CASE STUDY 1 · PRODUCT DESIGN, RESEARCH & FRONT-END

KubeDB UI: moving database operations from the terminal to the UI

How I turned YAML and kubectl work for 25+ database engines into guided, consistent screens.

COMPANY	MY ROLE	USERS	STACK
AppsCode Ltd.	Only designer · owned UI product decisions · front-end	Platform engineers, DevOps, DBAs	Figma · Vue/Nuxt · design system

The problem

KubeDB runs production databases on Kubernetes. Before the UI, users created, scaled, backed up and restored databases by writing YAML and running kubectl. That works for experts, but it is easy to get a field wrong, hard to see the state of a database at a glance, and every engine (PostgreSQL, MySQL, MongoDB, Redis and 20+ more) behaved a little differently.

The result was a steady flow of **support issues about routine operations** (around 20–25 on database deploys alone) such as deploying, scaling and backing up databases, and a product that felt harder to adopt than it should.

My role

- Owned the problem end to end as the only designer: research, information architecture, interaction design, UI and front-end in Vue/Nuxt.
- Acted as **product owner** for the UI: set priorities with the CEO and the platform team, and decided what to ship first.
- Guided 5 front-end engineers on UI/UX standards and reviewed their work.

Research: understanding how people really work

LISTEN	REVIEW	AUDIT	HAND-OFF
Interviewed customers and our in-house developers about where database operations went wrong.	Went through recurring support questions to find the operations that caused the most confusion.	Ran a detailed UX review of the Scaling, Networking and Secret Management flows.	Turned findings into a Figma annotation checklist the engineers could act on.

What I learned: users did not need every YAML field. They needed to finish a few common jobs safely (create, scale, upgrade, back up, restore, monitor, fail over) and to see quickly whether a database was healthy. When something broke, the slow part was finding the right logs or shell, not fixing the issue.

Design decisions and the principles behind them

DECISION	PRINCIPLE	WHY IT MATTERS
Guided forms for Day-2 operations instead of YAML	Recognition over recall	Users pick from valid options instead of remembering field names, so fewer mistakes reach the cluster.
Few choices per step, sensible defaults, advanced options folded away	Hick's law · progressive disclosure	Common jobs stay fast; experts can still open the advanced settings when they need them.

One page structure for all 25+ engines	Consistency · Jakob's law	Learn it once, use it for every database. Engineers build new engines from the same pattern.
Status-first tables and a fleet view across 8 cloud providers	Visibility of system status	Health is the first thing users see, so problems stand out before they spread.
One-click path from a failing pod to its logs or shell	Reduce steps on the critical path	Cuts the slowest part of incident work: getting to the right place.
Clear confirmation and plain language for risky actions	Error prevention	Scaling, restore and failover are high-stakes; the UI makes the impact clear before the user commits.

Building it so it scales

A single good screen was not enough: 5 product consoles (Cluster, KubeDB, Platform, Billing, Self-host) needed to look and behave the same. I built and maintain the **AppsCode design system** they all share:

- **Three-tier tokens** (primitive → semantic → component) with theming per product and per light/dark mode.
- **Shared Vue components and one icon library**, so engineers reuse parts instead of rebuilding them and ship new screens faster.
- **Moved 7 repos from Bulma to Tailwind CSS 4** in one month with no behaviour change; builds became **up to 59% faster**.

Results

- Create, scale, monitor, back up, restore and fail over moved from the terminal into the UI.
- **Support issues about deploying databases dropped from ~20–25 to almost zero**, freeing the support and platform teams for harder problems.
- One consistent structure across **25+ database engines** and **5 consoles**, built on one design system.
- New screens ship faster because engineers start from shared, tested components.

> **What I would do next:** add simple product analytics to the key flows, measure task completion and drop-off for each operation, and run a short usability round on the restore and failover flows, where mistakes cost the most.

Case study 2 in short: one theme for 7 product websites

AppsCode had 6–7 product and docs sites that each drifted in layout and quality. I built **one shared Hugo theme**: common layouts and patterns live in the theme, and each product keeps its own colour. A design or bug fix now ships once to every site, and every site scores **100 on Lighthouse SEO**.

How I work

1 · UNDERSTAND
Talk to users and support, review issues, analyse how people use the product.

2 · DECIDE
Frame the problem, prioritize with the CEO and team, define what "done" means.

3 · DESIGN
IA, flows and interaction design in Figma, grounded in UX principles.

4 · BUILD
Ship production UI in Vue/Nuxt on the design system, then review and iterate.

More work and case studies: mohin.uxatom.com · linkedin.com/in/mohin7 · hello.mohin4@gmail.com