NATWEST × SYNTASSO

Applying Platform as a Product in a *300-year-old* bank

How one of the UK's largest banks applied industry leading ideas to turn undifferentiated heavy lifting into a self-service platform

THE INDUSTRY VIEW

Abby Bangser
Syntasso Founding Engineer
KubeCon Co-Chair and CNCF Ambassador

THE CASE STUDY

Joel King
NatWest Group Principal Engineer

Most engineering effort is non-differentiating

80%

Generic tooling
CI/CD pipelines, Kubernetes, monitoring stacks…

Duplicated heavy lifting
The same plumbing repeated across every team

Tech debt & maintenance
Keeping the lights on crowds out innovation

Gartner

“Platform engineering has emerged in response to this increasing cognitive load.”

McKinsey

“Reducing the cognitive and operational load on developers is a critical step toward increasing product innovation.”

Google DORA, 2025

“There is a direct correlation between a high quality internal platform and an organization’s ability to unlock the value of AI.”

NatWest is a 300-year-old, 20-million-customer bank

A UK-focused bank serving over 20 million customers — at a scale that makes every platform decision compound.

20M+
customers served

#1
lender for UK infrastructure

~59k
staff worldwide

11.4M
mobile app users

1M
new customers added in 2025

700k+
new credit-card customers in 2025

81.8%
of retail banking customers bank entirely digitally

WELCOME

Every story needs the forest view and a tree example



ABBY · THE FOREST

The big picture

Tracking industry wide socio-technical patterns.

Codifying trends and supporting measured improvements.



JOEL · THE TREE

A real example

How a single 300-year-old bank actually built it.

The problem, the architecture, and next step scale.

OUR ROADMAP

Where we're going today

The Shift

Why platforms, why now

Building

From theory to outcomes

Scaling

Supporting future vision

At NatWest, scale and complexity collide

COMPLEXITY

135+

patterns in production — with **unsustainable growth** , ever-increasing complexity, and delivery that has slowed to a crawl.

SCALE

10s departments

100s teams

1000s staff

All moving to **containerisation** at once — every team solving the same infrastructure problems in parallel.

The cost: cognitive load, drift, and slow change

01 High cognitive load on development teams

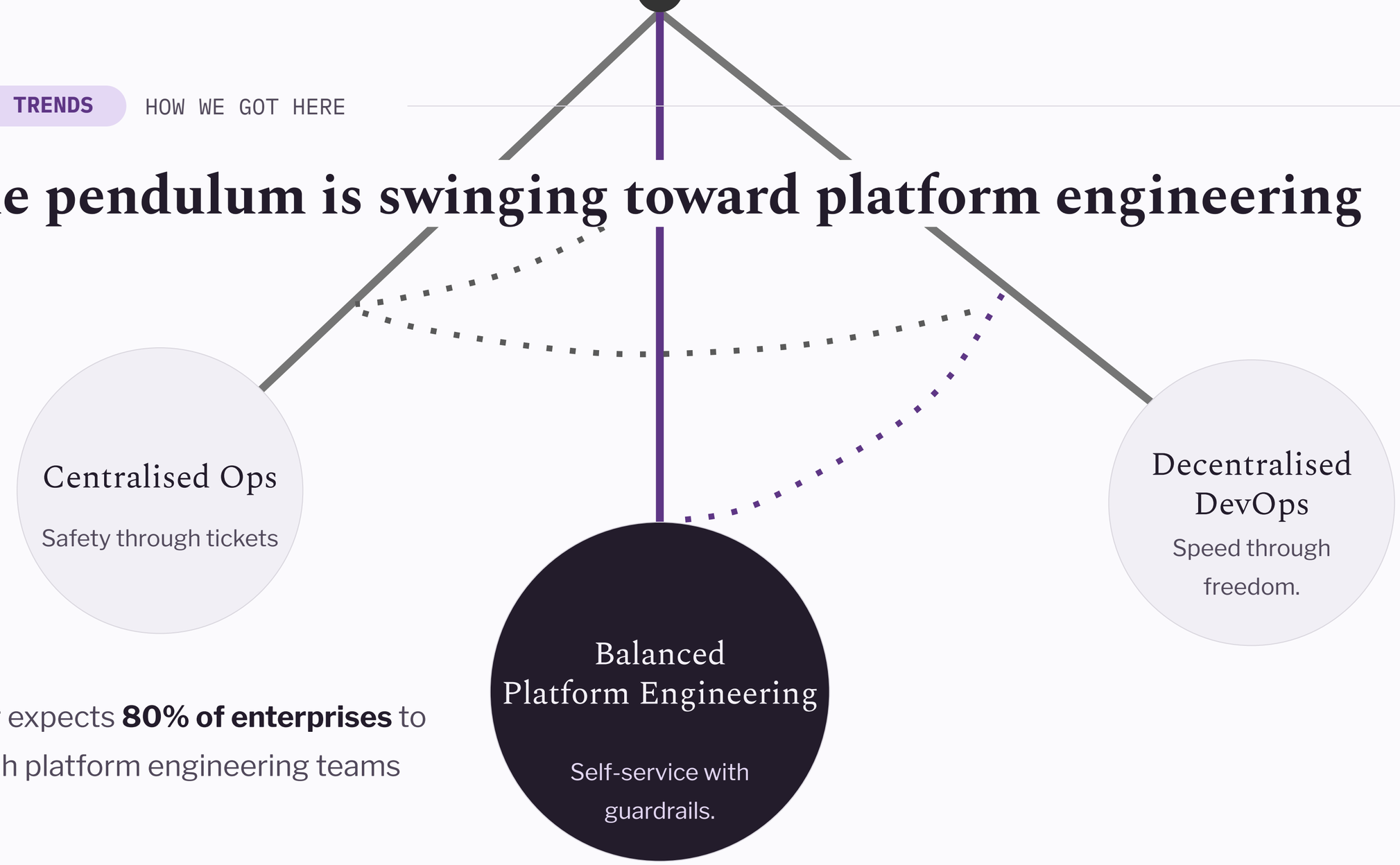
02 Duplication of effort across application teams and services

03 Inconsistent environments and configuration drift

04 Slow, manual, and error-prone change processes

05 Limited visibility and control across multi-cluster fleets

The pendulum is swinging toward platform engineering



Gartner expects **80% of enterprises** to establish platform engineering teams

The hype-cycle needed grounding

A fragmented landscape: everyone agreed platforms mattered but not on what they were

SCEPTICISM

Dismissed as "new hype" or "rebranded ops".

TERMINOLOGY CHAOS

No industry-wide agreement on what this is.

UNCERTAINTY

Big projects with poor ROI.

TOOL-CENTRIC

Repurposing tools over business outcomes.

THE TRENDS

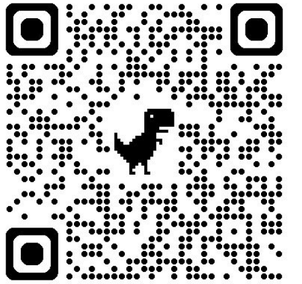
Capturing target state

CLOUD NATIVE
COMPUTING FOUNDATION

BLOG / COMMUNITY POST

Announcing a white paper on Platforms for Cloud Native Computing

Posted on April 11, 2023 by Josh Gavant + Abby Bangser



For a more detailed view of platform maturity, please refer to the Platform Engineering Maturity Model.

CLOUD NATIVE
PLATFORM ENGINEERING COMMUNITY

Attributes of platforms

After defining what a platform is and why an organization might want to build one, let's identify some key attributes that affect the success of a platform.

1. **Platform as a product**. A platform exists to serve the requirements of its users and it should be designed and evolved based on those requirements, similar to any other software products. Platforms should provide the necessary capabilities to support the most common use cases across product teams, and prioritize those over more specific capabilities that are only used by a single team to maximize the value delivered.
2. **User experience**. A platform should offer its capabilities through consistent interfaces and focus on the user experience. Platforms should endeavor to meet their users where they are, acknowledging that users can be human or agentic, which means providing a combination of GUIs, APIs, command-line tools, IDEs, and portals. For example, developers might consume such a capability via the IDE, testers might use a command-line tool, whereas a product owner might use a GUI-based web portal.
3. **Documentation and onboarding**. Documentation is a key aspect of a successful software product. To be able to use a platform's offerings, users require documentation and examples. A platform should be delivered with proper documentation addressing the needs of its users. It should also provide tools to accelerate the onboarding of new projects that can help users consume the necessary platform services in a quick and simple way. For example, the platform could offer a reusable supply chain workflow for building, scanning, testing, deploying, and observing a web application on Kubernetes. Such a workflow could be offered with an initial project template and documentation, a bundle often described as a *golden path*.
4. **Self-service**. A platform should be self-serviceable. Users must be able to request and receive capabilities autonomously and automatically. This property is key to allowing a platform team to enable multiple product teams and scale as needed. The platform capabilities should be available on-demand via the interfaces described above. For example, it should be possible for a user to request a database and receive its locator and credentials by running a command-line tool or filling out a form on a web portal, without waiting for a manual review or approval.
5. **Reduced cognitive load for users**. An essential goal of a platform is to reduce the cognitive load on product teams. A platform should encapsulate implementation details and hide any complexity that might arise from its architecture. For example, a platform might delegate certain services to a cloud provider, but users should not be exposed to such details. At the same time, the platform should allow users to configure and observe certain services as needed. Users must not be responsible for operating the services offered by the platform. For example, users may often require a database, but they shouldn't have to manage the database server.
6. **Optional and composable**. Platforms are intended to make product development more efficient, so they must not be an impediment. A platform should be composable and enable product teams to use only parts of its offerings. It should also enable product teams to provide and manage their own capabilities outside of the platform's offerings when necessary. For example, if a platform doesn't provide a graph database and it's required for a product, it should be possible for the product team to provision and operate a graph database themselves.
7. **Secure and compliant by default**. A platform should be secure by default and offer capabilities to ensure compliance and validation based on rules and standards defined by the organization. Security, governance, and compliance requirements for the business should be baked into the platform, reducing users' cognitive burden while ensuring consistent enforcement.

A new approach: treat the platform as a product, run it with GitOps

01 · PLATFORM AS A PRODUCT

A dedicated feature team owns the platform as a product, adding a layer of abstraction over the plumbing.

PRODUCT TEAM

02 · CURATED PRODUCTS

Components are packaged as reusable, declarative building blocks behind one common API.

COMMON API

03 · SOURCE OF TRUTH

Git is the single source of truth: every change a reviewed, traceable commit with guardrails.

GITOPS MODEL

04 · RECONCILIATION

Continuous reconcile loops compare live state with desired state, so drift is corrected.

NO CONFIG DRIFT

05 · FLEET CONTROL

One definition, many clusters: Git-driven rollout across public and private cloud.

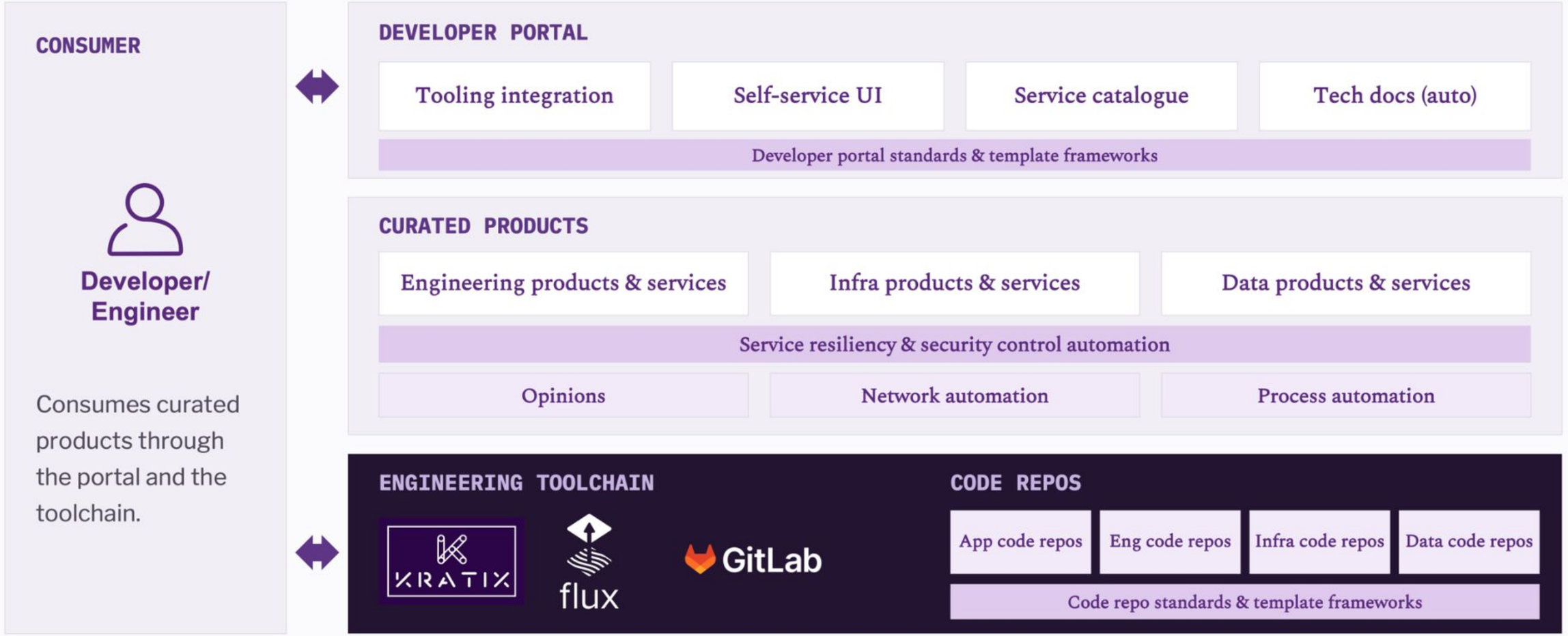
PUBLIC + PRIVATE CLOUD

06 · DEVELOPER EXPERIENCE

Consumers request curated products via portal, API or GitOps, with best practice built in.

BEST PRACTICE BUILT IN

Platform as a Product, end to end



From request to fleet, at a glance

1 · REQUEST

Self-service request
Raised from the developer portal, via GitOps, or directly against the API.

One contract
Consumers don't need to know what sits underneath.

2 · CONTROL PLANE

Kratix
Promise CRD + pipeline

Resource configure pipeline
Workflows run per request

ECR / registry
Container images and OCI artifacts

3 · DESIRED STATE

GitLab
Holds the desired state

Flux CD
Reconciliation loop keeps clusters aligned with Git

4 · PROVISION

ACK controllers
K8s-native AWS

Terraform Enterprise
HCP Terraform

Other integrations
On-prem IaaS, ServiceNow etc.

AWS accounts: EKS, S3, IAM, RDS, VPC

5 · FLEET

Customer clusters 1 … N
Each one reconciles from Git

Workloads + config
Delivered by GitOps

Product updates
Roll out across the fleet

Everything after the request is declarative, versioned in Git, and reconciled continuously.

How a request becomes desired state

01 · REQUEST

Ask for a product

A developer or app team requests a product from the portal, the K8s API or via GitOps.

02 · PROMISE

The API is the contract

Every product and component is a Kratix Promise. Its CRD defines what can be requested.

03 · PIPELINE

Workflows do the work

A resource configure pipeline runs per request, with opinions, best practice and guardrails built in.

04 · STATE

Desired state lands in Git

The output is committed to GitLab. Flux and Kratix reconcile from there.

PRODUCERS · PLATFORM TEAMS

Build and own components, then curate them into products with opinions, security and guardrails built in. For example an EKS product or SPAs.

CONSUMERS · DEVELOPERS AND APP TEAMS

See one product API, agnostic to what implements it: Terraform, ACK, Crossplane or something else.

Provision and manage the fleet

01 · PROVISION

Downstream tooling creates the real infrastructure.

ACK controllers
K8s-native AWS

Terraform Enterprise
HCP Terraform

→ **AWS accounts**
EKS, S3, IAM, RDS, VPC

Other integrations
On-prem IaaS, ServiceNow etc.

02 · MANAGE THE FLEET

Every customer cluster reconciles from Git.

Cluster 1	Cluster 2	Cluster 3	… Cluster N
workloads + config	workloads + config	workloads + config	workloads + config

- Product versions roll out to clusters in managed groups
- Security patches and new features need no consumer action
- One model across public and private cloud

RUNS ALONGSIDE

CI guardrails for security scanning and code quality · Kyverno and cloud control frameworks for policy · CloudWatch and Grafana for observability

Paper Proposal

Innovators are building composable marketplace platforms

EXPERIENCES

> “This whole connected, integrated experience is also removing this cognitive load” - Lego

> "Platform teams are most responsible for the interfaces and user experiences." - CNCF

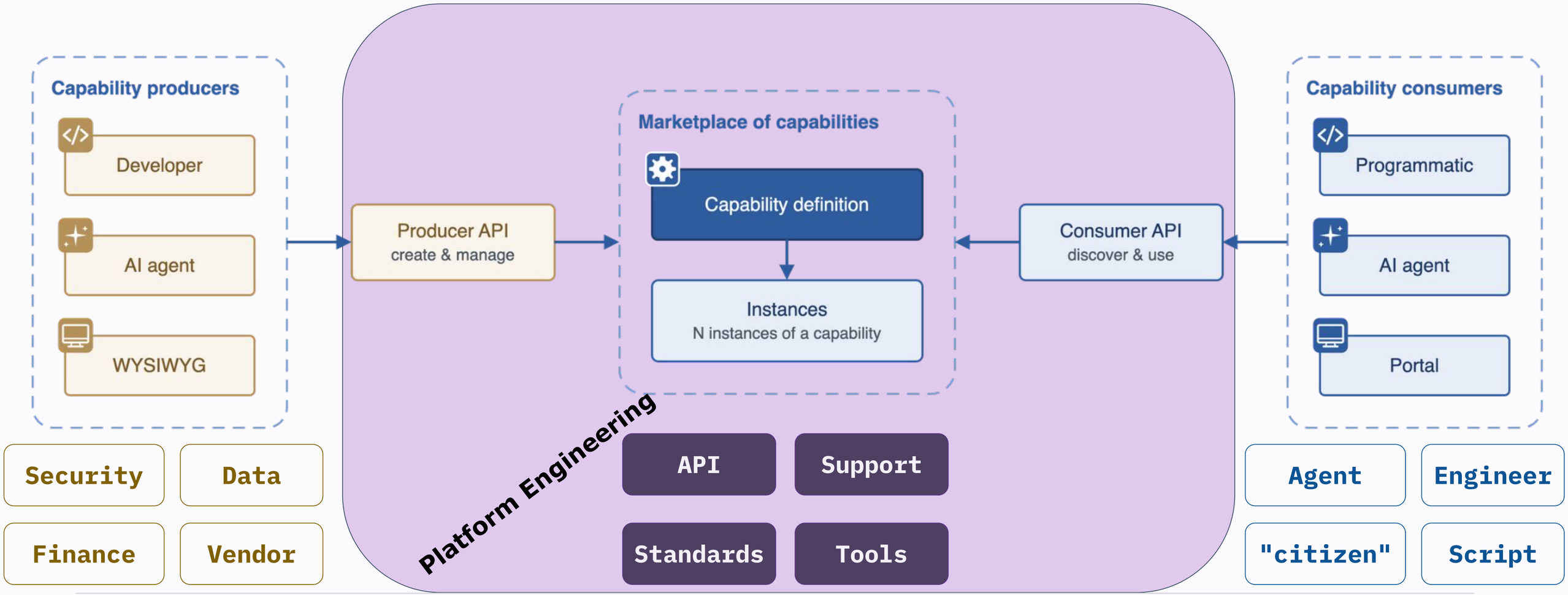
CAPABILITIES

> "Each capability becomes something teams can adopt with confidence: it has an owner, a clear contract" - ING

> “From a technical perspective, it’s the plugin architecture of Backstage that enables this autonomy.” - Spotify

Paper Proposal

Defining the multi-player friendly interface



Paper Proposal

Building collaboration through clear expectations



Capability principles

- Product-scoped
- Self-service
- Operationally owned

Platform principles

- Extensible
- Composable
- Declarative
- Governable
- Observable
- Stable

Paper Proposal

And then the five factor contract that enforces it



The five factors in the NatWest platform

01	Defined Contract	Every product and component is a Kratix Promise: a K8s CRD API that consumers use via portal, API or GitOps.	PROMISE CRD
02	Declarative Inputs	Consumers declare the outcome they want. Requests land in Git as desired state, not as tickets or scripts.	GITOPS · GITLAB
03	Encapsulated Dependencies	Products are built from independently owned components. Consumers don't see Terraform, ACK or Crossplane underneath.	COMPONENTS → PRODUCTS
04	Safe Reconciliation	Flux and Kratix run continuous reconcile loops, and product updates roll out fleet-wide in managed groups.	FLUX · KRATIX
05	Operational Evidence	Observability for producers and consumers is built in, with central policy and Git history making every change verifiable.	GRAFANA · KYVERNO

Outcomes of applying Platform as a Product

01 Faster time to value

02 Higher Autonomy

03 Improved Security and Compliance

04 Standardisation at scale

05 Clear ownership boundaries

THE RECAP

Where we've been today

The Shift
It happened, what does it mean

- Platforms fill the gap of what is unique to your business but common to your teams
- It is worth building once, but accept not everyone will fit

Initial Build
Product over project approach

- Success socio-technical both in investment and outcomes
- Early customers and contributors validate the path

Extended Innovation
Sustainability breeds growth

- The only guarantee is change, reducing bottlenecks is key
- Clear ownership boundaries and outcomes drives engagement

THE TAKEAWAY

Outcomes *over* outputs demands flexibility

The industry is moving toward composable, self-service platforms. NatWest shows it's buildable today with golden paths, reconciled state, and capabilities consumed as products.

THANK YOU AND QUESTIONS?

NatWest × Syntasso

THE FOREST
Abby

THE TREE
Joel