When two services become one



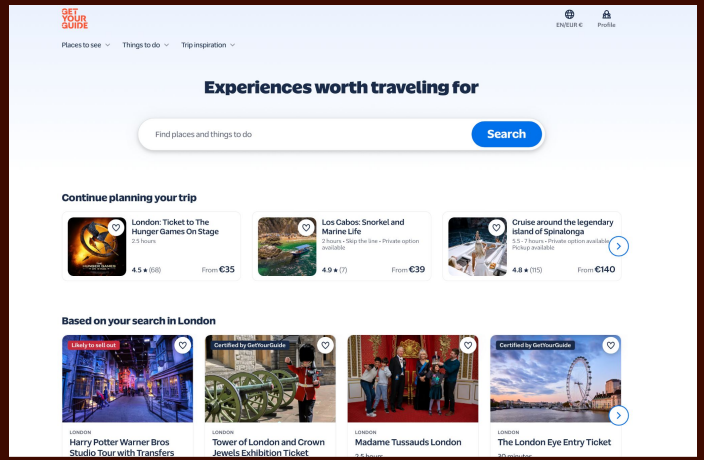
Ayush Kumar

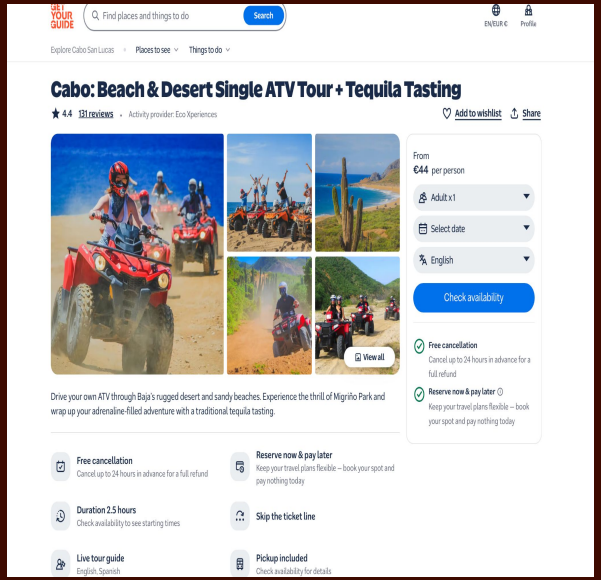
15 October 2026

What is GetYourGuide

Global marketplace for travel experiences — tours, activities, attraction tickets — connecting travelers with local suppliers.

- Founded 2009 · HQ Berlin
- 200,000+ bookable experiences, from 20,000+ supply partners, across 150+ countries
- 30M+ experiences booked yearly
- ~90,000–100,000 bookings/day

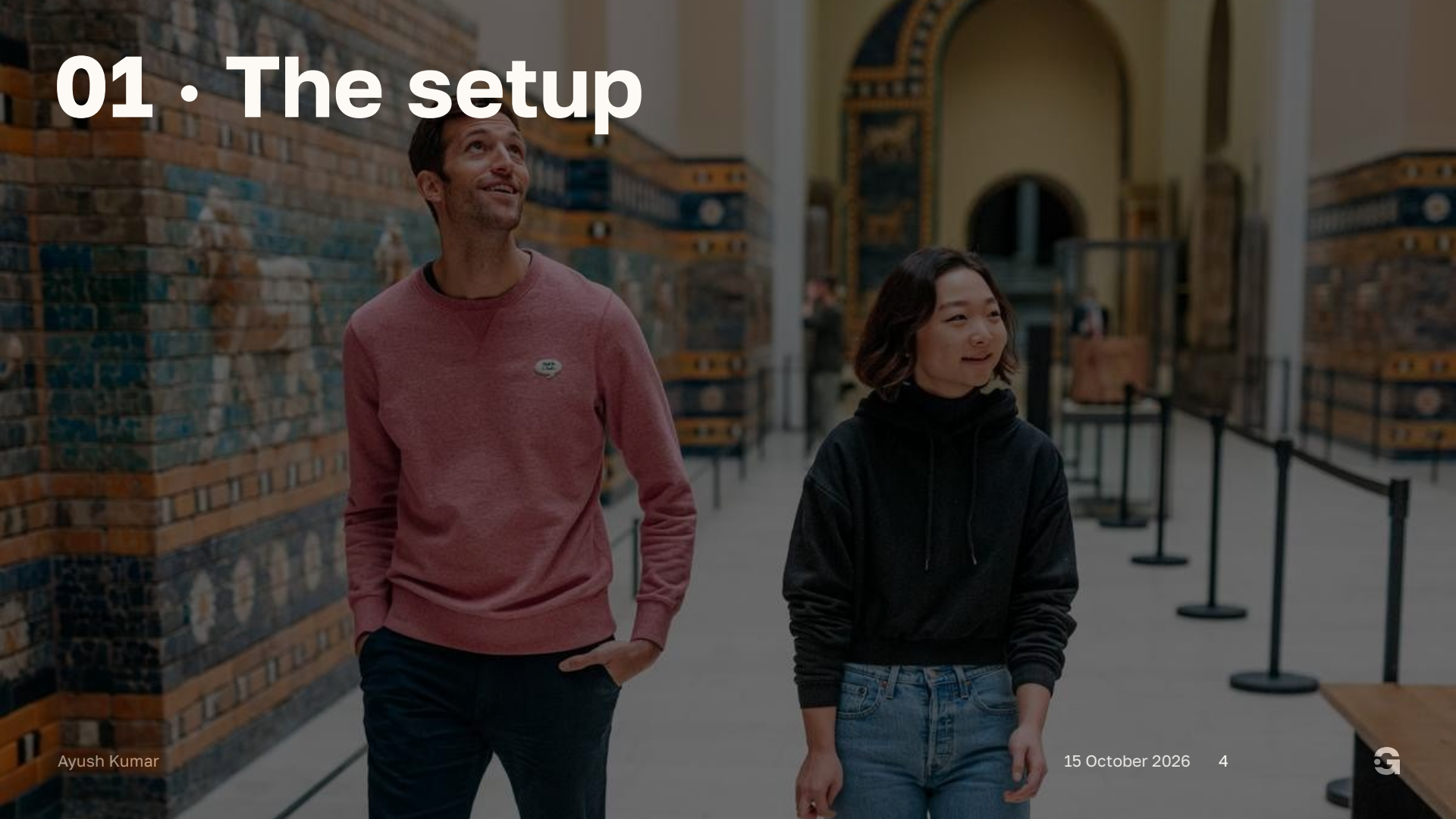




Agenda

1. **The setup**
   Two services, one domain
2. **Four signs**
   What it looks like in production
3. **Four fixes**
   The architectural changes that fixed it
4. **De-risking it**
   Shadow Rollout
5. **Takeaways**
   Things to take back home

01 · The setup



Two services. One responsibility.

Availability and pricing for 200,000 activities, split across two services. It looked logical at the time. Different technical requirements, different teams.



Multiple incidents, different root causes

We started having incidents every week, sometimes multiple in a week. Oncall getting paged every night



At some point, our oncall found calmness in this chaos

Incident, Fix, PostMortem, Repeat

We fixed that incident, moved on, then another incident

We joked we could reuse the same postmortem and just change the date

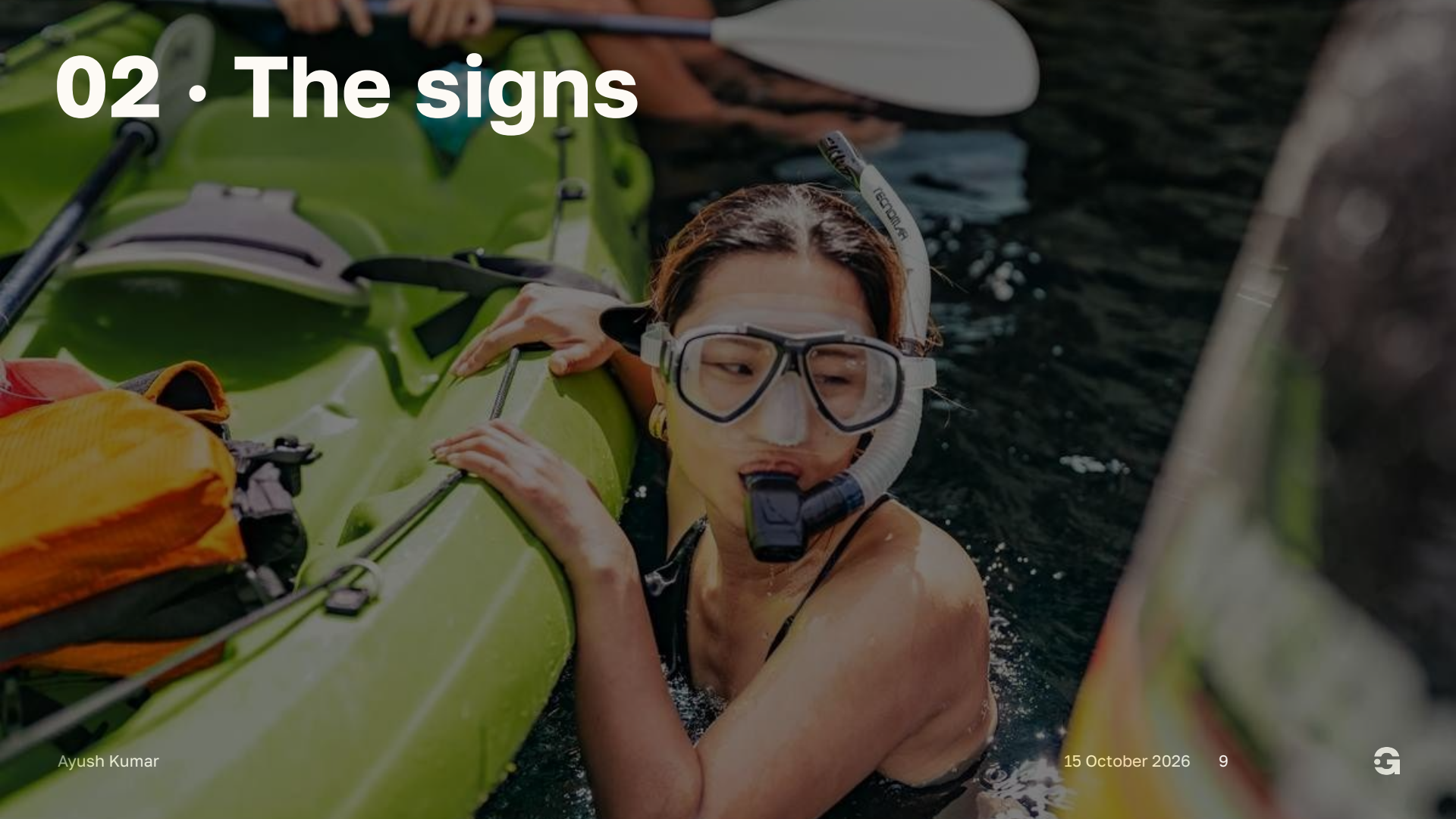
At some point, we decided to take a step back and analyse





Four weeks. No sacred rules.

02 · The signs



Four things we found

01

Duplication
Core data duplicated across five systems.

02

No single owner
Cached status flags drifted out of sync.

03

Bad split
Split by how, not by what the data meant.

04

Hidden logic
Triggers fired silently on every commit.



01 · Data duplication

The signs

Service A called Service B to validate, B called A, C, D to store data on every request. Three services shared the config, so one change meant multiple releases.

The cost

Recovery took longer, every new feature is cross team collaboration.

02 · no source of truth

isBookable : can a customer book this right now? Multiple services stored copies of different flags and each consumed in it’s own speed.



03 · the boundary was wrong

The boundary

We drew it around the integration mechanism, not the business capability.

The consequence

A single customer journey crossed two teams. Every incident needed both teams online.

04 · if you can't grep for it, it shouldn't run

An application trigger cascaded changes across 12,000 records with no application logs, no metrics, and no way to track the magic code.



03 · Four fixes



Four decisions, one direction

01

One owner

All availability and pricing logic moves into one service.

02

Single truth

isBookable is computed once. Only one service checks it

03

Sync reads

Services query on read, not from pushed copies.

04

Loud events

Triggers become named, versioned events.

01 · One service, one business capability



Before

Two services shared the domain. Both owned status flags and validation, and neither could ship alone.

After

Service A owns availability and pricing entirely. Service B becomes a stateless integration layer. That bought independence and no cross-team coordination on these changes.

02 · isBookable is one service ownership

Before

Six separate checks across five services, each with its own Kafka topic and consumer lag. A delay could make search show a product as available while checkout showed it sold out.

After

Service A computes and stores isBookable with every change, with no caching layer on the read path. That reduced staleness on checkout and made the field explainable again.

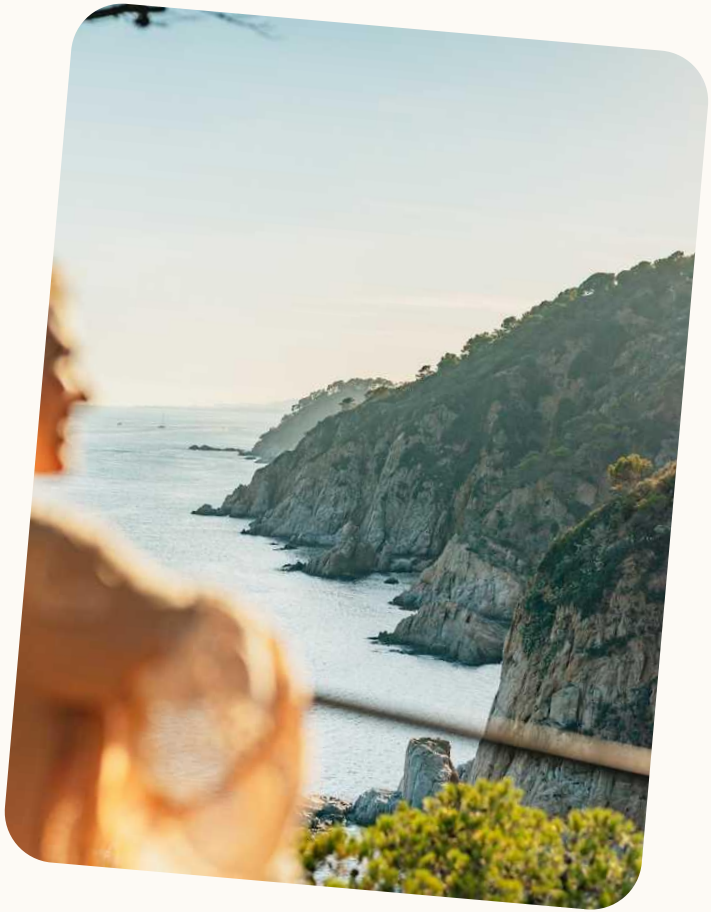
03 · ask the source, don't copy it

Before

Downstream services kept their own copies through Kafka consumers and their own db columns. MTTR reached 8 hours when the backlog was big.

After

Consumers query source directly. MTTR dropped to about 30 minutes, 16 times faster.



40ms Added latency

30mi n New MTTR

Strong Consistency

The honest trade-off

The 40ms isn't free, but it's cheap where it lands. Analytics events are non-critical. Search still needs pushed data. Replicate only when it's fine to be stale.



04 · loud failure is a feature

External calls used to silently overwrite our config, with mismatches handled instead of rejected, and bugs surfacing days later in support tickets. Now every external write is logged and tracked, and mismatches reject with a structured error. That bought back trust in the data.



05 · Rollout without fire



Stage 1 of 3 · shadow rollout

No live traffic yet. Old logic stayed authoritative; new logic ran alongside it, logged but never served. We only moved to stage two once the mismatch rate held at zero for two weeks straight.

Stage 2 of 3 · live, with a fallback

The last 5% took longer than the first 95% combined.

01

Day 1
10%, internal traffic only.

02

Days 2-3
Ramped from 20% to 50%.

03

Day 4
75% of traffic.

04

Day 5
100%. The hardest stretch.



Stage 3 of 3. No fallback left.

It’s never over until it’s over

We didn’t miss any fallback once removed

Three things to take home

Audit boundaries

If a single user journey/business use case crosses multiple teams, the boundary might be wrong.

Find shared fields

Anything in two or more databases is a distributed monolith waiting to happen.

Use shadow mode

Keep the old system as the source of truth until the new one proves it's ready.

Thank you.

Ayush Kumar
Senior Software Engineer, Pricing Tech — GetYourGuide

We're hiring
Join the team building the tech behind global travel experiences.
getyourguide.com/careers