Platform Engineering Day North America 2026

Where Should Platform Complexity Live?

Lessons from a Cross-Team PR Preview Platform

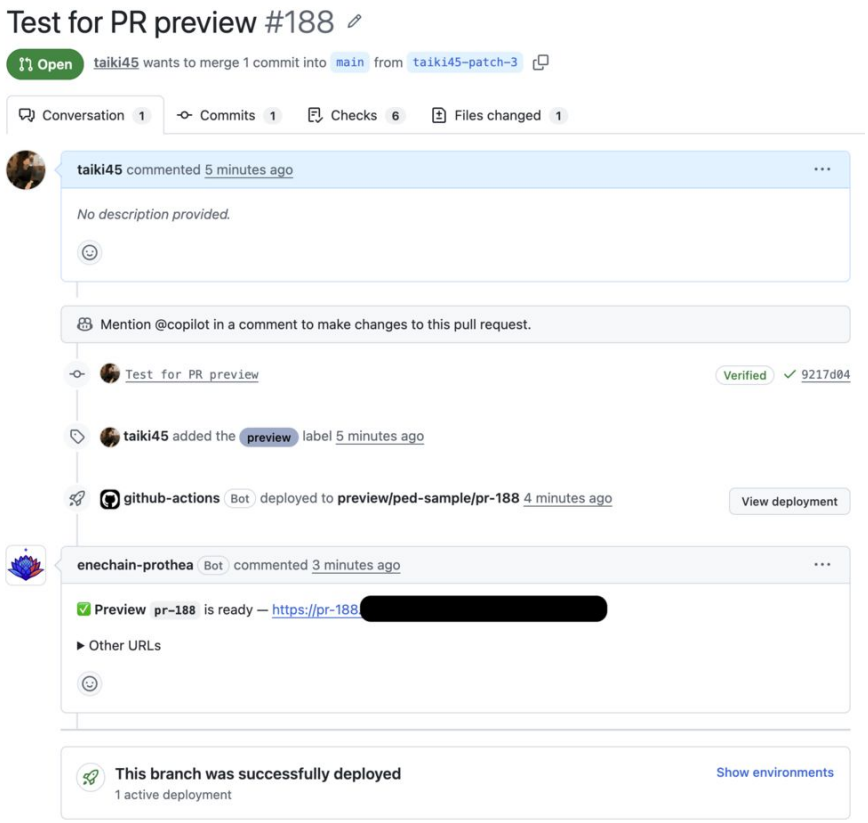
Taiki Ono
Principal Software Engineer

enechain

Introduction

What is a per-PR preview environment?

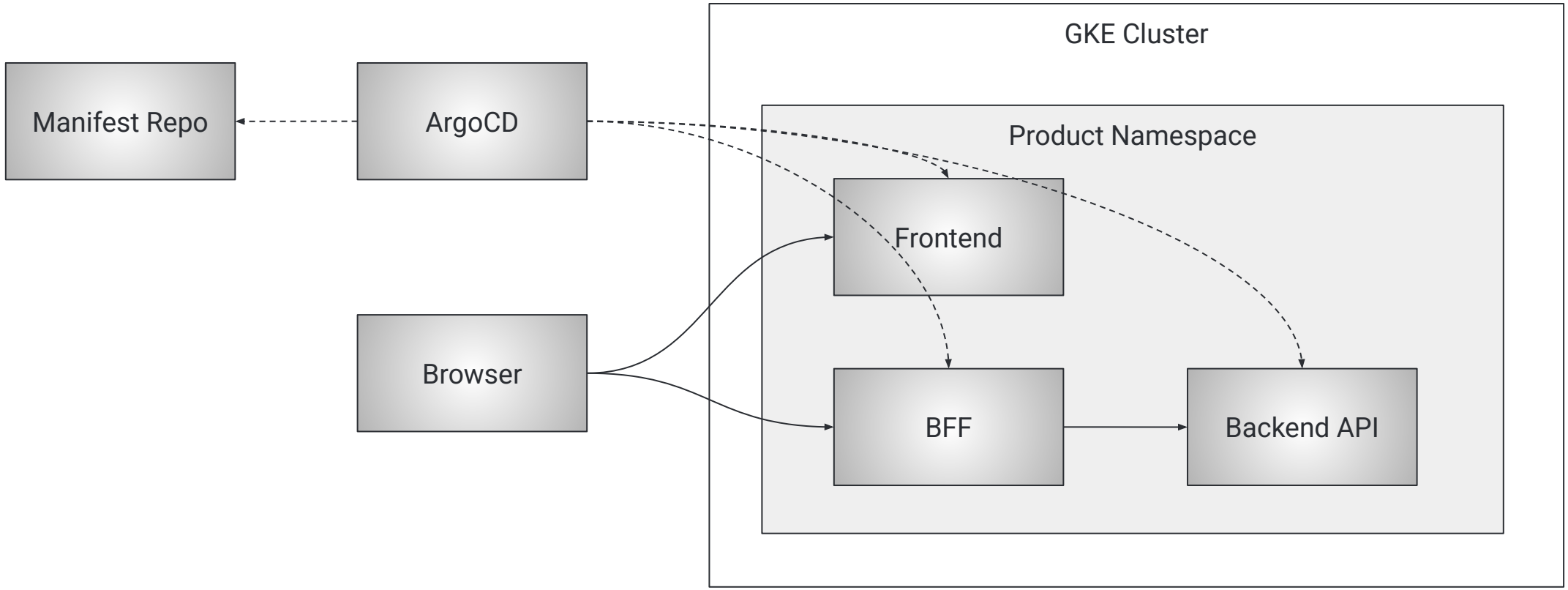
- Problems we want to solve
  - Growing PR review load
  - More diverse developers
  - Shifting feedback left
- User journey
  - A preview is created when a PR is opened
  - Each preview is accessed via its own URL
  - The preview is deleted when the PR is merged



PR previews at enechain

- First built by one product team
  - Proven in-house, with strong demand as a platform feature
- Now built and run as a company-wide platform feature
  - Design started in February, rollout from late March
  - Adopted by our most active and our most complex products
  - Actively used across the company
  - Technical details on our tech blog (in Japanese)
    - https://techblog.enechain.com/entry/intro-prothea

Our tech stack



Choosing an Approach

Two approaches

- Generate manifests
  - Generate preview manifests and apply them with Argo CD or similar
- Clone in the cluster
  - Leave manifests untouched and create preview resources directly in the cluster
  - Requires a controller, so the platform gets more complex

→ We chose the latter. Why?

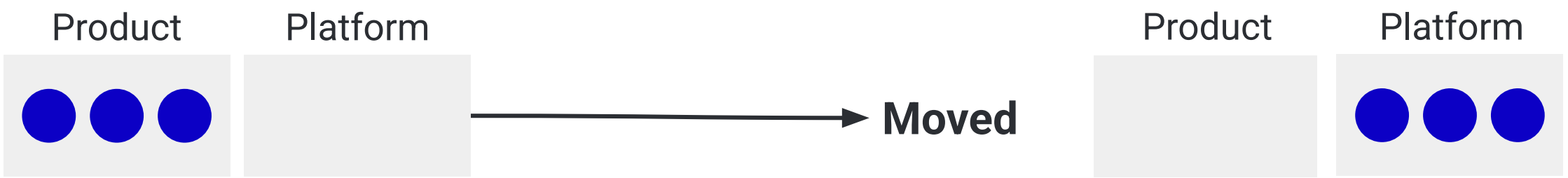
What manifest generation leaves to developers

- Kubernetes manifests carry high cognitive load on their own
  - Our platform doesn't fully abstract them yet, so developers still deal with them
- Templating with Helm or Kustomize adds more
- Do we add a preview layer on top?
  - Preview conventions are in-house, not an industry standard

→ The cognitive load stays with product teams

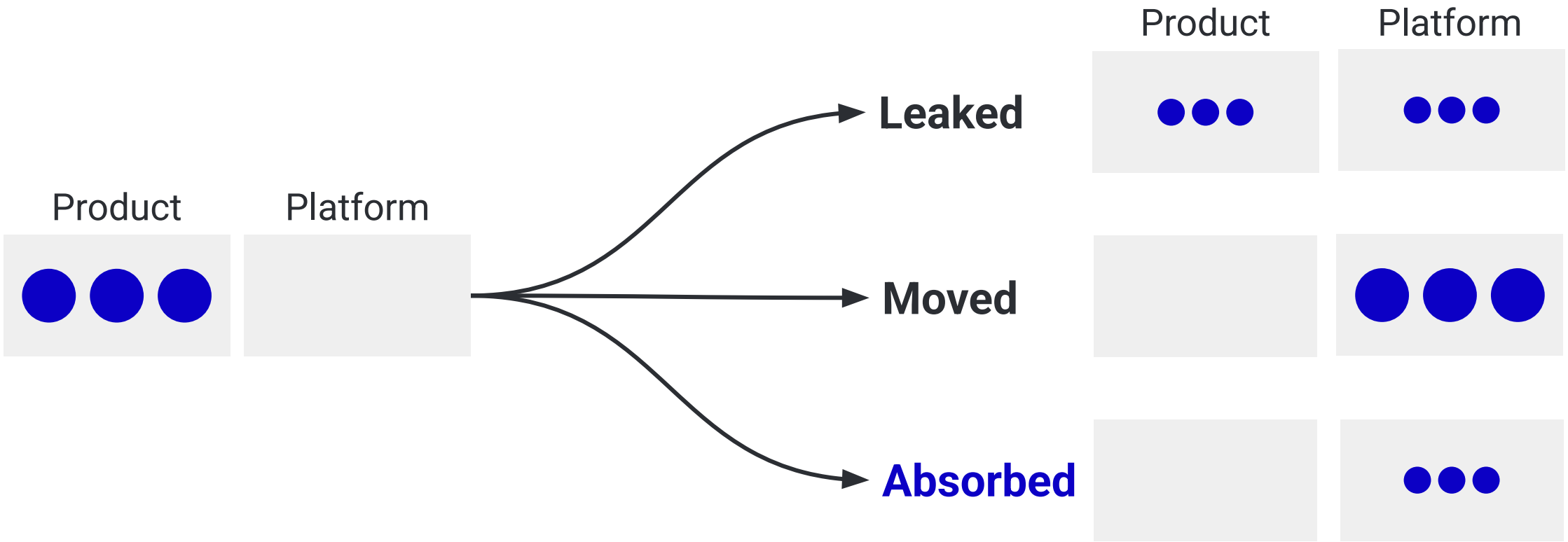
Mixing in in-house conventions increases it further

Platforms and complexity



→ How do we break this trade-off?

Leaked, Moved, or Absorbed?



Absorbed: the platform takes in the complexity and keeps it simple

How we decided

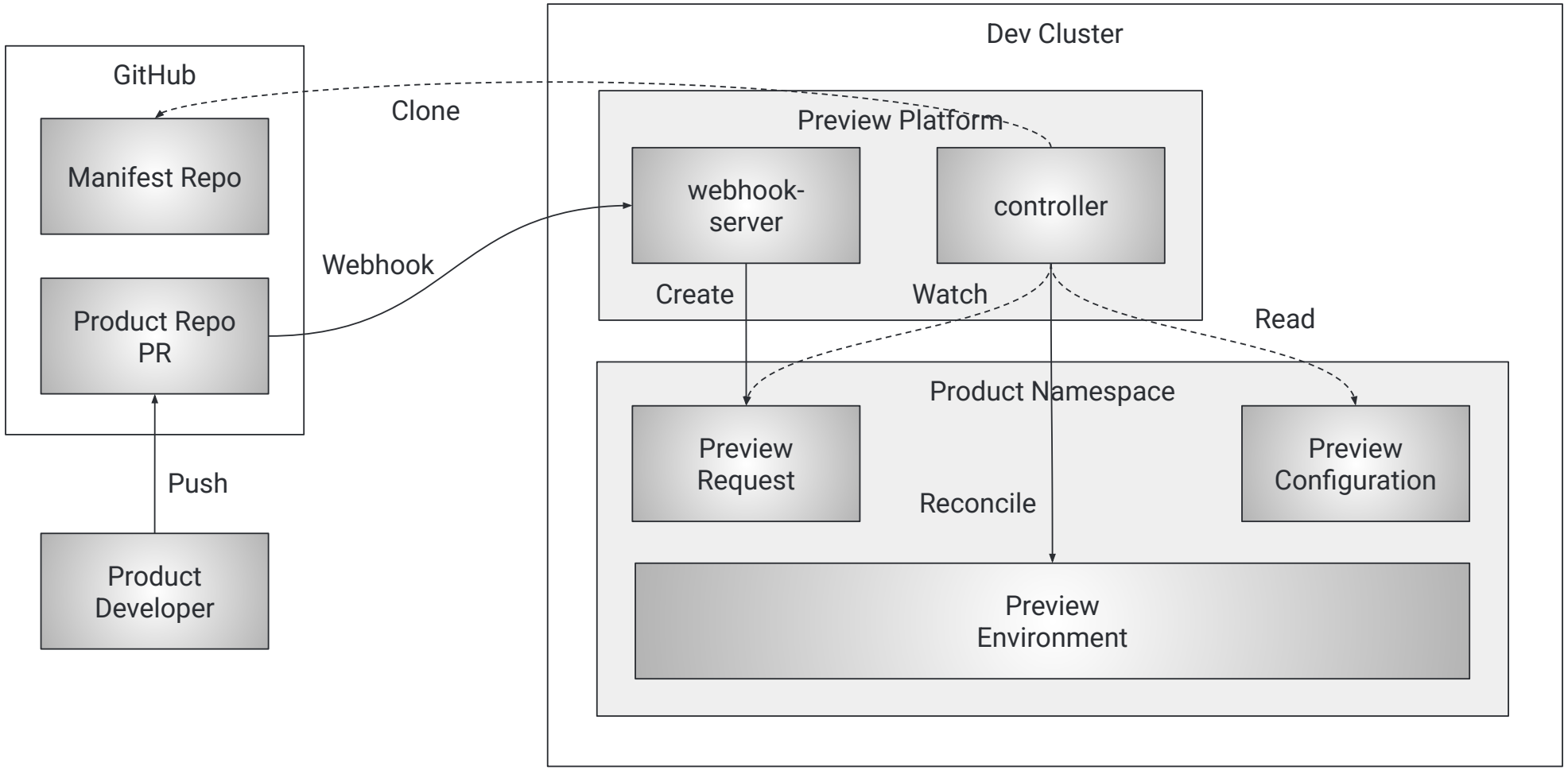
- Is it worth taking on?
  - Many products and varied setups, with more on the way
  - If not, choosing Leaked is the right decision
- Can we get to Absorbed?
  - Keep platform concerns out of product manifests → not Leaked
  - Git clone and the controller pattern → stays simple, not just Moved
  - If not, compare Moved with Leaked

Design

Three ways complexity leaks

- 🚫 Making them write
  - Developers write or review detailed config
- 🚫 Touching what they own
  - Platform concerns get into product manifests and routing
- 🚫 Making them dig
  - Developers read platform internals and logs when stuck

Overview


GitHub
Manifest Repo
Product Repo
PR
Product
Developer
Push
Clone
Webhook
Dev Cluster
Preview Platform
webhook-
server
controller
Create
Watch
Read
Product Namespace
Preview
Request
Reconcile
Preview
Configuration
Preview
Environment

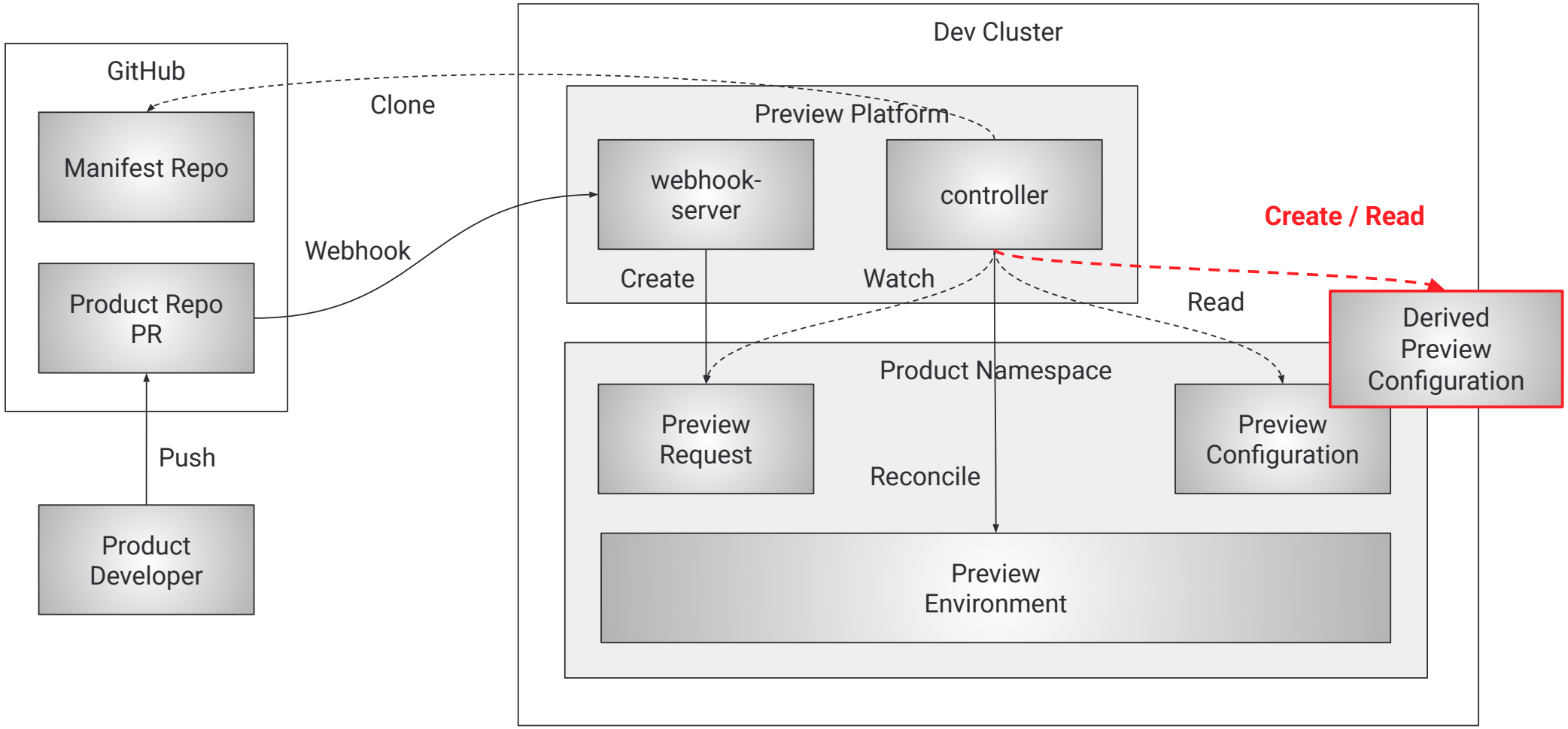
Don't make them write

Config holds only basic bindings

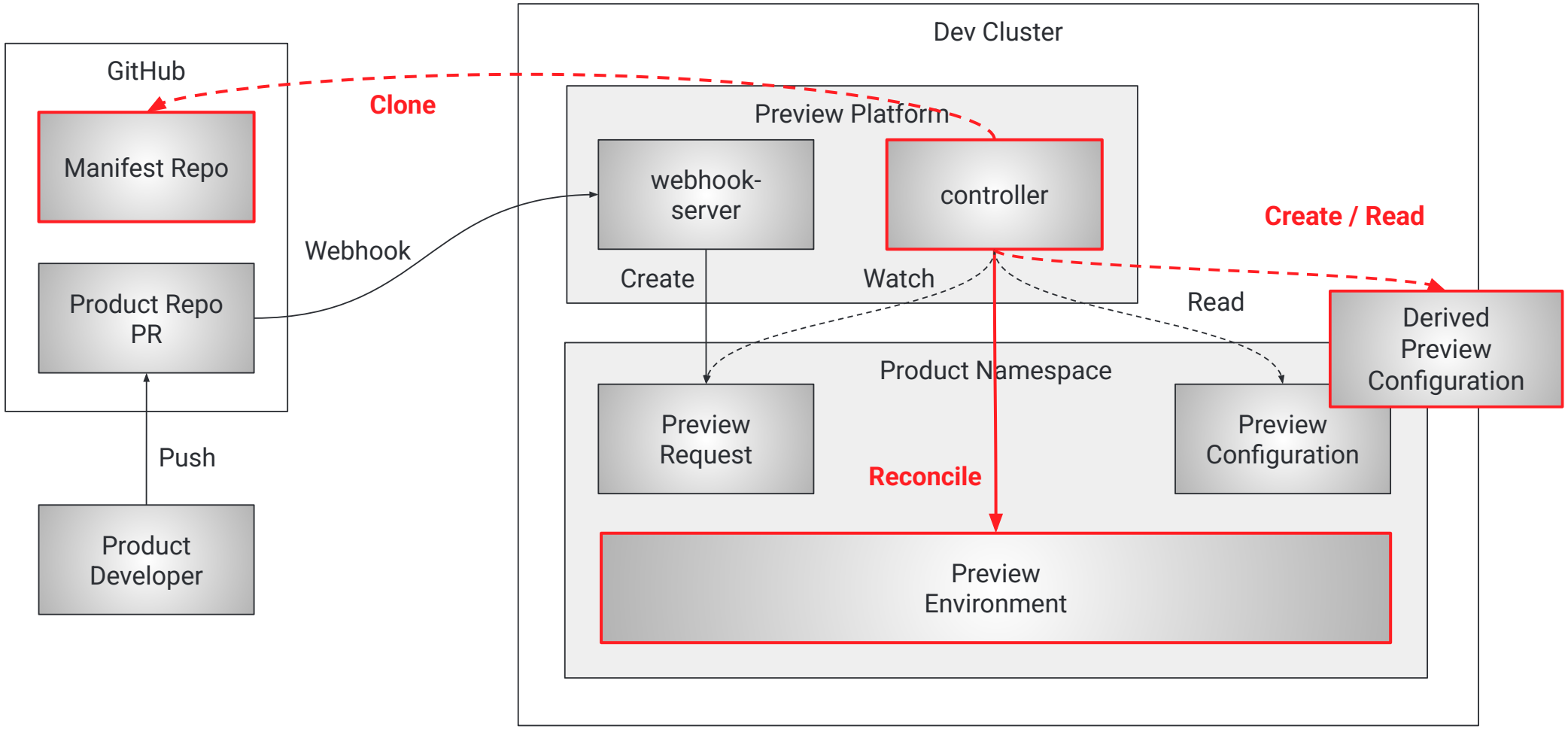
→ How do we handle differences between products?

```yaml
apiVersion: prothea.enechain.com/v1alpha1
kind: PreviewConfiguration
metadata:
  name: demoapp
spec:
  webhookRepository: enechain/demoapp
  manifestRepository: enechain/demoapp
  kustomizationPaths:
    - k8s/overlays/dev
  externalRouting:
    primaryOrigin:
      hostname: demoapp.example.com
```

Deriving the preview config



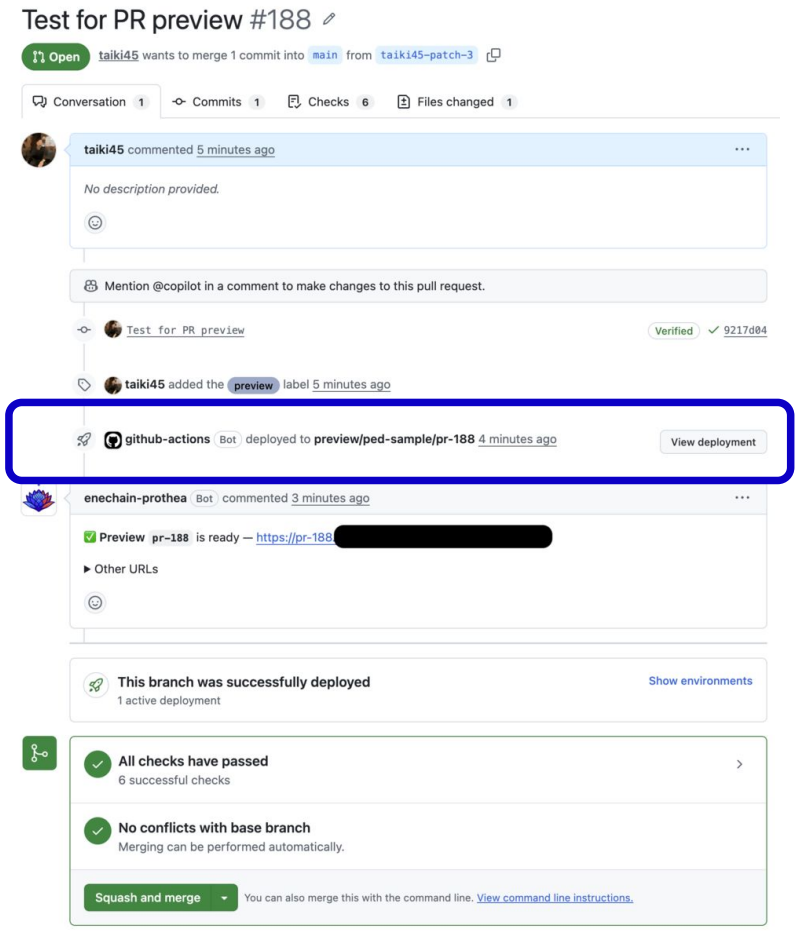
Don't touch what they own: Clone from the base



Don't make them dig

- Day to day: low-noise status on the PR
  - Via the GitHub Deployments API
- When they want details: one layer down
  - The derived config in a custom resource spec
  - Custom resource statuses
- When things go wrong: clear feedback on the PR
  - Even for developer mistakes

→ Platform as Product



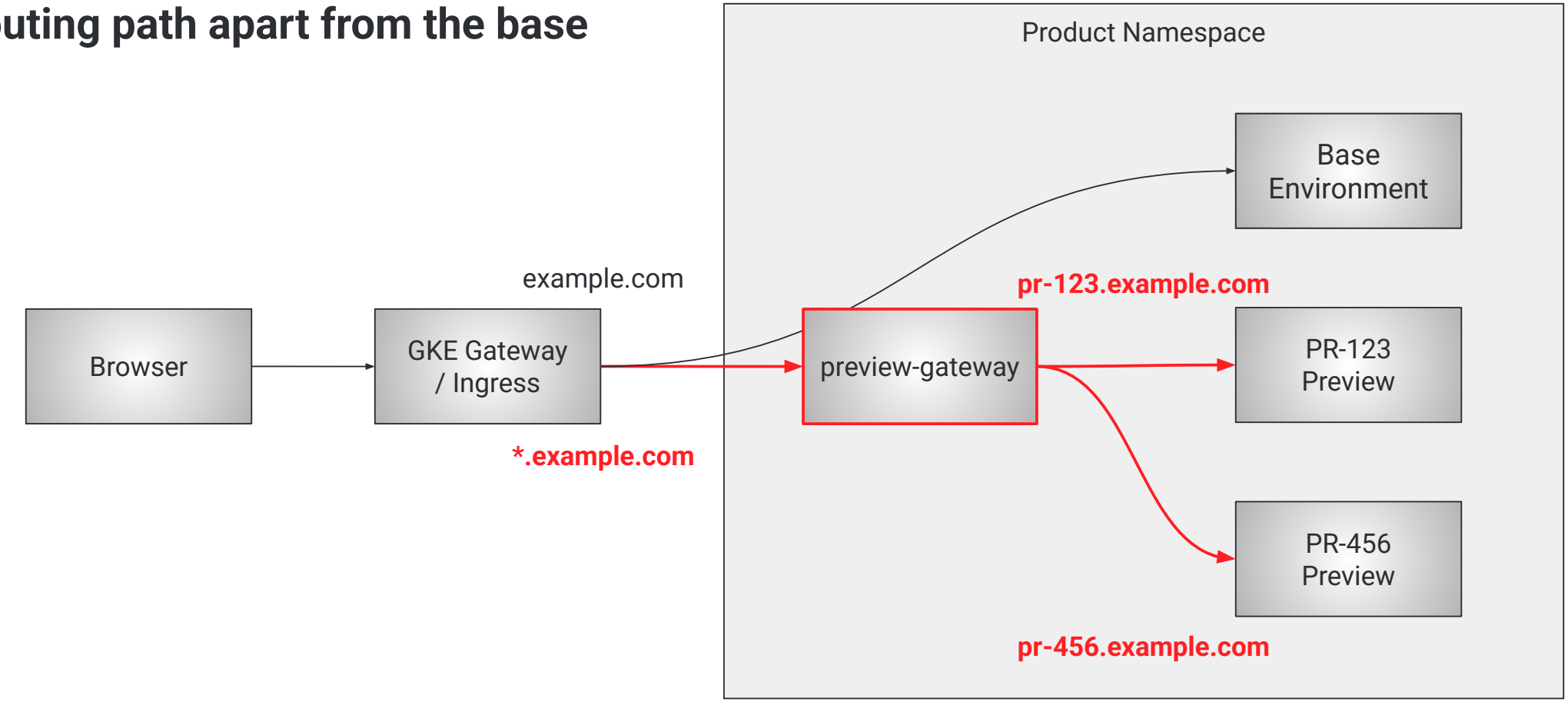
Lesson: progressive disclosure

- At first: developers wrote detailed config
  - Even when a CLI or an AI agent generates it,
    they still have to understand it → Leaked
- Explicit or implicit? Consider progressive disclosure
  - Hide it day to day, show it one layer down when needed

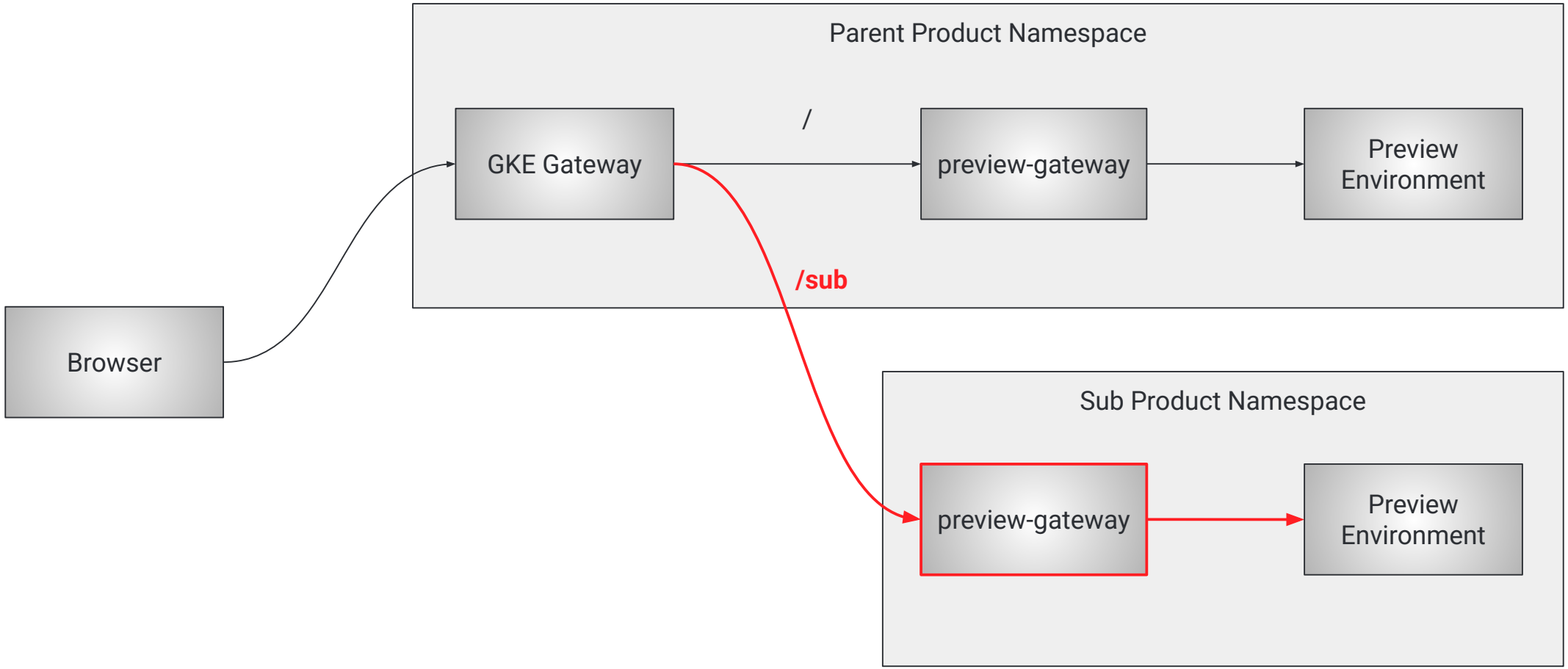
Transparent Networking

Don't touch what they own

A routing path apart from the base

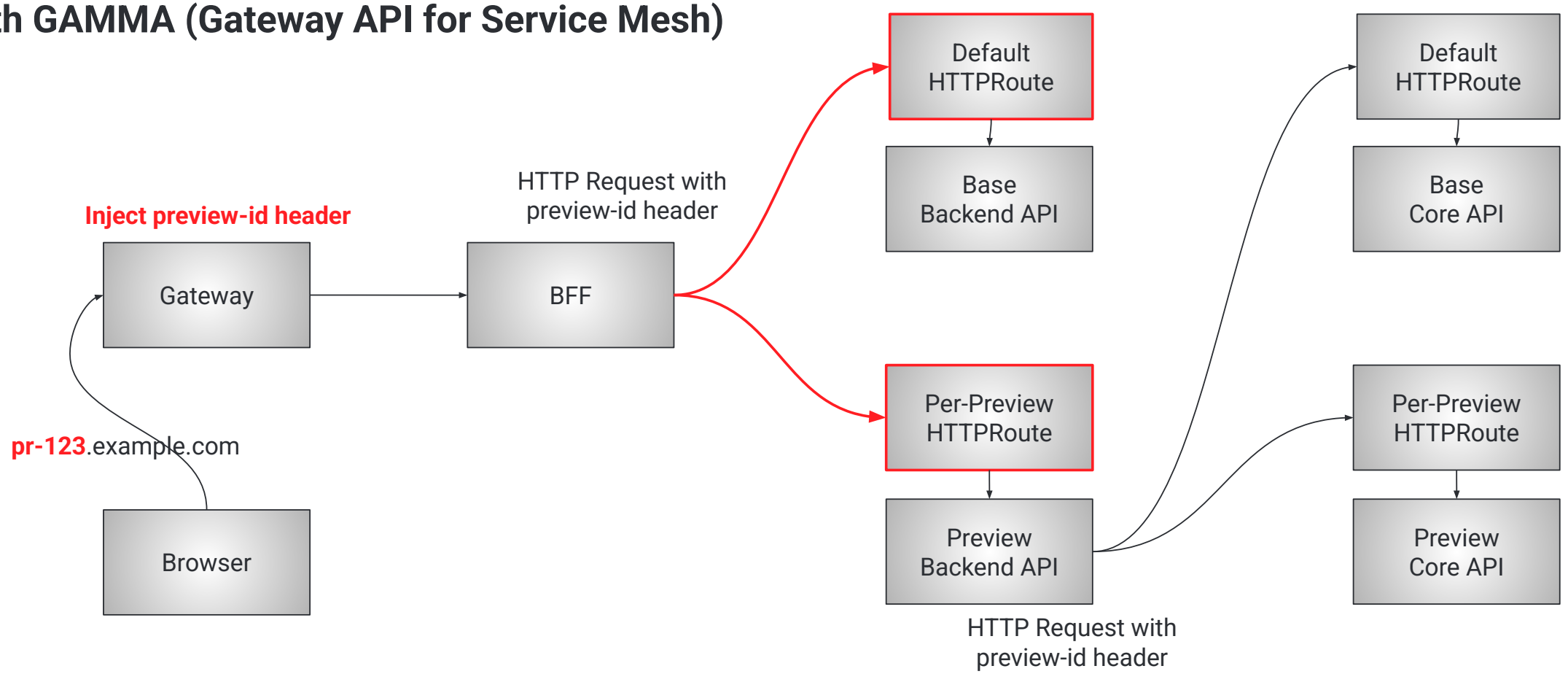


The platform absorbs unusual setups too



Service-to-service traffic

With GAMMA (Gateway API for Service Mesh)



Drawing the Boundary

What we left to apps

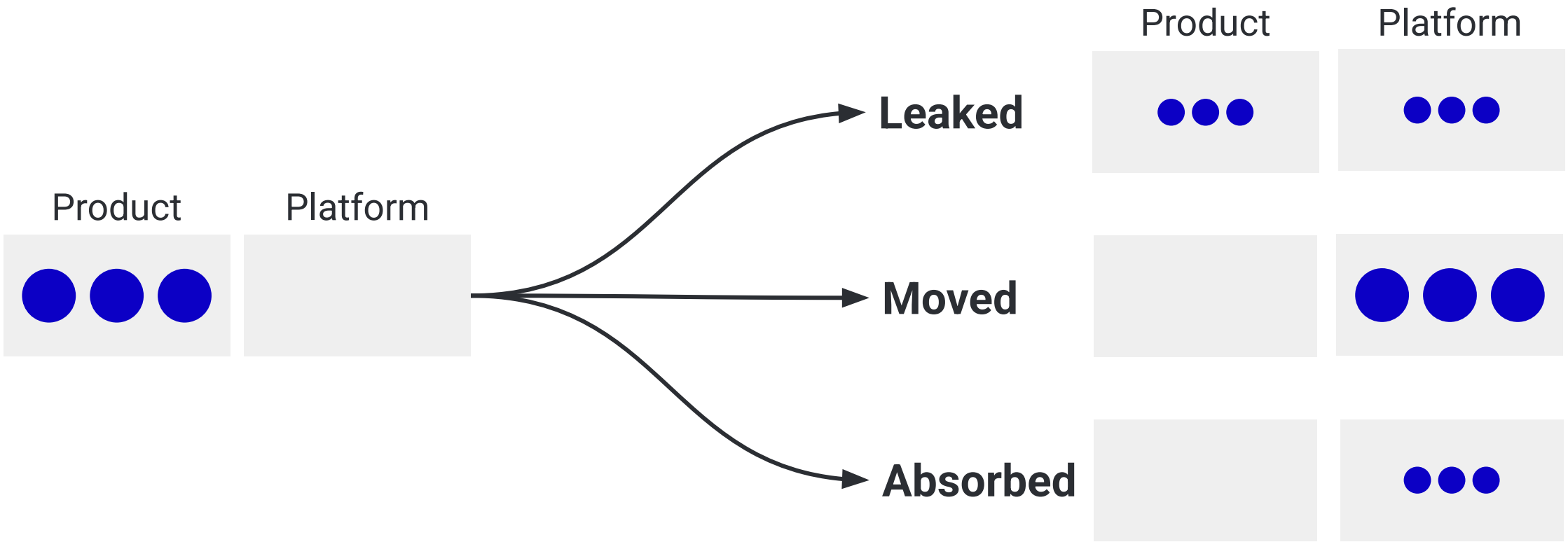
- Building preview container images
  - A post-build hook for the shared build workflow
- Pointing the frontend at the preview
  - Part of the setup steps (no good solution yet)
- Cross-Origin Resource Sharing
  - Up next

Cross-Origin Resource Sharing

- Domains change in previews, so this is a platform concern
  - Answer preflights at preview-gateway, keep apps unchanged
- If it can't stay Absorbed, hand it back to apps
  - The CORS spec doesn't allow wildcard headers on credentialed requests
  - Keeping it means making product teams write header config → Leaked
  - On the app side, it's a one-line change

Wrapping Up

Summary: where should complexity live?



→ **Picture Absorbed, then decide where it lives under your constraints**

Summary: getting to Absorbed

- Don't let it leak
  - Don't make them write: bindings only, derive the rest
  - Don't touch what they own: keep platform concerns out of product assets
  - Don't make them dig: troubleshoot without knowing the internals
    - Platform as Product
- Don't stop at the Moved pattern
  - Build on standards to stay simple (controllers, service mesh)
- Keep asking whether it stays Absorbed
  - Redraw the boundary when needed

Thank you

enechain