:15:55  634 Bytes  demo/slapd/demo-db/20261007T071554Z.ldif.gz
```

And the object itself, out of the bucket and through `gzip`

```
KEY=$(kubectl -n slaptain-demo get slapdbackup demo-backup-1 -o jsonpath='{.status.path}')
slctl aws -n slaptain-demo --from sb/demo-backup-1 s3 cp "s3://slaptain-backups/$KEY" - \
  | gzip -dc
```

A plain `slapcat` LDIF. Getting your data back out does not need slaptain.

Backups, on a schedule

A cron expression, a retention policy, a bucket.

```yaml
kind: SlapdScheduledBackup
metadata:
  name: nightly
spec:
  databaseRef: directory
  schedule: "0 3 * * *"
  compression: gzip
  retention:
    maxCount: 14
    maxAge: 720h
  storage:
    bucket: slapd-prod
    endpoint: https://s3.example:7443
```

```yaml
status:
  lastScheduleTime: 2026-09-29T03:00:00Z
  nextScheduleTime: 2026-09-30T03:00:00Z
  lastBackupRef:    nightly-1790650800
```

Each tick creates an owned `SlapdBackup`. The schedule does no work itself — it is a factory and a broom.

Retention deletes the **S3 object** too, not just the record. Only *Completed* backups are pruned: failed ones are kept on purpose, so the morning after is diagnosable.

You declare the policy. Keeping it is somebody else's problem.

The easy restore is a new cluster

Nothing exists yet. So the backup is simply where the database comes from.

```
kind: SlapdDatabase
spec:
  suffix: dc=example,dc=org
  bootstrapFrom:
    backupRef: nightly-1790650800
```

It takes the place of `spec.seed` — the two are mutually exclusive, and it is one-shot. Or point `bootstrapFrom.s3` straight at an artifact: a `slapcat` dump from a directory that never heard of this operator.

WHY THIS ONE IS EASY

- nothing to scale down and nothing to stop — there are no pods yet
- nothing to wipe: the database is **created from** the artifact, it never had an earlier state
- it is ordinary **desired state** — declare the database, get the data
- and it is the right tool for a **new site**: seed locally, then let syncrepl carry only the delta

Put it on a database that already holds data and it declines — it is never an in-place overwrite. Which is exactly why there is a second way to do this.

Why `SlapdRestore` exists

You could do this by hand: delete the database, declare it again with `bootstrapFrom`. Nothing stops you — and that is worth saying, because the automation is not doing anything you could not.

BUT THE MANUAL VERSION DESTROYS FIRST

Is the artifact the right suffix? Did the gzip survive? Does the replication password in it still match the cluster's?

By hand you find all of that out *after* the data is gone — and every step is one you can skip at four in the morning.

SO THE OPERATOR DOES IT IN THIS ORDER

1 — preflight, cluster still serving. Fetch it, gunzip it, check the first DN *is* the target suffix, verify the replication password against the artifact's own metadata. Fail here and nothing has been touched.

2 — only then scale to zero, and per pod wipe *that database's* directory immediately before loading it. The other databases on the cluster are not touched.

Destroy last.

Nothing is wiped until the artifact has proved it can be put back. And if a pod fails to load anyway, the peers still hold the old data — come back up on them, and fail loudly.

LIVE run from `../slaptain-demo/`

Put it back

One paste, both lines — the watch has to be running *before* the operator starts scaling, or the interesting part is already over

```
kubectl apply -f manifests/06-restore.yaml
kubectl -n slaptain-demo get slapdcluster slapd -w
```

Nothing else to type. The `SlapdCluster` controller drives the whole window; the `SlapdRestore` only asked for it.

MANIFESTS/06-RESTORE.YAML

```yaml
kind: SlapdRestore
metadata:
  name: demo-rollback-1
  namespace: slaptain-demo
spec:
  databaseRef: demo-db
  source:
    backupRef: demo-backup-1
```

Immutable, and one-shot. To restore a second time you create a second `SlapdRestore` — it is a command, not desired state.

WHAT GOES PAST IN THE WATCH

PHASE	`Running` → `Restoring` → `Running`. The cluster says it is in a restore window, not merely unhealthy
READY	3 → 0 → 3, and the read-only pod goes with them. By the time anything scales, preflight has already passed
`status.restore`	`Preflight`, `ScalingDown`, `Restoring`, `ScalingUp` — the sub-state, for anyone who asks where it is

`READY` down to zero and back is the outage — and it is the whole outage.

How this got built

A project this size, in spare time, is only realistic because AI agents did substantial implementation work. So: how that actually works here, where the industry stands — and what I think it costs.

How we work here

Four things, in this order. The first two are mine, and neither is delegated.

1 · EXPERTISE, FIRST

- Kubernetes, operators, distributed state: ground I can stand on.
- `slapd` is not. So: back to school — upstream docs, prior art, configurations that have been in production for years, this conference, and people who know it better than I do.

The agent is not a substitute for that. It is a multiplier on it — including on the gaps.

2 · THE DESIGN IS MINE

- requirements, and the in-scope / out-of-scope argument
- the architecture, and every decision that constrains it
- ADR-driven: the decision, the alternatives, the *why*
- principles, test discipline, history awareness — written into `CLAUDE.md`, and the agents are held to it

None of this is delegated.

Where the agents come in

Three and four. Only the last one is the agents doing the work.

3 · SPARRING

- root-cause analysis on complete evidence (`slctl debug-dump`) — genuinely brilliant at it
- carrying a working thing across: a deployment script is excellent input for an operator
- it has read everything, and proposes components I had never heard of

I never take an RCA I cannot explain myself, and never hand over the decision. Ask for a re-word, more context, a definition — it does not judge, and it explains it again.

4 · THEN THEY BUILD

- solution design and execution plan, worked out together, before a line is written
- an orchestrator briefs implementation agents and reviews what comes back
- red-first, e2e, no regressions
- real labs: three sites, real `slapd` , real replication — no mocks, no facades

A green test against a mock is the most expensive green there is.

Chances, and risks

Personal, and I am not going to talk the second half away.

COGNITIVE DEBT

I think a feature through, completely. I brief it. It goes through implementation, the tests, the plumbing, the small ugly fixes, review, main, release — and I was in almost none of that. Then I context-switch to the next big thing.

Months later I read my own documentation to find out how my own system works.

The mitigation happens to be good practice anyway: document while you still know. There is no other way back in.

BECOMING AN EXPERT

This section started with *be an expert in something*. Two things are true at once:

harder — expertise came from doing, and the doing is shrinking.

easier — "teach me Kubernetes, start at zero" is a genuinely excellent way to learn. I am convinced of that one.

But it no longer happens along the way. It costs explicit attention and explicit energy. Nothing comes for free.

WORKING ALONE

A team of agents is always available, never busy, and never needs the conversation. The tempting conclusion is that you need fewer people around you.

Human-to-human exchange is what quietly gets squeezed out to make the room.

It used to happen by itself. It does not any more — so it has to be culture, and it has to be deliberate.

Great benefits and great risks

Both — and I am not trading the second away to make the first sound better.

This is not going away. Either we learn to use it well and carry what it costs, or that gets settled by the people who do not care.

You only shape the rules if you are in the game.

Where this stands

The same list as before the walkthrough. You have now watched most of it run.

- ☑ Declarative end-to-end, gitops all the way
- ☑ N-way multi-master delta-syncrepl, plus consumers
- ☑ Multi-site, cluster per site, cross-site transports
- ☑ Migration of existing deployments, hot and cold
- ☑ Backup, restore, disaster recovery — to S3
- ☑ Databases, schemas, ACLs as custom resources
- ☑ Security defaults that do not cost functionality
- ☑ Tooling: `slctl`, conditions, e2e across three sites
- ☐ Production-grade OpenLDAP out of the box

First commit 22 February 2026. One lab, one transport tested. One person, spare time, no release. What I cannot get by working harder is the half I do not have — twenty years of running directories in production.

A vision to sustainability

OWNED BY ITSELF

Apache-2.0, developed in the open, with governance that routes through neither me nor an employer. It should outlive my interest, my job, and my being wrong about things.

CO-MAINTAINERS

People with commit rights who know `slapd` better than I do — expertise and bandwidth both. Every design decision in this talk deserves someone able to say "no, and here is why".

GIVE IT A TRY

If your next project pushes you onto Kubernetes, put slaptain on the list — especially if the alternative is writing your own chart. Been there. This is further along.

> Use it, break it, tell me what broke. That is how this grows past the lab and past one person.

So — what is it that we have here?

A toy?

An academic proof of concept?

Or something with a future?

SPARE TIME
dominik.epple@gmail.com

DAY JOB
dominik.epple@open-xchange.com

slaptain.chuck-chuck-chuck.net · Apache-2.0

Back-pocket

Not part of the talk. Here if you ask.

Migration

Getting an existing directory in — with a maintenance window, or without one.

Getting an existing directory in

Nobody starts empty. Two ways in, and the choice is made by what you can afford, not by what is elegant.

COLD — a maintenance window

Freeze the writers, `slapcat` each database on the source, `slapadd` it into slaptain — that is `bootstrapFrom.s3` from earlier — and cut the clients over.

Simple and robust, and it costs a window. It asks *nothing* of the source: no config change, no reachability, no cooperation.

HOT — no window at all

slaptain joins the source's syncrepl mesh as a peer, takes client traffic gradually, and the source's nodes drain and retire.

It needs a source you are allowed to reconfigure — a `syncprov` overlay, a bind user, ACLs that let it read everything. A source whose replication config is frozen leaves only cold.

Choose by **downtime tolerance**, **directory size**, and **whether the source can be touched**. Cold is the floor and the fallback — and it stays available in the middle of a hot migration, which is what makes the hot path safe to attempt.

Hot migration, one stage at a time

Each stage is a stable configuration you can sit in. The operator never advances on its own — every transition is a deliberate edit.

	slaptain	the source	reversible?
0	not present	authoritative, serving everyone	—
1	consumer-only, read-only, pulling	unchanged	**totally**
2	peer, bidirectional	reciprocal stanzas added	yes, with source-side undo
3	peer, taking client traffic	losing client traffic	per client
4	peer, authoritative	draining node by node	until the last node drains
5	authoritative, alone	retired	forward only

1 → 2 is a field edit. Patching `replication.mode` promotes consumer-only to peer **in place** — no pod restart, and `entryUUID`s unchanged, which is how you know nothing re-synced.

Confidence, stated plainly. Stages 1 and 2 are covered by an e2e that stands up a deliberately legacy provider. Stages 3 to 5 are a written runbook, not a tested path.

Why an operator

Why a chart would not have been enough.

A Helm chart would get us far

With that image, we could write a Helm chart.

With thorough topology arithmetics in the bootstrap — *"I am pod 2 of 4, so this is my* `serverID`*, and these are my* `syncrepl` *stanzas."*

That bootstraps a working cluster, and it carries us a long way.

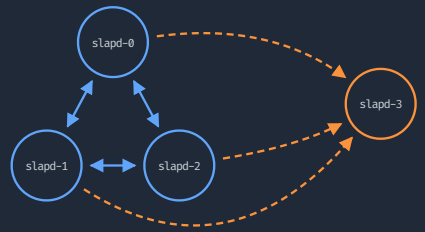
So what does an operator do beyond that?

Adding a fourth pod

Same image, same StatefulSet, same three pods to start with. One thing differs: whether something is still running when you ask for the fourth.

A — the Helm chart

```
% kubectl scale sts slapd --replicas=4
```

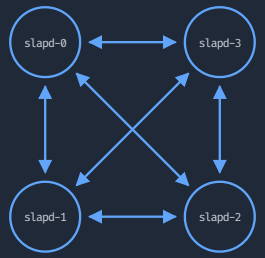


No template change, so nothing restarts — and nothing tells slapd-0..2 that slapd-3 exists.

It receives everything and sends nothing. Writes that land there are lost, and every pod reports Ready.

B — the operator

```
% kubectl edit slapdcluster slapd  # replicas: 4
```



The controller rewrites the syncrepl stanzas in `cn=config` on every existing pod, over `ldaps://`.

Four pods, twelve stanzas, no restart — and no odd one out.

You could *almost* do it with a chart

> `helm upgrade --set replicas=4` *does* converge — at the price of a rolling restart of every pod, and other side effects.

SCALING IS THE SMALL CASE

- **multi-step operations over live data** — backup, restore, migration, replacing a pod without losing its volume
- **anything cross-cluster** — the local config is a function of another cluster's live pods
- **the repairs nobody has named yet**

WHICH THE OPERATOR HAS

- it is **running** — there is a now, not only an install time
- it sees the **whole cluster**
- it knows what `slapd` is
- it can **remember**, in the resource's status
- it can **act**, over `ldaps://` — because the config is a *database*, not a file

the chart is the preprocessor
It runs once, before anything starts — and whatever it computes, it may only emit Kubernetes objects.

the operator is the daemon
It is still there at runtime, and it can do anything a program can do. Including speaking LDAP.

Limited expressive power at install time — or full power, for the whole runtime.

Bootstrap, step by step

What happens between `kubectl apply` and a replicating directory.

Bootstrap, part 1 — what a pod can do alone

The `SlapdCluster` from the previous slide — 3 + 1, TLS on, replication on — lands. Then:

The operator creates — the Secret it reaches the pods with, the Services, and the StatefulSets.[1]

Kubernetes creates the pods — and their PVCs, from the StatefulSets.[2]

The init container runs — makes the LMDB directories,[3] and renders the `slapd.conf` it converts into `cn=config`.[4]

slapd starts — `-F /config/slapd.d`, ports 1024/1025.

Nothing on this slide needs another pod.

1 — `<name>-config-password` is the `cn=config` admin credential the operator binds with. Services: `slapd-headless` and `slapd`, plus the same pair for `-readonly`. Those two and the StatefulSets wait until the TLS Secret is present and usable; until then `TLSReady=False`, and the operator comes back.

2 — pod creation is ordered by the StatefulSet controller (`OrderedReady`): pod-0 is Ready before pod-1 is created. PVCs come from `volumeClaimTemplates` — config, data and accesslog per RW pod, config and data per read-only pod.

3 — `mkdir -p`, on every start. In a fresh `SlapdCluster` only the node-local auth DB exists; workload DBs are added later, via `SlapdDatabase` CRs.

4 — once per volume lifetime: `slaptest -f … -F /config/slapd.d`. Modules · core schemas · `serverID` = ordinal + 1 · TLS paths · the config database — and *no* data database, *no* custom schema, *no* ACLs, *no* syncrepl.

Bootstrap, part 2 — what a pod can’t do alone

The operator dials every pod — one connection each, over headless DNS. Not the Service.[1]

The cluster controller writes — server-global tunables, and the node-local auth database.[2]

The loop converges — and stays there.[3]

That is an *empty* cluster: four pods, a config database, no directory. More happens the moment a `SlapdSchema` or a `SlapdDatabase` exists.

1 — bound as `cn=admin,cn=config`, not through the ClusterIP Service: `cn=config` is node-local, so every pod is visited and written on its own.
2 — node-local, on every pod including the read-only fleet; it holds the per-database syncrepl bind identities.
3 — every pass re-reads and re-compares: drift, hand edits and operator upgrades heal unnoticed. Compare first, write only on divergence.

Bootstrap, part 3 — Adding content

Same loop as before — dial every pod, compare, write only on divergence. Now there is something to write.

The schema controller writes — for each `SlapdSchema`, on every pod: the attribute types and the object classes.[1]

The database controller writes — for each `SlapdDatabase`, on every pod: create the database → ACLs → indices → `syncprov` + `accesslog` overlays → the journal DB → modules → serverIDs → the bind identity → **then** the syncrepl stanzas.

And on pod-0 only, once: the seed.[2]

The order *is* the safety.[3]

1 — schema is global to the slapd process, so it is written per pod, not per database — hence `clusterRef`. Additive-only (ADR-006): you add, you never silently drop.

2 — recorded in the status, never repeated; replication carries it to the peers, and there is no second write path.

3 — a provider that lacks the bind identity rejects every bind with `err=49`, and the link stays dead until the entry appears — while slapd is up and serving. So the stanzas are written last, and withheld until every RW pod carries the identity.

Internals

The derivations, and where the configuration actually lives.

What the operator derives, so nobody maintains it

serverIDBase

`serverIDIndex × 0×10`, then `base + ordinal + 1` per pod — so `site 2, pod 3` is the one a CSN shows as `#024#`, which reads straight off the wire at three in the morning.

externalPeers

Every site the cluster spans, except this one — N×(N−1), a full mesh. Never peer with yourself; a pod must not be its own provider.

Network mode

From `mesh.network` — two clusters on one fabric cannot sensibly disagree about whether pod CIDRs are routed. Peer addresses are then *discovered* from each remote API server, never listed.

Trust wiring

Each site's `caSecretName`, mounted at every other site — N×(N−1) again. The consumer verifies the provider; peer *authentication* is the bind, not the certificate.

Applied in memory, never written back to the spec — writing it back would make each site's object differ from its neighbours' and destroy the property the whole design rests on.

Set any of them by hand *alongside* `meshRef` and the cluster refuses: `MeshResolved=False`, nothing reconciled. Deliberate — an ambiguous `serverIDBase` is how a serverID collision gets in quietly, and slapd validates nothing there.

Where the syncrepl stanza actually lives

Nowhere on disk that you would edit. It is an attribute of a running slapd, written by the operator over `ldaps://`, per pod:

```
dn: cn=config
olcServerID: 35

dn: olcDatabase={2}mdb,cn=config
olcSyncrepl: {0}rid=001 provider=ldaps://…:1025
  type=refreshAndPersist
  searchbase="dc=demo,dc=example"
  bindmethod=simple
  binddn="cn=repl-demo-db,cn=slaptain-auth"
  credentials=…
  logbase="cn=accesslog-demo-db"
  logfilter="(&(objectClass=auditWriteObject)
             (reqResult=0))"
  syncdata=accesslog
  retry="10 +"
```

One stanza per peer

In-cluster peers and cross-site peers, each with its own `rid` out of this database's band — so the number alone says which peer the stanza points at.

delta-syncrepl, per database

`syncdata=accesslog` and a `logbase` that names *this* database's journal: `cn=accesslog-≺db>`, its own LMDB directory under the accesslog volume.

The bind identity

Not `cn=replication,<suffix>` in your data tree. The next slide is about why that matters more than it looks.

The credential, and where it is allowed to live

A syncrepl consumer binds to its provider. That one secret must exist twice, in two forms — the protocol requires both: plaintext on the consumer to present, a hash on the provider to verify.

where it used to live

An entry in the replicated data tree. Which means: it propagates mesh-wide and converges on one winner, while the per-site Secrets do not propagate at all.

Measured: three sites, three generated Secrets, one converged entry — two sites holding credentials that matched nothing, replication dead for **nineteen days**, every health signal green.

Where it lives now

A node-local database, `cn=slaptain-auth`, that is never replicated — no syncprov, no syncrepl, no accesslog, by construction.

The Secret is the single source of truth; the LDAP copy is a projection the operator solely owns, converged on every reconcile like any other node-local state.

→ The problem was never "two stores". It was that one of the two stores was shared, replicated, multi-writer state — in the customer's tree, under customer-defined ACLs. Rotation stopped being a design problem and became an ordinary operation.

Access control on the thing that answers binds

The auth database exists to answer binds. Its ACLs say exactly that, and nothing more:

```
olcAccess: to attrs=userPassword by anonymous auth by * none
olcAccess: to *                  by * none
```

`auth`, not `read`

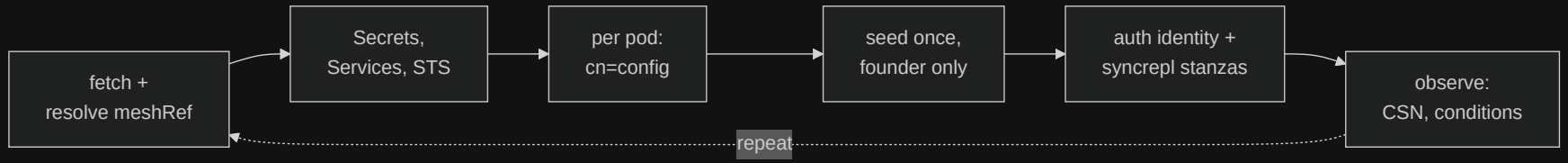
slapd needs the *auth* right to verify a simple bind. Granting only that means a successful bind never implies being able to read the hash back out.

the catch-all is not redundant

A database with no matching rule falls through to the frontend default — and that default is *read*. This database holds every replication credential on the pod.

One identity per database — `cn=repl–<db>,cn=slaptain-auth` — so one database's replication identity cannot read another database's journal. Your own ACLs, schemas and databases are custom resources, and they are yours.

The reconcile loop



A reconcile is not "apply the desired state". One step, repeated — and safe to interrupt at any point, because it will be.

An unresolvable mesh does not fall back and does not half-derive: that would empty the peer set, and the next pass would tear every cross-site stanza off every pod. Report and stall.

Unhealthy → tight retry. Healthy → a slow resync floor, so a rotated Secret is still picked up on a cluster where nothing else happens.

The lab underneath

Where three sites come from, when nobody is paying for three sites.

Where does a spare-time project get three sites?

Everything you just saw — the derived serverIDs, the peer sets, the cross-site syncrepl stanzas — is only worth anything if it was exercised on something that is genuinely three sites. A single `kind` cluster cannot tell you that.

Disjoint, routable pod networks

Cross-site syncrepl is pod-to-pod. Every site's pod CIDR must be reachable, as itself, from every other site.

an API reachable *from pods*

The operator discovers peers by dialling the other sites' Kubernetes APIs — from inside a pod, not from your laptop.

Sites that are actually separate

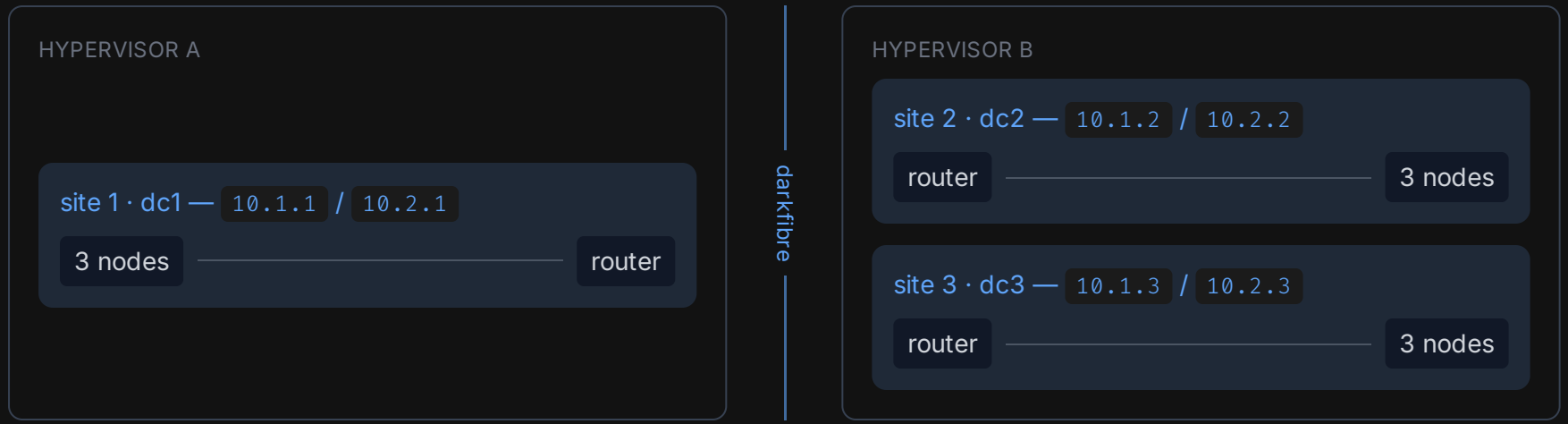
Distinct L3 segments with one path between them. On a flat network every topology bug reads as green.

No managed-cloud magic

Most places OpenLDAP actually runs have none. Whatever the lab leans on has to exist on plain VMs.

> From the basement. Two Linux boxes at home — 192 GB of memory between them and a handful of cores, bought privately. In the evenings one of them is a Steam box. That is the entire budget, and it is enough for three sites.

Two hypervisors, three sites, one darkfibre



libvirt networks in route mode, not NAT — so a VM keeps its real address everywhere, which is the whole point.

One bridged L2 across the two hosts plays the darkfibre. It is the only inter-site path, and a router VM per site is the only thing on it.

An nftables policy on each host drops bridge-to-bridge traffic and masquerades only towards the internet. Without it the sites are accidentally one network.

Eighteen VMs — three sites of five, plus a three-node control plane this project does not use — two physical machines, one `make apply`.

Dual-homed, by role

Every node has two NICs, and which one is which is a decision, not an accident.

internal — `10.1.N.0/24`

Site-local. Consumers, everything else on that site, and the only route to the internet. Does *not* cross sites.

replication — `10.2.N.0/24`

Routed between sites over the darkfibre. Carries cross-site pod traffic and the cross-cluster API. No internet.

The consequence lands directly in the operator's API:

the address in your kubeconfig is not the address a remote pod can dial.

```
sites:
- name: site-1
  endpoint: https://10.2.1.10:6443
  serverIDIndex: 0
```

That `endpoint` is the site's API on the *replication* network. Admin access uses the other one. Two addresses for the same apiserver, and only one of them is peer discovery.

A mesh field that looks like redundant configuration is usually a dual-homed network that has not been admitted to yet.

Every node is its own router

Cilium, native routing, no kube-proxy

Each node owns a pod `/24` out of the site's `/18`. No overlay, no encapsulation — a pod packet keeps the pod address on the wire the whole way.

The payoff

Nothing static about pod CIDRs anywhere. Add a node, drain a node, rebuild a cluster — its `/24` appears and disappears on its own, and no peer or router is hand-edited.

bird2 on every node

A system extension of the immutable OS. It advertises the node's own `/24` upstream and *installs* what it learns into the kernel FIB — which Cilium's own BGP will not do, being advertise-only.

the bill

An ASN tier per role *and* per site. Two co-sited clusters that share an AS number silently drop each other's routes — AS-path loop prevention working exactly as designed, on a topology that is not a loop.

Routers full-mesh over the darkfibre

Three sessions, re-originating each site's prefixes. Learned routes land in the kernel as `proto bird`.

This is what `network.mode: pod-routed` in the `SlapdMesh` is resting on: a pod IP in site 1 is an ordinary routable address in site 3.

The "semi" in semi-cluster-mesh

WHAT IT IS

- routed pod-to-pod, both directions, across sites
- a consumer outside the clusters reaches the site-local pods directly, SNAT'd to its own node address
- discovery by forwarding one DNS sub-zone to the site-local cluster
- **no proxy anywhere on the data path**

WHAT IT IS NOT

- not a service mesh, not a sidecar, not an overlay
- no cross-cluster `Service` abstraction, no global load balancing
- a consumer is pinned to its local site, on purpose — it can only route to the pods it is allowed to route to

We ran a real Cilium ClusterMesh, took it out — nothing queried it, and it sat at **3 of 9 nodes reachable** forever — then put it back when a workload turned up that genuinely needed cross-cluster `Service` semantics.

Why it can never be green: the health probe targets the node's `InternalIP`, picked by a hard-coded priority in the source, with no configuration hook. So the question was never "tune the mesh". It was which NIC carries the node address — and that one reshapes the whole lab.

slaptain is indifferent to all of it. Syncrepl is pod-to-pod; the operator needs routing, not a mesh.

What the lab proves — and what it does not

PROVEN

- three separate clusters, three L3-separated sites
- cross-site pod-to-pod with no overlay and no proxy
- each site’s API dialable from a remote site’s pods — the operator’s own discovery path
- rebuildable from VMs to clusters by automation, not by memory

HONEST

- the darkfibre is a bridge between two machines: no latency, no loss, no partition worth the name
- two machines wearing nine node hats: "three sites" here is a routing arrangement, not three buildings
- one lab, one transport — `pod-routed` is exercised, the alternative is written and untested
- it has taught me more about reboots than about replication

It is not a production proof. It is the smallest thing on which "three sites" stops being an assumption — and that turned out to be enough to find real bugs in the operator.

Things that break, and how they lie to you

nftables is runtime state. A hypervisor reboot silently removes the site isolation and the egress masquerade. Now a systemd unit — and something to *verify*, never assume.

`ip_forward` reverts to 0 on the router VMs, because modern `systemd-sysctl` does not read `/etc/sysctl.conf`. Tell: the box answers on its own addresses and drops every packet that was only passing through.

Asymmetric host routes. Small requests are instant, large uploads black-hole: `kubectl get nodes` fine, `helm` dead after 45 seconds with `http2: client connection lost`. MTU is a red herring; it was a missing route back.

Impatience. After a restart the pods come back with new addresses and a stretched control plane sits in CrashLoopBackOff for minutes after losing quorum. Verifying inside that window produces confident, false failures.

None of these are LDAP problems. They are the rent on a multi-site lab — and the reason a "does it replicate across sites?" question can take a day to answer honestly.

How this got built — in context

Where the industry stands. Kept back because it is other people's argument, not mine.

In context: what does the industry do?

DORA 2025: **90% of development professionals use AI at work.** So the interesting question is not whether — it is under what discipline.

LINUS TORVALDS

Open Source Summit NA, May 2026

"You need to understand not just your prompts, but you need to understand your end result, because that's the only way you can maintain it long term."

The most change-resistant codebase on the planet settled it institutionally — and attached exactly one condition.

ANDREJ KARPATHY

Sequoia Ascent, April 2026

"Vibe coding raises the floor. Agentic engineering raises the ceiling."

"You can outsource your thinking, but you can't outsource your understanding."

The man who coined *vibe coding*, calling it insufficient and naming the replacement a discipline.

THOUGHTWORKS RADAR

vol. 34, April 2026 — ring: Caution

"Codebase cognitive debt: the growing gap between a system's implementation and a team's shared understanding of how and why it works."

The risk I just described, with an industry name, a published definition and a *Caution* ring.

And the honest counterweight — METR, 2025/26: in a randomised trial, experienced developers on familiar codebases were **19% slower** with AI while believing they were 20% faster. Nobody on this slide is claiming the question is closed.

Unclean stops

Something e2e testing surfaced, where it turned out to come from, and what we are doing about it.

How we found it

In e2e — but not by the assertions

We test this thing continuously and try to torture it: kills, drains, scale changes, node reboots.

The suite *passed*. No data lost, every client request answered correctly, every `contextCSN` converged.

Which is the whole point — this does not look like a failure from the outside.

You hear the fans

The lab is mini PCs in a living room. A provider pinned at 2380m CPU is *audible*.

And once you know to look, logging and monitoring are unambiguous: thousands of stale-cookie answers a minute, consumers in permanent full refresh.

The analysis led to **ITS#9398** and its neighbours — long-standing upstream behaviour in delta-syncrepl's journal, not something slaptain introduces.

> Handling it correctly is still ours to do. The contract between Kubernetes and the workloads on it is that a restart is fine — *normally* a graceful one, and normally it is. This is a place where normally is not enough.

It is visible from a long way off

`% kubectl top pod -n slaptain-testing --context scm-sN` — three sites, nothing but CPU and memory.

site-1

slapd-0	409m	734Mi
slapd-1	411m	735Mi
slapd-2	**1698m**	734Mi
slapd-readonly-0	412m	728Mi

site-2

slapd-0	437m	745Mi
slapd-1	443m	745Mi
slapd-2	**1696m**	747Mi
slapd-readonly-0	4m	724Mi

site-3

slapd-0	11m	743Mi
slapd-1	13m	743Mi
slapd-2	**735m**	747Mi
slapd-readonly-0	6m	727Mi

A directory nobody is using. Idle is the `4m` – `13m` on this slide — so every other number here is work that no client asked for.

Memory is flat everywhere, which is the tell: not load, not a leak — the same conversation being had over and over.

ITS#9398, end to end

Every logical database is two databases: the data DB, and `cn=accesslog—<db>` — a journal of recent writes. Both node-local, one per pod.

Two numbers on the journal

`contextCSN` — the newest change held, per serverID.
`minCSN` — the oldest still held.

Together they bound the window the journal can answer from.

They are not persisted the same way

`minCSN` is written to disk eagerly.
`contextCSN` only at a syncprov checkpoint — and no delta-syncrepl recipe anywhere puts a checkpoint on a log DB.

`logpurge` sweeps the journal daily. It advances `minCSN` on disk and leaves `contextCSN` behind — **and it deletes exactly the entries slapd would rescan at startup to rebuild one.** The safety net upstream relies on is removed by the only database that gets pruned on purpose.

> After an unclean stop the provider comes back with `minCSN` ahead of `contextCSN`, and answers every consumer, on every connection, *"your cookie is stale — re-read everything"*. There is no backoff on that path.

Measured on the lab mesh: **6968** stale answers per minute, five consumers in permanent full refresh, one provider at **2380m** CPU serving all of them. Every full refresh succeeds — it costs resources, not correctness.

Known, and not yet routine

In memory, `contextCSN` leads `minCSN`. On disk, `minCSN` leads `contextCSN`. **Health checks read over LDAP — the in-memory view — which looks perfect.**

The signal is on disk, and tools read the wire

The shipped checkers — LTB's, `check_syncrepl_extended`, the Prometheus exporters — read the *data* suffix. `slapd-monitor(5)` mentions replication zero times. `minCSN` is read by nothing we found.

`syncmonitor` is a passive cookie observer by design, and a consumer shrugging at `4096` is legitimate behaviour — it was not built to see this.

The bug is documented; the handling is not

The ITS has been open since 2020 and its intended fix depends on an enhancement that is parked. There is no published recovery procedure for a provider in this state.

No delta-syncrepl guide or upstream test fixture puts a checkpoint on a log DB — and no Kubernetes chart we surveyed sets a grace period or one either. Ours did not, before this.

> That is a gap, not a grievance. A maintainer wrote in 2023 that both `contextCSN`s *"should be monitored as such"* — we had implemented exactly the half that advice calls insufficient, and we found out by measuring rather than by reading.

Two failure modes, two cures

mode A — a missing serverID

`err=4096`, the storm. A serverID present in `minCSN` and absent from `contextCSN` altogether.

Cured only from **the pod that owns that serverID** — usually another site.

mode B — own-serverID regression

`err=53`, ~30/min, no measurable CPU. The pod's own `contextCSN` went backwards across the restart.

Cured **locally**, on the pod itself. Only pod P ever mints CSNs stamped P — so no peer should ever hold a newer one than P does.

The cure in both cases is one trivial write — issued at the serverID that is missing, because a write is stamped where it originates. Measured: on a live incident three writes took 6968/min to **0** and CPU recovered mesh-wide; against an armed fault, one write took `err=53` from ~30/min to **0**.

Every recovery procedure that exists assumes a human with a shell and the authority to stop a node. On Kubernetes that human is not there — but a controller is.

Avoid, diagnose, heal

Where the research actually stands. Nothing here ships today.

Avoid

`olcSpCheckpoint` on every `cn=accesslog—<db>` — the only candidate that *prevents* rather than notices, and the one that departs furthest from every published recipe.

Grace periods and `preStop` only cover stops that go through the pod lifecycle. An OOM kill or a node crash does not.

Diagnose — criteria identified

1 — the serverIDs in `minCSN` minus those in `contextCSN`, on the journal's root entry. Non-empty is mode A.

2 — no peer may hold a newer CSN for a pod's *own* serverID than that pod does. Mode B, and it needs no disk read at all.

3 — `slapcat` against `ldapsearch` on the log DB: the only one that sees exposure *before* a restart.

Heal — candidates

A **targeted write** at the serverID that is missing. The most promising by a distance, and the only one measured red-to-green twice.

For mode B that write is purely local. For mode A it has to originate at another site, which is a question about what a controller may reach, not about whether the cure works.

Heavier fallback: rebuild the journal from a converged peer.

Verification is in progress on the criteria and the path is not chosen. If you have run delta-syncrepl through unclean stops at scale, I would rather hear it than guess — input very welcome.

One e2e run after cutting 0.3.0

Three-site mesh, OpenLDAP 2.7.1

```
% make e2e

------------------------------

data loss recovery via replication
no 'sync cookie is stale' storm follows the recovery (ITS#9580, ADR-021)
tests/e2e/dataloss_recovery_test.go:227
  "msg"="stale-cookie occurrences since the data-loss event"
  "perPod"={"slapd-0"=4075 "slapd-1"=0 "slapd-2"=0} "total"=4075
  [FAILED] 4075 lines since the PVC-loss recovery (limit 50).
  Expected <int>: 4075 to be < <int>: 50

------------------------------

Ran 98 of 100 Specs in 978.855 seconds
FAIL! -- 97 Passed | 1 Failed | 0 Pending | 2 Skipped
```

First e2e test run after cutting the "LDAPCon release" `v0.3.0`. All tests green but one — the ITS#9398 / ITS#9580 upstream issue we don't yet know how to avoid or handle.

> There is a (partial) fix in upstream 2.7; evidently we're hitting the "partial" here.