Overview

Infra-devops are some of the most busiest teams in most companies, they run multiple tools and still spend hours manually wiring DATABASE_URL into config files.
Juju is the only tool in the ecosystem that brings in the idea of service relationships, and this makes a significant difference.

This talk is in two halves: first, why Juju is genuinely compelling; second, what production with Juju actually looks like, including some case-studies of debugging production issues.

Part 1: Why Juju Changes the Game (10 min)

The gap in the standard stack

Most teams use:

- **Terraform** - provisions cloud resources (VMs, VPCs, EKS clusters)
- **Ansible** - configures software on those machines
- **Helm** - deploys workloads onto Kubernetes
- **Devs** - manually wire DATABASE_URL, handle failover, rotate credentials

That last bullet is the operational toil that is most painful, it is manual, error-prone, and it never ends.

What Juju does differently

Juju answers a different question: *how do I operate a distributed system, across any infra, without writing a separate runbook for every piece of it?*

Where Terraform, Ansible, and Helm each solve one part of the puzzle, Juju covers the full stack:

- Provision infra
- Configure software
- Deploy to K8s
- Auto-wire services
- Works on bare metal
- Works across clouds

The relation protocol: Juju's real superpower

When we run `juju relate webapp postgresql` :

- Postgres charm writes host, port, dbname, and credentials into a shared relation databag
- Webapp charm reads them and restarts with the correct DATABASE_URL
- If Postgres fails over, it updates the databag, webapp gets a `relation-changed` hook, and reconfigures itself automatically

No other tool in the stack has this. It is why Juju eliminates an entire category of day-2 operational toil.

Mental model for K8s engineers

- **Deployment** → Application
- **Pod** → Unit
- **Namespace** → Model
- **Helm chart** → Charm (with operational logic, not just YAML)
- **kubelet** → Juju unit agent
- ***(nothing)*** → Relation (Juju-only concept with no K8s equivalent)

Part 2: Production Reality (18 min)

The Canonical Sites stack (2 min)

In this section, we discuss the realities of running Juju in production, detailing the challenges we encountered and the key lessons we learned.

The Canonical Sites team runs ubuntu.com, canonical.com, and related web properties on Juju and Charmed Kubernetes. Flask/Jinja2 apps packaged as rocks, operated via Charms, deployed into Juju models on Charmed K8s. This is what day-to-day production looks like from the inside.

< A demo of how Juju in production looks like>

Case Study 1: Charm upgrades silently breaking relations (7 min)

The symptom: App is running. `juju status` shows `active/idle` . But an integration is silently failing with no error and no alert.

What happened: A charm upgrade changed the relation interface. The new charm version writes different keys to the relation databag. The consuming charm's `relation-changed` hook never fired, so it is still reading stale values.

Why it is hard to debug: `juju status` reports unit state, not relation data integrity. A unit can be `active` while its relation databag is stale or empty.

How to actually diagnose:

```
# Stream hook logs for a specific unit
juju debug-log --include unit:webapp/0 --replay

# Inspect actual relation data
juju run webapp/0 -- relation-get -r <rel-id> - postgresql

# Re-trigger the hook manually
juju exec --unit webapp/0 -- hooks/relation-changed
```

Fix patterns:

- `juju remove-relation` then re-add forces a fresh `relation-joined` on both sides
- `juju refresh <charm> --force` only when you fully understand the interface change

Takeaway: Never trust `active/idle` after a charm upgrade without manually verifying relation data.

Case Study 2: Model drift between staging and production (7 min)

The symptom: Works perfectly on staging. Breaks on production with no obvious diff.

What happened: Over three months, staging accumulated hotfixes applied via `juju config` CLI that were never committed back to the bundle. Production was running the committed bundle. Different charm revisions, different config values, different relation states.

Why it happens: Juju config changes via CLI do not sync back to bundle.yaml automatically. The bundle is not a live source of truth; it is a snapshot you have to maintain yourself.

How to detect it:

```
# Export current model state
juju export-bundle > current.yaml

# Diff against your committed bundle
diff bundle.yaml current.yaml
```

What `export-bundle` misses: secrets, manually-set config values, cross-model relations.

The discipline:

- Treat `bundle.yaml` as source of truth and make all config changes via PRs
- Run `juju diff-bundle` before every production deployment
- Pin charm revisions explicitly; never rely on `latest/stable` in production

Takeaway: Model drift is an ops culture problem as much as a tooling problem.

Debugging Toolkit (2 min)

For each common failure mode, here is where to start:

- **Unit stuck in `maintenance`:** `juju debug-log --include unit:app/0 --replay`
- **Relation data wrong or stale:** `juju run app/0 -- relation-get -r <rel-id> - app`
- **Hook silently failing:** `juju debug-hooks app/0`
- **Config drift between envs:** `juju export-bundle > current.yaml && diff bundle.yaml current.yaml`
- **Pod not starting:** `kubectl logs -n <model> <pod>` (bypass Juju entirely)

Closing (1 min)

Juju is operator knowledge encoded in code. The relation protocol solves a problem no other tool in the ecosystem addresses. But that power comes with a debugging model that is opaque if you do not know where to look.

The goal of this talk: give you the mental model and the commands to actually trust Juju in production, not just deploy it.