CLOUD NATIVE SUMMIT MUNICH 2026

Crossplane as Your Platform API: Decisions, Scars, and the v2 Upgrade

Stephan Hüttner & Philip Welz

white duck

Who we are

Philip Welz
Cloud Solutions Architect
Azure MVP

philip.welz@whiteduck.de

www.linkedin.com/in/philip-welz

Stephan Hüttner
Head of Platform Engineering | Cloud Solutions Architect

stephan.huettner@whiteduck.de

www.linkedin.com/in/stephan-huettner

Agenda

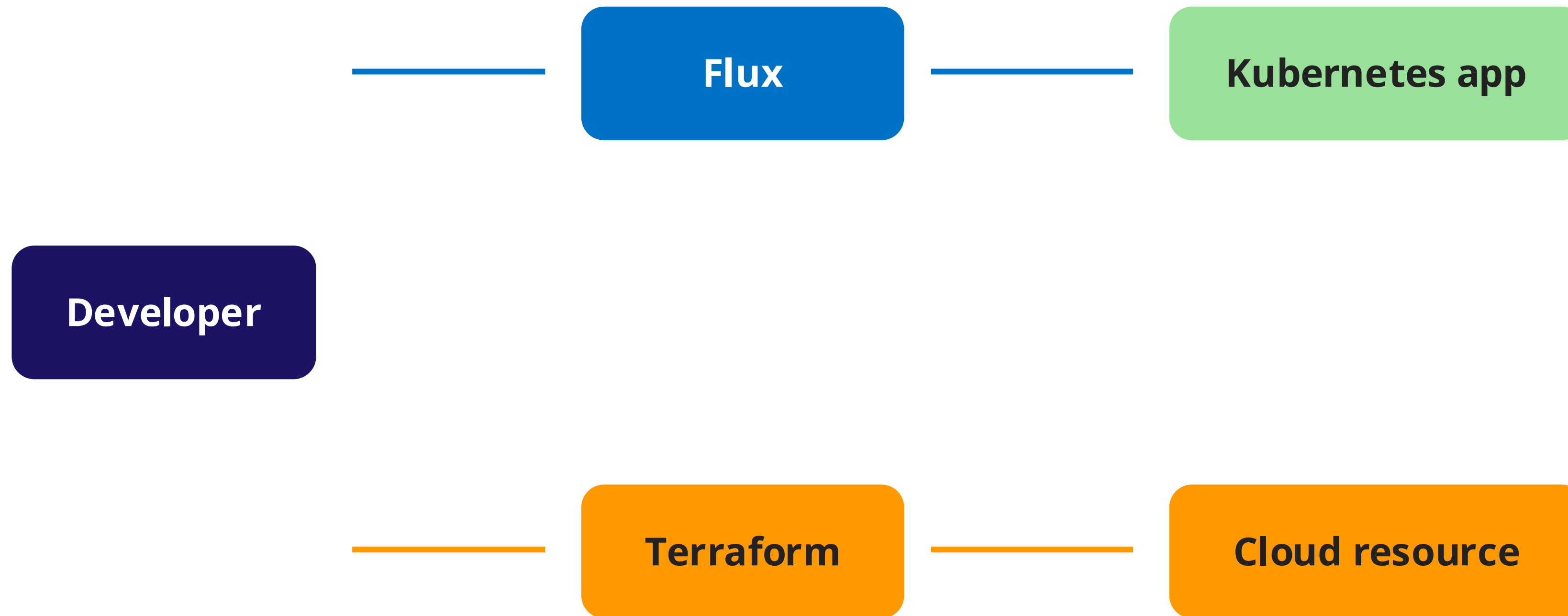
- The platform problem
- Decision path and GitOps architecture
- Control-plane pattern
- Composition scars and validation
- Observability
- Crossplane v2 migration
- Recommendations and Q&A

The platform problem

The platform problem was not "create a portal"

The requirement was repeatable automation for APIs, jobs, events, and infrastructure — without exposing every implementation detail to app teams.

First instinct: Flux for apps, Terraform for infrastructure



For this use case, per-use-case Terraform would have added state, credentials, modules, and review flows to every team.

Decision path

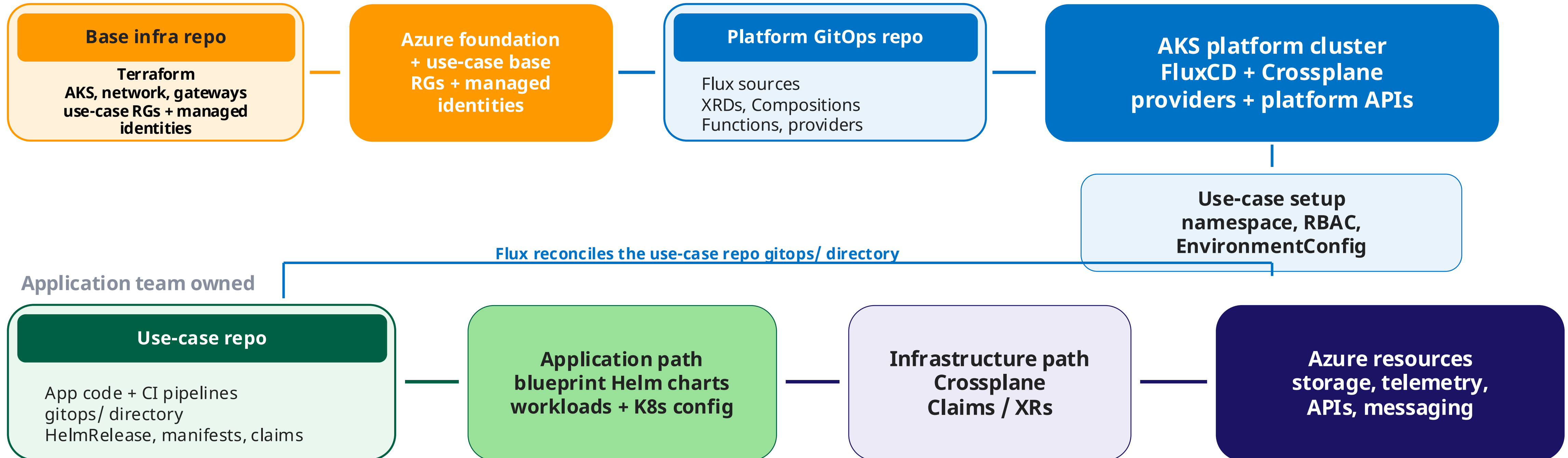
Why Crossplane

- Terraform remained the right tool for base infrastructure
- Cloud-specific operators managed resources, not the productized platform API
- Kro did not meet maturity requirements at project start

- Crossplane gave us providers, XRDs, Compositions, and GitOps fit
- Kubernetes became the shared API surface
- The trade-off moved into API design, validation, and function ownership

Architecture: GitOps-driven platform API

Terraform builds the shared foundation and use-case Azure base. Flux connects platform and use-case repos. Crossplane turns small claims into governed cloud resources.



Side path for API workloads: app registrations, app roles, and cross-tenant service principals across 3 Entra tenants are composed behind the same claims.

The control-plane pattern

Small claim, governed outcome

```yaml
apiVersion: platform.example.io/v1alpha1
kind: ApplicationInsights
metadata:
  name: my-use-case
spec:
  writeConnectionSecretToRef:
    name: app-insights
```

naming

identity

RBAC

policy

resources

No resource group. No subscription. No tenant IDs. No platform-specific wiring.

XRD design: the product API contract

An XRD defines what teams may ask for. The Composition defines how the platform fulfills it.

```
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
spec:
  group: platform.example.io
  names:
    kind: XTelemetry
    plural: xtelemetries
  claim:
    kind: Telemetry
    plural: telemetries
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema: ...
```

Required decisions

What the application team must choose: product-level options, outputs, and safe variability.

Guardrails fail early

OpenAPI schema, required fields, enums, defaults, and cross-field validation rules where useful.

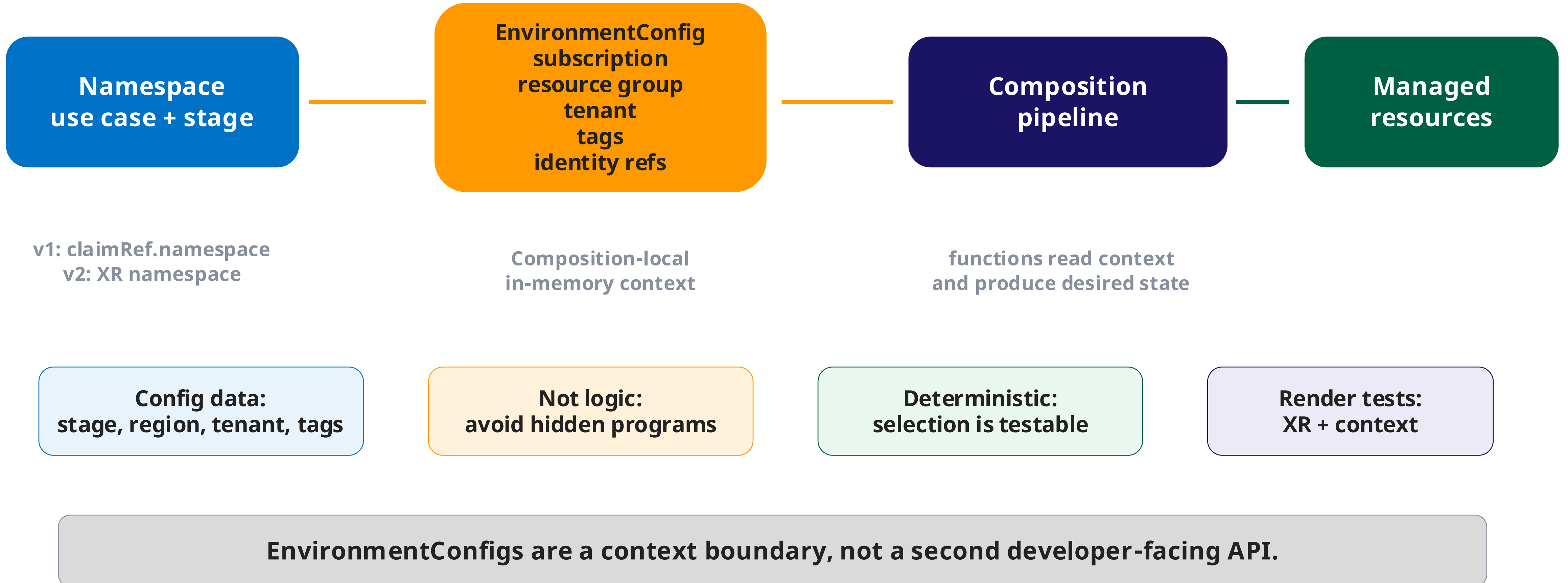
Hidden implementation

Resource groups, subscriptions, tenant IDs, provider fields, naming, policies, and RBAC stay platform-owned.

Rule of thumb: design XRDs as product APIs, not provider mirrors.

EnvironmentConfigs: context without input sprawl

The small claim stays context-free because platform context is injected during composition.



Composition evolution: use the simplest tool that fits

The complexity does not disappear. You choose where it lives.

The control-plane pattern, end to end

A small claim becomes governed managed resources: the namespace selects context, the Composition does the rest. In v2, the XR namespace is the context boundary.

Composition scars and validation

Scar 1: Put logic in Crossplane primitives

- XRD contract: OpenAPI schema, validation rules, defaults, claim/XR spec
- Composition pipeline: EnvironmentConfig, Function input, desired composed resources
- Function code: observed XR parsing, conditions/events, deterministic names

- Ask where each Crossplane rule should fail
- Ask who needs to understand the Function result
- **Use the earliest layer that gives useful feedback**

Scar 2: Custom functions are software products

- Deterministic output avoids reconcile noise
- Complex policies and identity relationships need code-level validation
- Unit tests become part of platform operations

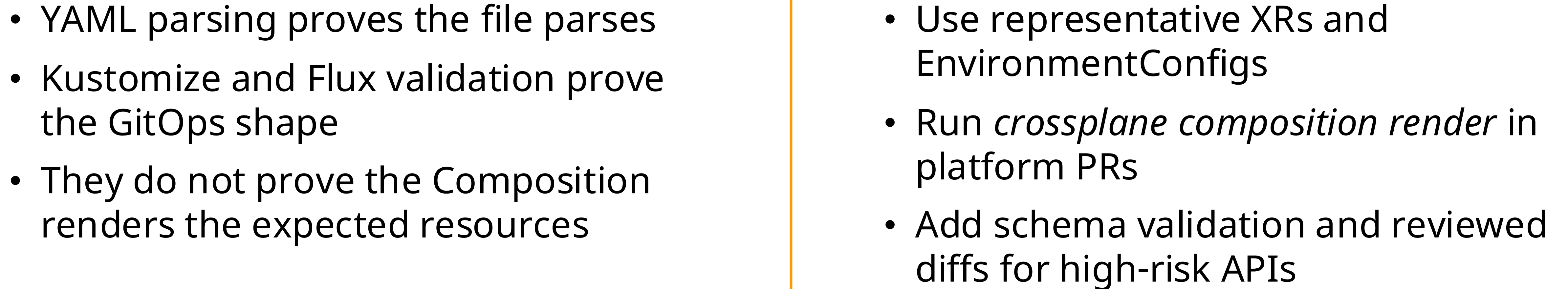
- Functions need CI, release management, dependency updates, and ownership
- **Worth it only when complexity is real and recurring**
- Do not hide business-critical logic in untested templates

Scar 3: Deletion is a platform API decision

- managementPolicies define allowed actions per managed resource
- Remove Delete for data-bearing resources
- Examples: storage, databases, and log workspaces

- Default behavior grants full control; provider support varies
- **Test create, update, delete, orphan, and adoption behavior**
- **Document removal semantics for application teams**

Validation gap: YAML validity is not platform validity



Recommended validation pipeline

Platform PRs

- XRD schema and validation-rule checks
- Render representative XRs with real functions
- Validate rendered output with *crossplane resource validate*

Use-case PRs

- YAML, Kustomize, Flux, and HelmRelease validation
- Platform resource schema validation against published XRDs
- Server-side dry-run or ephemeral control-plane cluster for high-risk changes

Observability

Observability: build it around Crossplane signals

- Conditions: Ready, Synced, Responsive
- Events first; Crossplane, provider, and Function logs next
- Trace the graph with *crossplane resource trace*

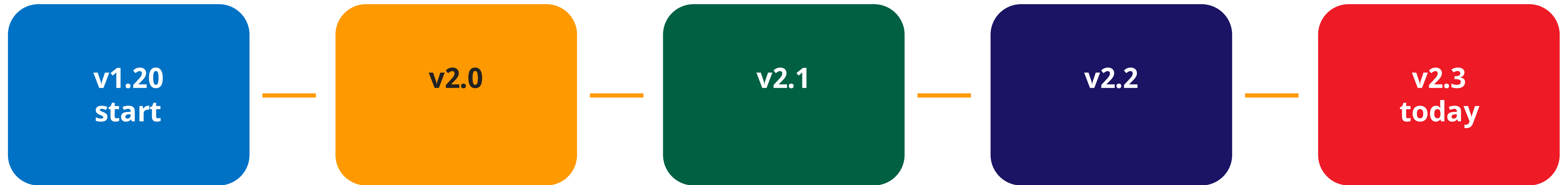
- Enable Prometheus metrics in the Crossplane Helm chart
- Alert on Function latency/errors and managed-resource readiness/sync/drift
- Use circuit-breaker metrics to spot reconciliation thrashing

Crossplane v2

Why v2 is not just an upgrade

Native v2 removes the claim abstraction. A true migration changes manifests, RBAC, validation, ownership, and rollback assumptions.

Phase 1: upgrade the control plane



Two migrations, not one. First the control plane: one minor at a time, latest patch, health-check after each step — the resource model stays unchanged.

Phase 2: migrate the resource model

Then the resource model: claims → namespaced XRs. One writer at a time — only one managed resource may write or delete a cloud resource. Verify cloud IDs and Secret hashes at every step.

What broke, what we shipped

- What broke / learnings
- Deleting a claim removes its connection Secret
- No push-button path — every step is a PR + reconcile + verify
- Observe-only adoption is slow, but reversible at each step

- What we shipped
- Crossplane v2.3.3 — no cloud resource recreated
- Claims gone — tenants use namespaced XRs
- Modern namespaced managed resources, XR Secrets app-facing

Recommendations

- Design XRDs as product APIs, not provider mirrors
- Keep environment knowledge out of developer input
- Make delete behavior explicit for data-bearing resources
- Treat `crossplane composition render` as platform API unit tests
- Build dashboards from conditions, Events, trace, and metrics
- Use custom functions only when the complexity pays for itself
- Treat v2 as a chance to simplify the platform contract

The goal is not "everything in Crossplane"

The goal is a reliable, testable platform API.

Small input

Strong contract

Tested output

CLOUD NATIVE SUMMIT MUNICH 2026

Thank You!