Navigating the No-Code to Full-Code Spectrum
A Platform Engineering Journey

Platform Engineering Day
NORTH AMERICA

November 12, 2024
Salt Lake City

Maximilian Blatt
Cloud Advisory Consultant
Accenture

Jared Watts
Founding Engineer
Upbound

Preflight Checks

Why are we building a Platform?

What do we want to accomplish?

Why build a Platform?

Many **dev teams face the same challenges** when deploying cloud infrastructure:

- Know and apply configuration best practices
- Ensure security and compliance
- Integrate into internal IT landscape

The solution:

A centrally managed platform to take the load of off developers so they can focus on their actual goals

We want: a *Universal* API

Many companies rely on a **lot of different technologies** to run their infrastructure and manage their development process.

Developers **need to understand a lot of different tools** and APIs

i.e. RDS, Azure Devops, S3, Artifactory and many more

By building a **central Kubernetes-based platform** we can provide a universal API with the **same set of tools for everything**

We want: a *Declarative* API

Many APIs and tools are designed in an imperative fashion:

- **Different data formats and schemas**
- **No continuously monitored state** unless the system is accessed
- **Manual action required** to update individual components

By building a **Kubernetes-based platform** that relies on **continuous reconciliation and eventual consistency** we allow developers to define their infrastructure in a **GitOps fashion**

We want: a *Simple* API

- Many APIs, i.e. cloud provider's, offer a **huge set of options** to configure infrastructure to specific needs
- Companies put their own regulation regarding compliance and security on top
- **Smaller dev teams don't have the capacity** to manage all theses components.

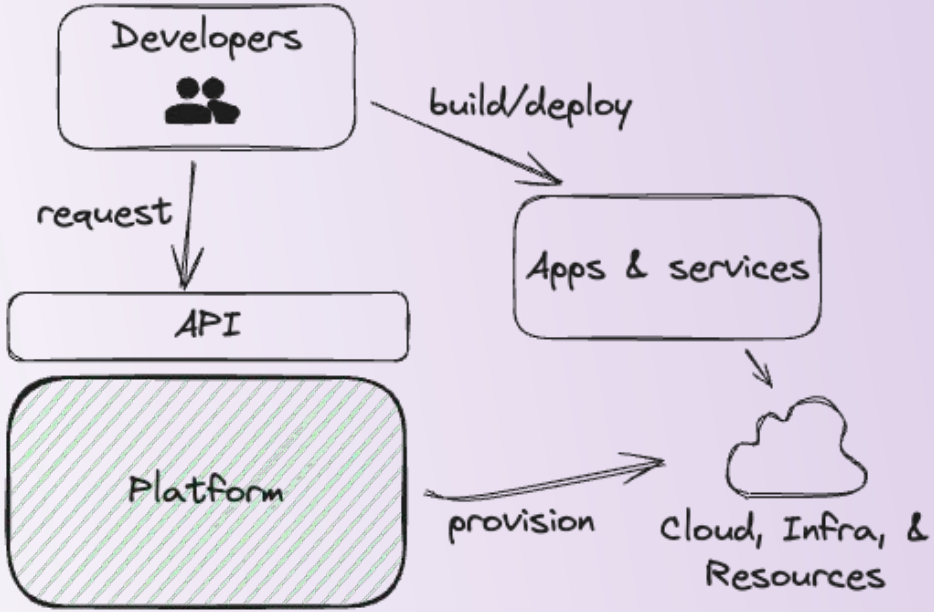
By providing an **abstract platform API with standardized components** we can limit the choice of configuration values, provide **compliance by default** and **simplify operations**.

Step 1: Control Plane

2018 - The Journey Begins

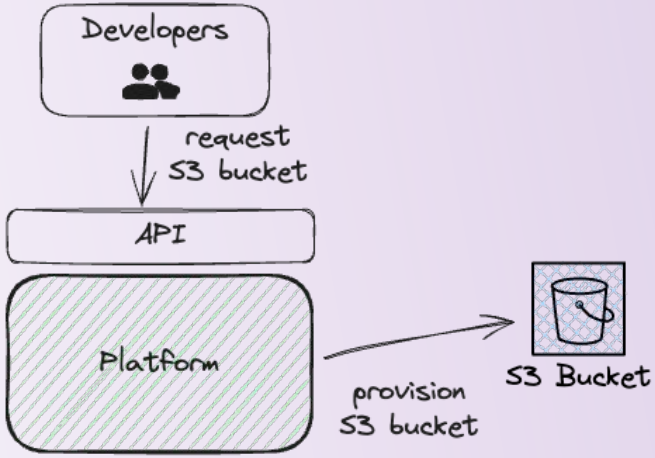
Where do we start?

- Provide developers with infrastructure/resources
- Keep those resources healthy



Platform API Starter Pack

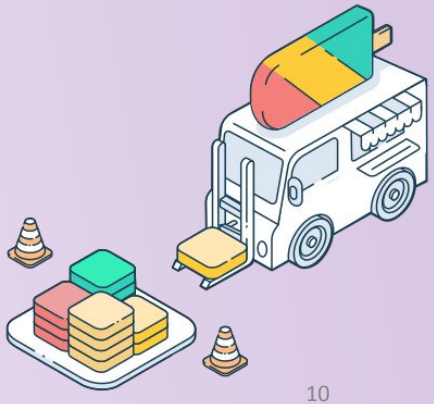
- Developers need to request infra & resources
  - We need an **API**
- Our API could be very **granular** - expose all the things
  - Buckets, VPCs, VMs, IAM Role, etc.
- Devs get what they need and into production **fast**!



Implementation w/ Crossplane

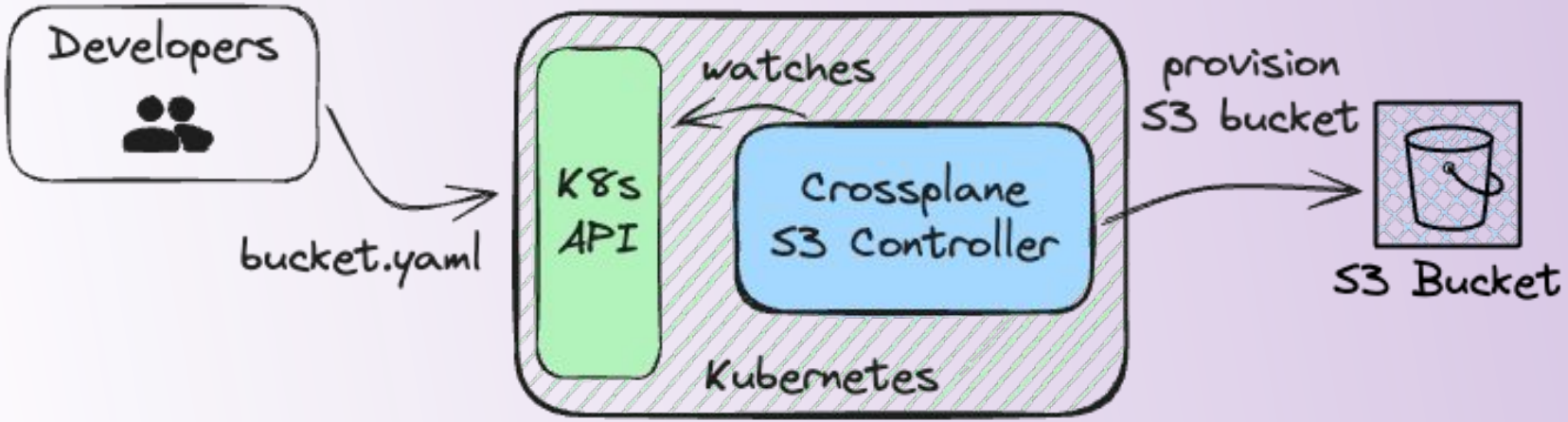
- CNCF open source project
- Extends Kubernetes to manage all your resources
  - **Universal** control plane, unifying all cloud providers & environments
- Each resource is exposed as a CRD in K8s API

```yaml
apiVersion: s3.aws.upbound.io/v1beta1
kind: Bucket
metadata:
  name: cool-bucket
spec:
  forProvider:
    region: us-west-1
    tags:
      team: core
      env: prod
```



Implementation w/ Crossplane

- Desired state is reconciled with actual state in real world



This is NOT a Platform…

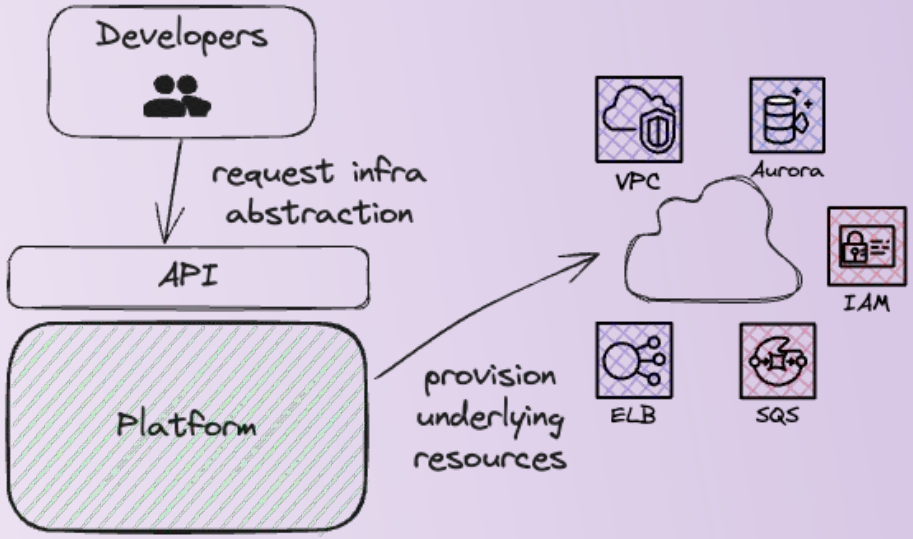
- …at least not a good one!
- No abstraction - everything is exposed
  - forces devs to understand all options & complexity
  - no separation of concerns between platform and devs
- Requires write permissions to low level resources
  - May as well give your devs AWS console access 🤪
- No custom logic or variable configuration
- Not great for building a platform

Step 2: A Real Platform

2020 - Crossplane Compositions

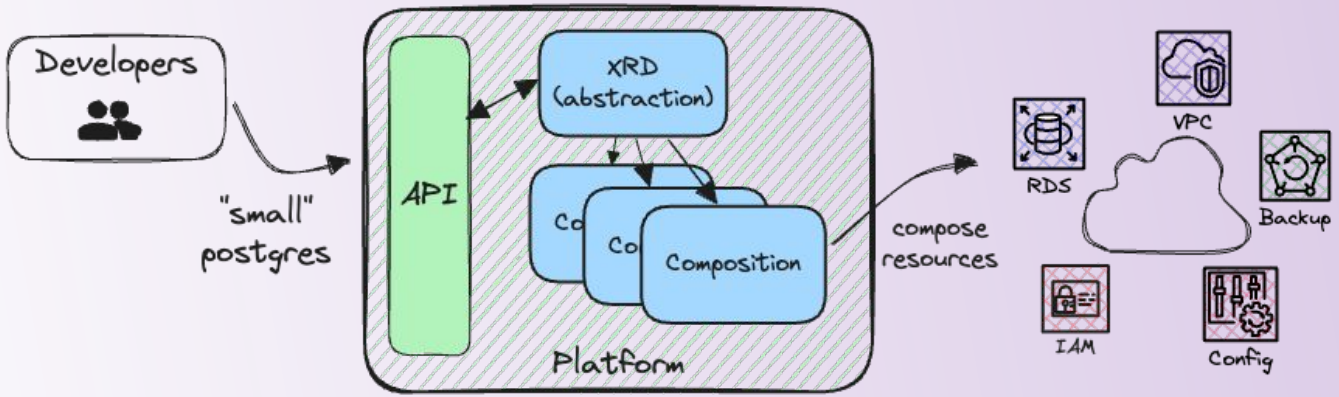
A real platform would …

- Utilize best practices
- Provide compliance by default
- Hide infrastructure complexity
- Provide simple config options
- Reduce manual steps required by devs to setup everything



Compose resources with Compositions

- “Patch-and-Transform” framework
- Generate K8s manifests based on user input and default values
- Crossplane takes care of create, update and delete



Finally, an actual platform

- Universal
- Simple & Abstract

… but we still hit limits

- “Patch-and-transform” is a very limited framework
  - Declarative approach not always possible
  - Can only copy data between input and output resources
- Basically a black box - very hard to debug

Step 3: Higher Level Logic

2023- The previously impossible is now possible

We need more…

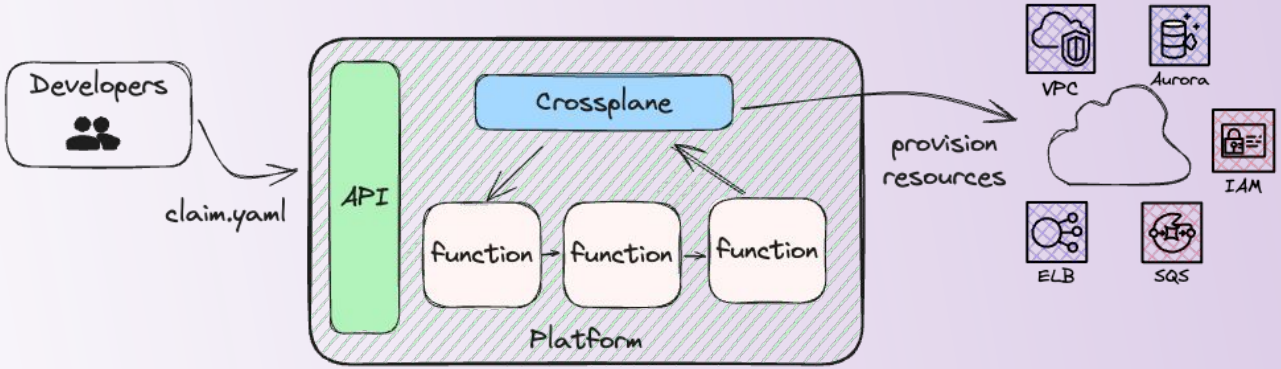
- We have a real platform w/ a real API
  - it’s complex to use but it doesn’t do complex things yet
- We need more advanced logic
  - flow control, conditionals, templating, etc.
  - composition is not a programming language
- We should also escape static YAML hell while we’re at it…

Use case

- Good for an Ops focused platform team that wants
  - a mostly declarative experience
  - to express more complex config and logic without full programming
  - to use a config language they specialize in
- no code → low code → **medium code** → full code

Crossplane Functions

- Run a pipeline of simple functions to compose resources
- Use your language of choice for your unique logic
  - helm/go templating, CUE, KCL, PKL, CEL, etc.
- Let Crossplane do the heavy lifting of CRUD-ing resources, reconciling, finalizers, owner refs, etc.



function-go-templating

```
pipeline:
- step: render-templates
  functionRef:
    name: function-go-templating
  input:
    apiVersion: gotemplating.fn.crossplane.io/v1beta1
    kind: GoTemplate
    inline:
      template: |
        {{- range $i := until ( .observed.composite.resource.spec.count | int ) }}
        apiVersion: iam.aws.upbound.io/v1beta1
        kind: AccessKey
        spec:
          forProvider:
            userSelector:
              matchLabels:
                crossplane.io/name: user-{{ $i }}
        {{- end }}
```

function-kcl

```
pipeline:
- step: render-instances
  functionRef:
    name: kcl-function
  input:
    apiVersion: krm.kcl.dev/v1alpha1
    kind: KCLInput
    spec:
      source: |
        regions = ["us-east-1", "us-east-2"]
        items = [{
            apiVersion: "ec2.aws.upbound.io/v1beta1"
            kind: "Instance"
            metadata.name = "instance-" + r
            spec.forProvider: {
                ami: "ami-0d9858aa3c6322f73"
                instanceType: "t2.micro"
                region: r
            }
        } for r in regions]
```

Thanks @Peefy!

function-cue

```
// if a base ARN exusts, render a policy with all ARNs.
if baseARN != "unknown" {
    let allTuples = list.Concat([
        [baseARN, baseARN + "/*"],
        [
            for a in additionalARNs {[a, a + "/*"]},
        ],
    ])
    let allResources = list.FlattenN( allTuples, 1)
    response: desired: resources: iam_policy: resource: {
        apiVersion: "iam.aws.upbound.io/v1beta1"
        kind:       "Policy"
        metadata: {
            name: "\(compName)-access-policy"
        }
        spec: {
            forProvider: {
                path: "/"
                policy: json.Marshal({
                    Version: "2012-10-17"
                    Statement: [
                        {
                            Sid: "S3BucketAccess"
                            Action: [
                                "s3:GetObject",
                                "s3:PutObject",
                            ]
                            Effect:   "Allow"
                            Resource: allResources
                        },
                    ]
                })
            }
        }
    }
}
```

Thanks @gotwarlost!

Benefits of Functions

- Start using high level languages of your choice
- Get more logic, flexibility, and expressiveness
- Choose the language/UX your team is most comfortable with
  - Mix and match languages if needed
- Lots of supported languages, community keeps building!
- Bonus: Earlier testing and validation
  - Rapid platform dev/testing feedback loop

Step 4: Full Code - Full Power

2024 - General Purpose Programming Languages

Certain use cases require a full scale programming language

- Multi-step setups (i.e. tenant onboarding)
- Complex transformations
- Easy code sharing between components
- Ability to unit tests certain features
- … or developers just prefer Go over KCL, CUE or YAML

Custom functions in Go

- Full code - full power - full complexity
- Import CRD structs from Crossplane providers
  - Static code validation ensured by Go compiler
- Access the whole Go toolchain
  - Linters
  - Unit tests
  - Code generators (Kubebuilder etc.)

Functions are modular!

- We don’t have to write everything in Go

E2E Tests in Go

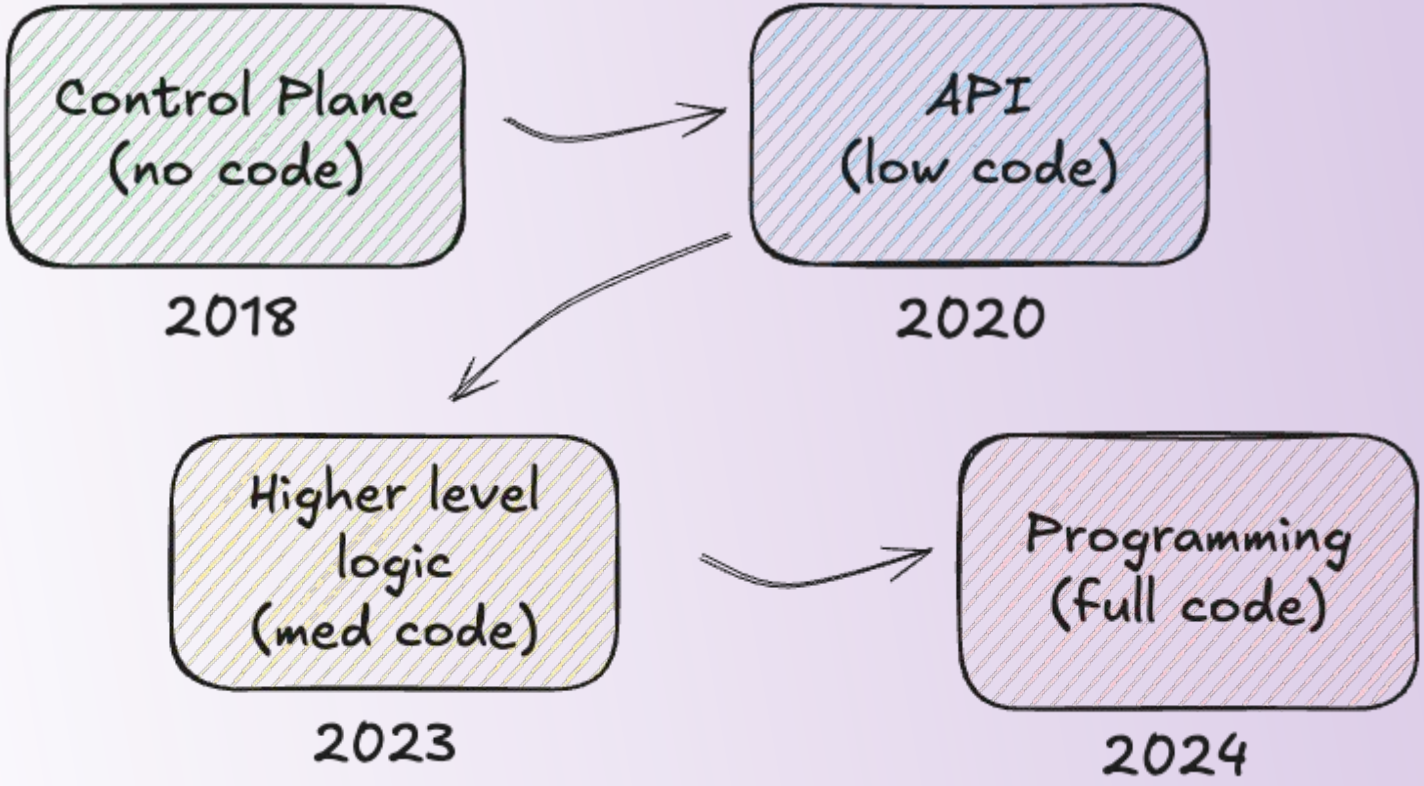
Treat your platform as a real software project

Replay full real-life use cases as Go tests before every rollout

1. Create resource claims
2. Wait until resources get ready
3. Access external system (like a real user)
4. Work with the external system (like a real user)

Platform Journey in Summary

Let’s review…



API that is Universal, Declarative, Simple …and Powerful

Links for further reference

Crossplane:

- Website: crossplane.io
- GitHub: github.com/crossplane/crossplane
- Slack: slack.crossplane.io

accenture

maximilian.blatt@accenture.com

Slack: @mistermx