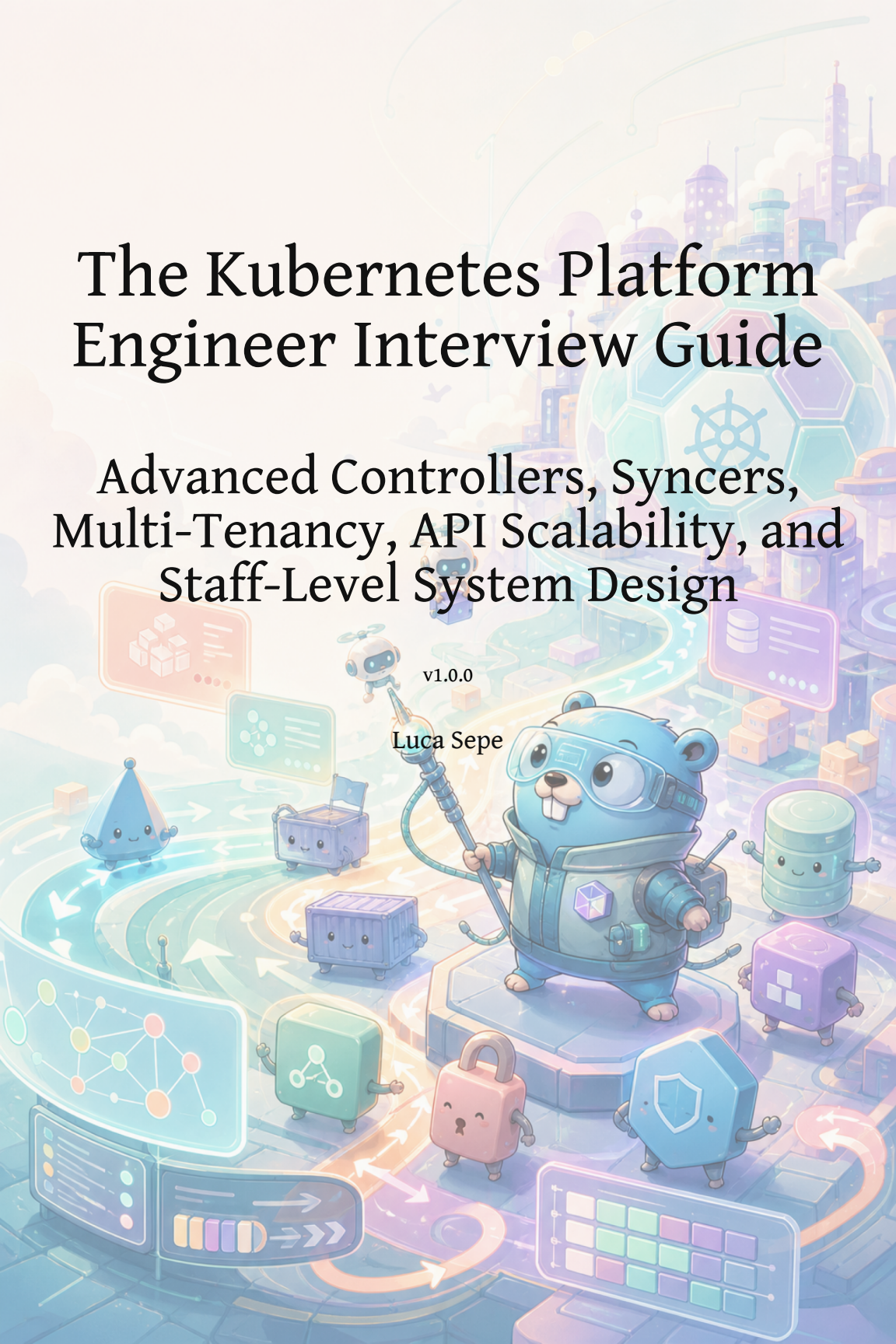
The Kubernetes Platform Engineer Interview Guide

Advanced Controllers, Syncers, Multi-Tenancy, API Scalability, and Staff-Level System Design

v1.0.0

Luca Sepe



Index

The Kubernetes Platform Engineer Interview Guide

How to Use This Book

This book is a practical interview guide for engineers preparing for senior, staff, and senior staff roles in Kubernetes platform engineering.

It focuses on the kind of reasoning expected in advanced interviews:

- controller correctness
- informer and cache mechanics
- optimistic concurrency
- leader election and lifecycle behavior
- finalizers and deletion safety
- API server scalability
- multi-tenant platform design
- syncer architecture
- production debugging
- Staff-level communication

The goal is not to memorize phrases. The goal is to develop a reliable way to explain Kubernetes systems under real constraints.

What a Strong Answer Sounds Like

A strong answer usually follows this shape:

1. Clarify the requirement.
2. State the mental model.
3. Describe the simplest correct design.
4. Identify failure modes.
5. Explain observability and testing.
6. Discuss how the design changes at scale.

For example, if asked how to design a controller, do not begin by listing libraries. Begin with the invariant:

> A controller should converge current state toward desired state. Events are only hints, so the reconcile loop must be idempotent and correct even if it runs multiple times or misses an intermediate event.

That answer shows that you understand the control-plane model, not just the framework.

Chapter Structure

Most chapters use this format:

- **Mental model**: the core concept in plain language.
- **Interview questions**: realistic prompts.
- **Strong answer**: a concise answer you can use aloud.
- **Deepening notes**: Staff-level trade-offs and failure modes.
- **Commented snippets**: small examples that show the idea without hiding it behind framework magic.

The snippets are intentionally small. In interviews, short code that exposes the invariant is more useful than a full production controller.

Study Plan

First pass: read Chapters 02 through 06 to build the controller and API machinery foundation.

Second pass: read Chapters 07 through 10 to connect lifecycle, scalability, syncers, and multi-tenancy.

Third pass: read Chapters 11 and 12 aloud. These chapters are designed for production debugging and mock interviews.

Fourth pass: build the capstone in Chapters 13 through 21. Run it locally, deploy it into kind, break it on purpose, and practice explaining the design as if you were in a Staff-level system design interview.

Final pass: use Appendices A through C as rehearsal material. Practice the drill bank, defend the capstone as if you were in a design review, and run the pre-interview checklist without notes.

For every question, practice answering in two to five minutes. Then ask yourself:

- What invariant am I protecting?
- What can fail?
- Who owns each field or resource?
- What happens during restart or rollout?
- What metric would prove the system is healthy?

For the capstone, add three more questions:

- What does the API contract promise?
- What fields does the controller own?
- What happens if the controller dies halfway through reconciliation?

What Staff-Level Means Here

Staff-level Kubernetes engineering is not only knowing APIs. It is the ability to design systems that remain understandable under scale, concurrency, partial failure, and organizational pressure.

A senior engineer can implement a controller.

A Staff Engineer can explain whether the controller should exist, what it should own, how it will fail, how teams will operate it, and how to keep it from becoming a control-plane liability.

Interview Principle

When uncertain, be explicit.

It is better to say:

> I would first define which side owns status, which side owns spec, and what freshness guarantees are required.

than to pretend that Kubernetes gives exact, immediate consistency everywhere.

Advanced interviews reward engineers who can make constraints visible.

Controller Runtime Internals

Mental Model

`controller-runtime` is a framework for building Kubernetes controllers. It provides common building blocks such as managers, caches, clients, controllers, reconcilers, health checks, metrics, and leader election.

It reduces boilerplate, but it does not remove the hard parts:

- eventual consistency
- stale cache reads
- update conflicts
- idempotency
- API pressure
- lifecycle correctness

In an interview, you should show that you know both the framework and the underlying control-plane behavior.

Question

What are the main components of a controller-runtime controller?

Strong Answer

The main components are: `Manager`, `Scheme`, `Cache`, `Client`, `Controller`, `Reconciler`, event sources, event handlers, predicates and work queues.

The manager owns shared process-level infrastructure. The controller watches resources and enqueues reconcile requests. The reconciler processes those requests and performs convergence.

Manager

The manager is responsible for shared lifecycle and infrastructure.

It typically: initializes caches, creates clients, starts controllers, manages leader election, exposes metrics, exposes health and readiness probes, handles graceful shutdown.

For a binary with multiple controllers, the manager prevents each controller from creating duplicated caches and lifecycle code.

Commented Snippet: Manager Setup

```
func main() {
    scheme := runtime.NewScheme()
    _ = clientgoscheme.AddToScheme(scheme)
    _ = platformv1.AddToScheme(scheme)

    mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme: scheme,

        // Leader election allows multiple replicas of the controller
        // manager while keeping only one active mutating leader.
        LeaderElection:   true,
        LeaderElectionID: "platform-controller.example.io",

        // Metrics and health endpoints are part of
        // operability, not decoration.
        Metrics: metricsserver.Options{
            BindAddress: ":8080",
        },
        HealthProbeBindAddress: ":8081",
    })
    if err != nil {
        os.Exit(1)
    }

    if err := (&WorkspaceReconciler{
        Client: mgr.GetClient(),
        Scheme: mgr.GetScheme(),
    }).SetupWithManager(mgr); err != nil {
        os.Exit(1)
    }

    if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
        os.Exit(1)
    }
}
```

The setup is not only wiring. It defines lifecycle, leader election, cache behavior, and operational endpoints.

Question

What is the difference between the cached client and the API reader?

Strong Answer

The default controller-runtime client usually reads through the cache and writes directly to the API server.

Cached reads are fast and reduce API server load, but they may be stale because the cache is updated asynchronously by watches.

The API reader bypasses the cache and reads directly from the API server. I would use it only when I need stronger freshness at a specific point, because direct reads increase API server load.

Commented Snippet: Cached Read vs Direct Read

```
func (r *Reconciler) compareReads(ctx context.Context, key client.ObjectKey) error {
    cached := &appsv1.Deployment{}
    if err := r.Client.Get(ctx, key, cached); err != nil {
        return err
    }

    fresh := &appsv1.Deployment{}
    if err := r.APIReader.Get(ctx, key, fresh); err != nil {
        return err
    }

    // The objects may differ briefly after a write. The cache catches up
    // through the watch stream, not through synchronous write-through behavior.
    _ = cached.ResourceVersion
    _ = fresh.ResourceVersion

    return nil
}
```

The Staff-level point is to avoid using direct reads everywhere as a workaround. It can hide a design issue and create control-plane pressure.

Question

What is a common mistake with controller-runtime?

Strong Answer

A common mistake is assuming that a cached `Get` immediately after an `Update` will return the updated object.

The write reaches the API server. The cache observes the change later through the watch stream. There can be a delay.

The controller should tolerate this by designing reconciliation around eventual consistency or by using a direct read only where exact freshness is required.

Predicates

Predicates filter events before they are enqueued.

They are useful for reducing noise, but dangerous if they filter out changes required for correctness.

Commented Snippet: Generation Predicate

```
func (r *WorkspaceReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&platformv1.Workspace{}).
        // GenerationChangedPredicate ignores updates
        // that do not change spec.
        // This is useful when the controller only needs
        // to react to desired state changes.
        // It would be wrong if status or metadata changes
        // must also trigger reconciliation.
        WithEventFilter(predicate.GenerationChangedPredicate{}).
        Complete(r)
}
```

In an interview, always explain what correctness signal you might be dropping.

Owns and Child Resources

`Owns` watches child resources and enqueues reconcile requests for the owner.

For example, if a `Workspace` owns a `Deployment`, changes to the `Deployment` can cause the `Workspace` to reconcile.

```
func (r *WorkspaceReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&platformv1.Workspace{}).
        Owns(&appsv1.Deployment{}).
        Complete(r)
}
```

This is useful because the primary desired state often lives in one object, while actual state is represented by several children.

Staff-Level Follow-Up

At Staff level, discuss operational consequences:

- cache memory footprint
- number of watched resource types
- namespace scoping
- watch reconnect behavior
- reconcile latency
- queue depth
- leader election during rollout
- API request rate

The question is not only whether the framework can build the controller. The question is whether the resulting binary can run safely in a large cluster.

Generic Kubernetes Syncer Architecture

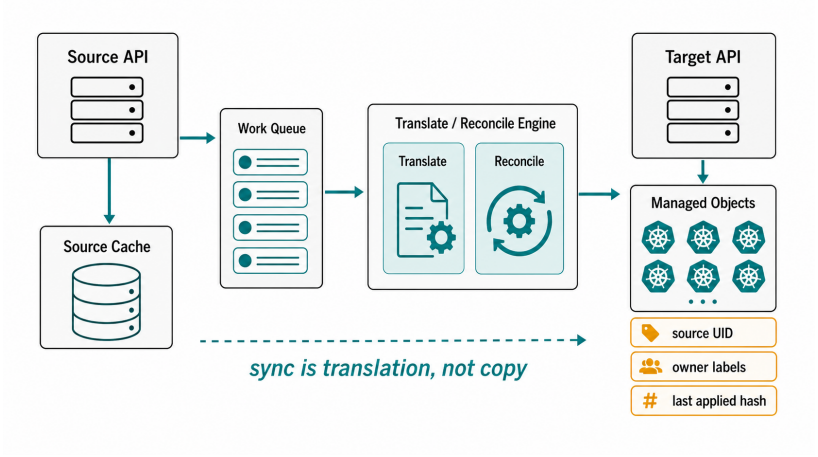
Mental Model

A syncer propagates selected state between two Kubernetes API servers or control planes.

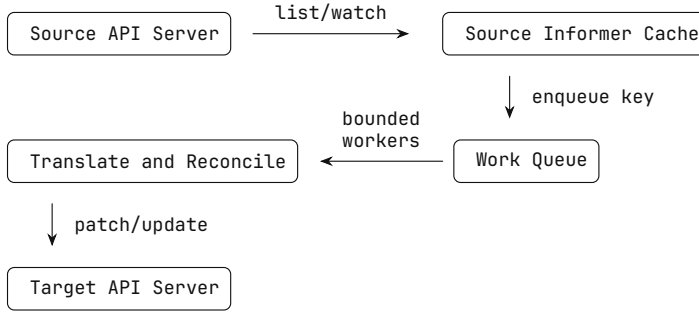
Examples:

- virtual cluster to host cluster
- management cluster to workload cluster
- central cluster to edge cluster
- tenant control plane to shared infrastructure

A syncer is not a YAML copy loop. It translates state between environments with different ownership, security, runtime, and scalability constraints.



High-Level Architecture



For bidirectional sync, each direction needs clear ownership and loop prevention.

Question

What are the hardest parts of designing a Kubernetes syncer?

Strong Answer

The hard parts are defining ownership, preventing loops, handling conflicts, mapping identity, preserving security boundaries, tolerating eventual consistency, scaling watches and queues, and debugging rare race conditions.

The core problem is not copying objects. The core problem is preserving semantics across two control planes.

Identity Mapping

A syncer needs a deterministic mapping between source and target objects.

Example:

```
source: tenant-a/default/nginx
target: host namespace tenant-a-default, object nginx
```

The mapping should be stable and encoded in metadata.

Commented YAML: Source Metadata on Target Object

```
metadata:
  labels:
    sync.example.io/source-namespace: default
    sync.example.io/source-name: nginx
  annotations:
    # UID distinguishes a recreated object from an
    # older object with the same name.
    sync.example.io/source-uid: "8f6a4f6d-5f2b-4b3b-9b19-111111111111"
    sync.example.io/source-cluster: "tenant-a"
```

Name alone is often not enough.

Translation

Translation decides which fields move from source to target and which fields are rewritten or ignored.

Commented Snippet: Translate a ConfigMap

```
func translateConfigMap(
    src *corev1.ConfigMap,
    targetNamespace string) *corev1.ConfigMap {

    return &corev1.ConfigMap{
        ObjectMeta: metav1.ObjectMeta{
            Namespace: targetNamespace,
            Name:      src.Name,
            Labels: map[string]string{
                "sync.example.io/source-namespace": src.Namespace,
                "sync.example.io/source-name":      src.Name,
            },
            Annotations: map[string]string{
                "sync.example.io/source-uid": string(src.UID),
                "sync.example.io/managed-by": "generic-syncer",
            },
        },
        // Data is safe to copy for this simplified example.
        // Other resources, such as Pods or Services, require stricter translation.
        Data:       maps.Clone(src.Data),
        BinaryData: maps.Clone(src.BinaryData),
    }
}
```

For real systems, translation should be resource-specific and explicit.

Loop Prevention

Bidirectional sync can create loops:

```
Cluster A -> Cluster B -> Cluster A -> Cluster B
```

The syncer needs origin metadata and field ownership rules.

Question

How do you prevent synchronization loops?

Strong Answer

I would mark objects with origin and ownership metadata, then define which side owns which fields.

For simple cases, annotations may be enough. For complex multi-writer systems, I would use field ownership, last-applied hashes, generation tracking, and explicit source-of-truth rules.

The key is that the syncer must know whether an event represents user intent or a change produced by the syncer itself.

Commented Snippet: Last Applied Hash

```
func desiredHash(obj runtime.Object) string {
    // In production, use stable serialization
    // and include only fields owned by the syncer.
    // Do not hash runtime fields such as resourceVersion.
    b, _ := json.Marshal(obj)
    sum := sha256.Sum256(b)

    return hex.EncodeToString(sum[:])
}

func shouldApply(target metav1.Object, desired runtime.Object) bool {
    last := target.GetAnnotations()["sync.example.io/last-applied-hash"]
    return last != desiredHash(desired)
}
```

Hashing the owned desired state can reduce unnecessary writes and avoid reacting to fields the syncer does not own.

Conflict Policy

Before implementing conflict handling, define the policy.

Possible policies:

- source always wins
- target owns selected fields
- field-level ownership
- reject conflicting updates
- surface conflict status to users

Without a policy, the syncer can oscillate.

Difficult Resources

Some Kubernetes resources are hard to sync because their semantics depend on the environment.

Pods: node name, pod IP, service account token projection, status and scheduler decisions.

Services: cluster IP, load balancer state and node ports.

PersistentVolumeClaims: storage class availability, binding behavior and topology.

RBAC: privilege escalation risk and tenant boundary impact.

CRDs: schema compatibility, version conversion and validation behavior.

Deletion Handling

Deletion must account for finalizers, tombstones, and orphan prevention.

Commented Snippet: Delete Target Idempotently

```
func deleteTarget(
    ctx context.Context,
    c client.Client,
    obj client.Object) error {

    err := c.Delete(ctx, obj)
    if apierrors.IsNotFound(err) {
        // Idempotent deletion: already gone means
        // desired cleanup is complete.
        return nil
    }

    return err
}
```

Cleanup paths should treat ”already gone” as success unless there is a security or consistency reason not to.

Failure Scenario: Target API Server Is Down

The source cluster is healthy, but the target API server is unavailable.

A good syncer should:

- keep source watches running if possible
- retry target writes with backoff
- avoid unbounded queue growth
- expose target write failures
- recover by re-reading source state

Correctness should come from source-of-truth reconciliation, not from an in-memory backlog alone.

Failure Scenario: Object Recreated With Same Name

A source object is deleted and recreated quickly with the same namespace and name.

The syncer should use UID-aware metadata to distinguish identity.

If the target still points to the old UID, the syncer may need to delete and recreate it or update ownership metadata according to resource semantics.

Staff-Level Design Checklist

In a design interview, drive the discussion with these questions:

- What is the source of truth?
- Which fields are synced?
- Which fields are intentionally not synced?
- What is the identity mapping?
- How are loops prevented?
- What is the conflict policy?
- What is the deletion strategy?
- What happens during partial outages?
- How is sync correctness measured?
- What are the scalability limits?

Capstone Architecture: Workspace Syncer

Goal

The capstone project is a small, production-oriented Kubernetes controller called `workspace-syncer`.

The companion source code is published at:

```
github.com/lucasepe/workspace-syncer
```

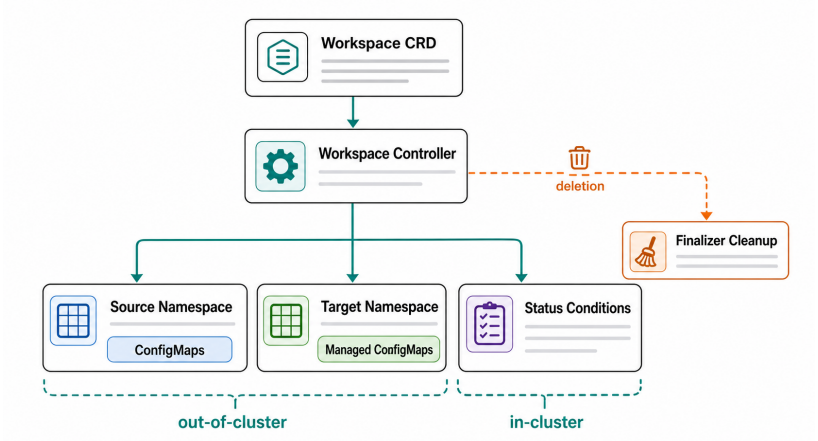
All file paths in the capstone chapters are relative to the root of that repository.

It implements a multi-tenant platform primitive:

> A `Workspace` declares a source namespace and a managed target namespace. The controller synchronizes selected safe resources from source to target.

The first supported resource is `ConfigMap`.

This is intentionally narrow. A useful capstone should be small enough to finish and rich enough to expose real platform engineering trade-offs.



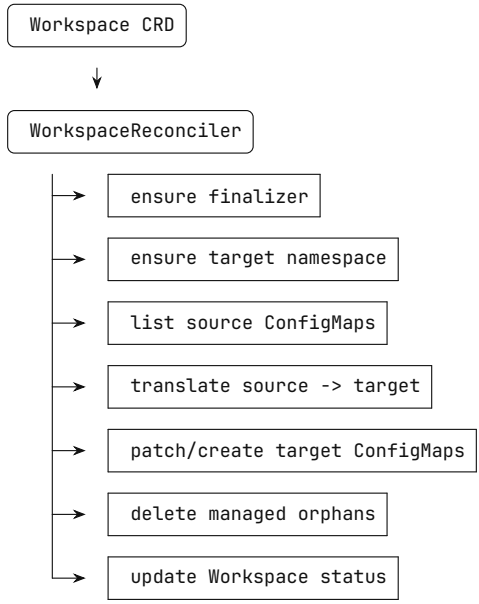
Why This Project

`workspace-syncer` exercises the same concepts that appear in advanced Kubernetes platform interviews:

- CRD API design
- spec and status ownership
- controller-runtime wiring
- reconcile idempotency
- informer watches
- ConfigMap translation
- source-to-target identity mapping
- sync loop prevention
- finalizer cleanup
- status conditions
- RBAC
- local and in-cluster execution
- Docker image build
- kind deployment
- failure exercises

It is not a toy CRUD controller. It is a compact teaching controller with real operational concerns.

Architecture



The source namespace represents tenant-owned input.

The target namespace represents platform-managed output.

Source of Truth

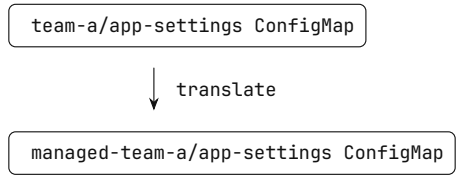
The `Workspace` object is the source of truth for sync configuration.

The source namespace is the source of truth for ConfigMap data.

The target namespace is managed output.

This distinction matters. If a user edits a managed target ConfigMap manually, the controller should eventually restore the source-derived state.

Resource Flow



The target object receives labels and annotations that describe ownership and source identity.

```
metadata:
  labels:
    workspace-syncer.example.io/managed-by: workspace-syncer
    workspace-syncer.example.io/workspace: team-a
    workspace-syncer.example.io/source-namespace: team-a
    workspace-syncer.example.io/source-name: app-settings
  annotations:
    workspace-syncer.example.io/source-uid: ...
    workspace-syncer.example.io/last-applied-hash: ...
```

The hash lets the controller skip no-op writes.

The UID distinguishes an object recreated with the same name.

Watch Strategy

The controller watches:

- `Workspace` objects
- target `Namespace` objects it owns
- source `ConfigMap` changes

When a source ConfigMap changes, the controller maps that event back to every `Workspace` whose `spec.sourceNamespace` matches the ConfigMap namespace.

That is the interview-relevant pattern:

> Watch what changed, enqueue what owns the reconciliation.

Lifecycle

Creation:

1. User creates `Workspace`.
2. Controller adds finalizer.
3. Controller creates target namespace.
4. Controller syncs ConfigMaps.
5. Controller updates status.

Update:

1. User changes source ConfigMap or Workspace spec.
2. Controller reconciles current state.
3. Controller applies only necessary target changes.

Deletion:

1. User deletes `Workspace`.
2. API server sets deletion timestamp.
3. Controller deletes managed ConfigMaps.
4. Controller removes finalizer.
5. API server completes deletion.

What This Capstone Does Not Do

The first version intentionally does not sync: Secrets, RBAC, Services, Pods, PVCs and CRDs.

That omission is a feature. It gives the reader space to discuss why those resources are harder.

Staff-Level Framing

In an interview, present the capstone like this:

> I built a small multi-tenant syncer that exposes the important controller design problems without hiding them behind scaffolding. The API defines source and target ownership. The controller uses idempotent reconciliation, finalizers for cleanup, metadata for identity, patches to reduce writes, status conditions for user visibility, and kind deployment to prove it runs in cluster.

That is much stronger than saying "I wrote an operator."

Capstone Failure Exercises

Why Break the Capstone

Failure exercises turn the capstone into interview preparation.

They help the reader practice explaining:

- what failed
- which invariant was protected
- how the controller recovered
- what metric or status signal revealed the issue

Exercise 1: Edit the Target Manually

Apply examples, then edit the target ConfigMap:

```
kubectl patch configmap app-settings \
  -n managed-team-a \
  --type merge \
  -p '{"data":{"LOG_LEVEL":"trace"}}'
```

Expected behavior:

- the target may temporarily differ
- the next reconcile restores source-derived data

Interview explanation:

> The target namespace is managed output. Source wins for ConfigMap data. Manual target edits are not treated as desired state.

Exercise 2: Delete the Target Namespace

```
kubectl delete namespace managed-team-a
```

Expected behavior:

- the controller recreates the target namespace
- synced ConfigMaps return after namespace creation completes

Discussion:

- namespace deletion can take time
- recreating too aggressively may hit API errors during terminating state
- production code may need clearer status during namespace termination

Exercise 3: Exceed the ConfigMap Limit

Create many ConfigMaps in the source namespace.

```
for i in $(seq 1 60); do
  kubectl create configmap "cm-$i" -n team-a --from-literal=value="$i"
done
```

Expected behavior:

- Workspace becomes not ready
- status reason reports `SyncFailed`

Discussion:

> Limits protect the platform from unbounded tenant-driven sync work.

Exercise 4: Delete the Workspace

```
kubectl delete workspace team-a
```

Expected behavior:

- deletion timestamp is set
- finalizer cleanup runs
- managed ConfigMaps are deleted
- finalizer is removed
- Workspace disappears

Discussion:

> Finalizers turn deletion into a reconciled lifecycle step.

Exercise 5: Break RBAC

Remove the controller's permission to delete ConfigMaps, then delete the Workspace.

Expected behavior:

- Workspace remains terminating
- logs show forbidden errors
- finalizer remains

Discussion:

> Stuck finalizers are often caused by RBAC, external dependency failure, or non-idempotent cleanup.

Exercise 6: Kill the Leader Mid-Reconcile

Run in kind, scale to two replicas, then delete the active Pod.

```
kubectl delete pod -n workspace-syncer-system -l app.kubernetes.io/name=workspace-syncer
```

Expected behavior:

- a new leader is elected
- reconciliation may repeat
- final state remains correct

Discussion:

> Leader election does not provide exactly-once processing. Idempotency still protects correctness.

Exercise 7: Delete and Recreate a Source ConfigMap

```
kubectl delete configmap app-settings -n team-a
kubectl create configmap app-settings -n team-a --from-literal=LOG_LEVEL=warn
```

Expected behavior:

- target converges to the new source
- source UID annotation changes

Discussion:

> Name alone is not identity. UID helps distinguish a recreated object from the previous object with the same name.

Exercise 8: Disable ConfigMap Sync

Patch the Workspace:

```
kubectl patch workspace team-a \
  --type merge \
  -p '{"spec":{"sync":{"configMaps":false}}}'
```

Expected behavior:

- managed target ConfigMaps are deleted
- Workspace remains ready if cleanup succeeds

Discussion:

> Disabling a sync class should converge managed output to the new desired state.

Staff-Level Debrief

For each exercise, answer:

- What was the desired state?
- What was the observed state?
- Which component owned the field or resource?
- Was the operation idempotent?
- Was the failure visible in status?
- Would this design scale to many tenants?

This is the fastest way to turn the capstone into interview fluency.