KubeCon | CloudNativeCon

North America 2021

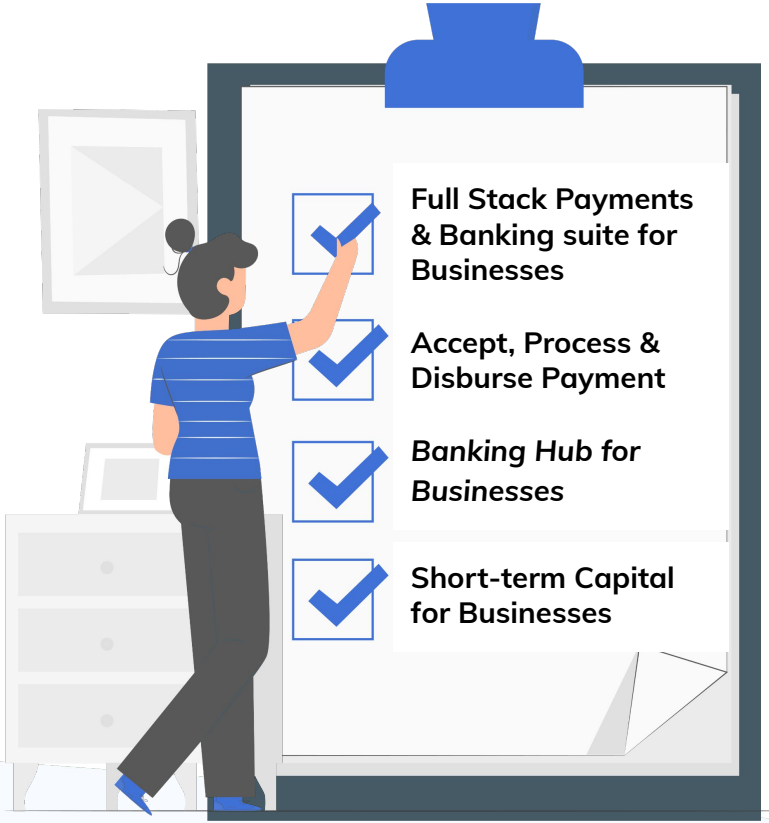
Improving Developer Experience: How We Built a Cloud Native Dev Stack for 100s of Engineers

- *Srinidhi S*
  Senior Software Engineer
  @ Razorpay

- *Venkatesan Vaidyanathan*
  Senior Architect
  @ Razorpay

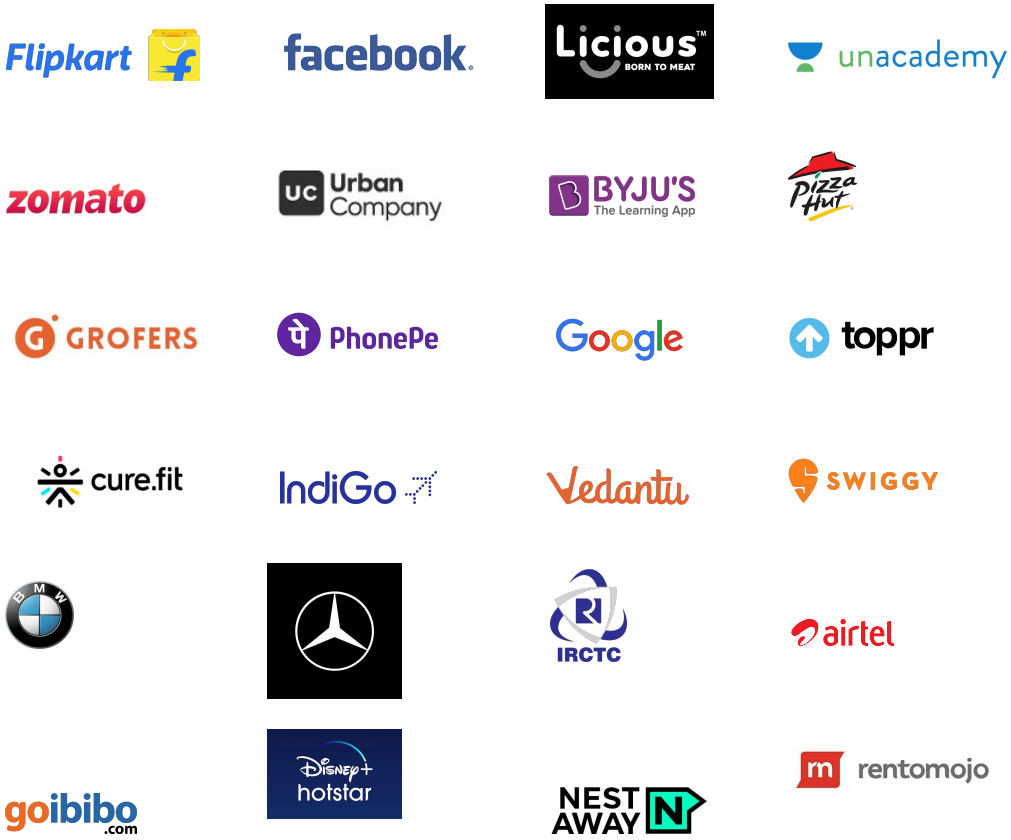
Who are we?

Transforming the world of business finance with insight, intelligence and innovation



Who uses Razorpay?

Powering payments for over 10 lakh + businesses



Our Growth Metrics

10x Growth

10x growth in Headcount in last 4 years

Scaling Teams

Full-Fledged Pods & BUs (4 BUs, 30+ Pods)

100+ Microservices

50+ micro services in the last 2 years and growing

3 Acquisition

3 companies in the last 3 years

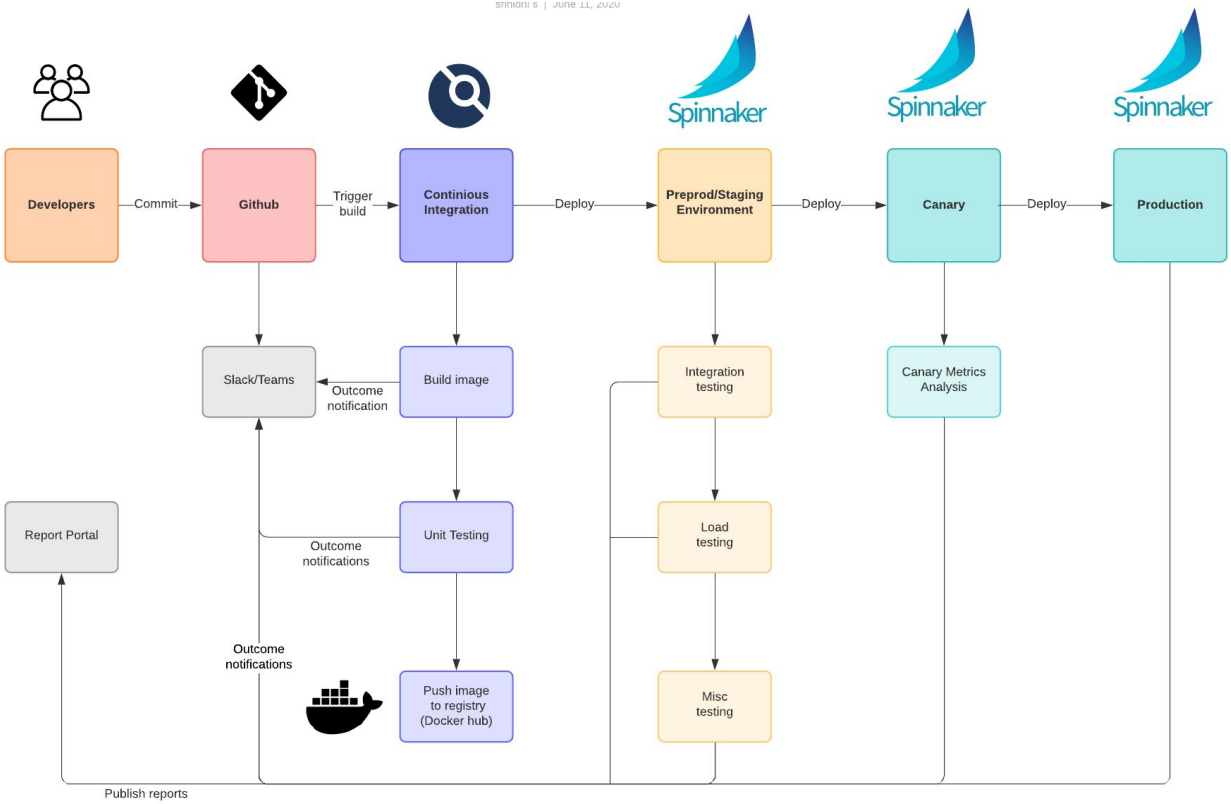
Polyglot Stack

Php, Golang, Python, Java, Nodejs, Scala

Deployments

#1500 of deployments per month

Overview of our CI / CD Practices



Tools :

- AWS
- EKS
- Helm
- Spinnaker
- Atlantis + Terraform
- Github Actions

Development Challenges

Deep Dive into agility: Problems that hinder dev productivity

Development Challenges

Development Dependencies

Increased feature roll-out time due to sequential deployment

- Adding multiple services and multiple dev's exponentially increases the pain in deployment
- Cloud resources mocked for local development reduces confidence in cloud environment
- Testing local changes on cloud requires full blown deployment process

Shared Environment

Scaling Shared Environments with engineering and service growth

- Scaling shared environments for 100's of developers and 100's of services magnifies the challenge
- Demoing feature to product and business is painful because of shared environment concerns
- Requirement of separate dev environments for integration with partners and gateways in a secure fashion

Development Challenges

Infra Provisioning

Increased overhead of provisioning on engineers & training

- Infra components like SQS / SNS etc requires significant cognitive load understanding infra than focusing on core tasks
- Trivial debugging requires checking logs on logging platform breaking the developer workflow

Build & Deploy

Containerization , CI workflows add additional time per feature iteration

- The code build time via docker containers are high and goes upto several minutes sometimes.
- Testing even a small change in the feature needs the same loop to be iterated

Finally.....

All of these problems waste a lot of a developer time ranging from an hour to days; time that could be utilized for doing productive things

Need of the hour

Simplify developer workflow and
reduce time to rollout features
independently

Goals

Reduce cognitive load on developers

Streamlined workflow
Streamlined Workflow & Faster Iterations to merge to master

Environment consistency
(Dev, Stage, Production)

Faster Feedback
Faster feedback loop from local development

Key Decision Factors

- Remove vendor lock-in (rely on OSS practically)
- Lesser learning curve ; no extra DSL
- Kubernetes Native - Our environment is K8s anyway
- Hassle free onboarding - minimal changes to application and development lifecycle
- Adhere to polyglot environment
- Cost effective - Eventually be able to bill business around dev lifecycle
- Slightly Opinionated - Not a PaaS

Codename: Devstack

Journey towards a **better** development lifecycle

Q1: How do we bypass the CI/CD loop for iterative development?

Need: Ability to directly expose local code on internal cloud environment

Solution: Leverage Telepresence(Proxy based Approach)

Details:

- Client for two way proxy for requests
- DNS resolution for kubernetes resources
- Mount volumes and network via telepresence CLI

Drawbacks :

- Slow network - The performance of the system is limited by telepresence's tunnelling
- Major connectivity issues with database
- Issues with volume mounts
- Network bottlenecks further over VPN

Q2 : How do we overcome network limitations?

Need: Ship code to remote container without tunneling

Solution: Devspace (File sync based approach)

Details:

- Devspace allows us to develop software directly in the kubernetes cluster via multiple dev options
- Fast + Reliable File Synchronization to keep all files in sync between your local workspace and your containers
- Port Forwarding that lets you access services and pods on localhost and allows you to attach debuggers with ease
- Live Reloading via container restart

Limitations :

- Container restarts are bound by kubernetes probes - Need something better and faster

```yaml
version: v1beta10
dev:
  replacePods:
    - labelSelector:
        app: ${APP_NAME}-${DEVSTACK_LABEL}
      replaceImage: docker/rzp:api-devstack
      namespace: ${NAMESPACE}
  sync:
    - labelSelector:
        app: ${APP_NAME}-${DEVSTACK_LABEL}
      localSubPath: ./
      containerPath: /app
      excludePaths:
      - .github/
      - .devspace/
      namespace: ${NAMESPACE}
      disableDownload: true
      initialSync: preferLocal
vars:
  - name: DEVSTACK_LABEL
    value: "srinidhi"
  - name: NAMESPACE
    value: "srinidhi"
```

devspace.yaml

Q3: How can we avoid container restarts?

Need: Faster app reloading without restarts

Solution: Hot reloading code inside the container

- Improvised the hot reloading introduction CompileDaemon in the container for go apps , Nodemon in nodejs ...
- CompileDaemon swiftly watches for the files changed via sync , rebuilds the binary and runs the new one
- A generic docker container per language built and used on every devspace command instead of the actual container

```
####################################
FROM golang:1.15-alpine as builder

RUN apk add git && apk add curl
RUN apk add make git tzdata

RUN mkdir -p /app
WORKDIR /app

COPY go.mod .
COPY go.sum .

RUN go mod download

RUN go get github.com/githubnemo/CompileDaemon

ADD . /app/

ENTRYPOINT CompileDaemon --build="go build -o /app/bin/api cmd/api/main.go"
                         --command="/app/bin/api"
```

Dockerfile

Q4: How do we orchestrate multiple services?

Need: Declaratively define and apply service dependencies

Solution: Helmfile : Wrapper on top of Helm

- Allows you to compose several charts together to create a comprehensive deployment artifact for anything from a single application to your entire infrastructure stack.
- Works seamlessly with our existing helm packages , No extra code needed
- Easier configuration via a single yaml file
- Support for hooks making application lifecycle management easier

```
# Helm defaults that cleans up on failure
helmDefaults:
  cleanupOnFail: true
  timeout: 600
environments:
  default:
   values:
# The devstack lable can be anything ranging from devname , Project Name , JIRA id , PR number
# this is the key on which the ephemeral components are created
   - devstack_label: srinidhi124
# ttl is the time to live for all the objects in minutes , default being 360(6 hrs)
   - ttl: 6h
releases:
# this contains the list of applications required for development based on the usecase
- name: dashboard-{{ .Values.devstack_label }}
  namespace: dashboard
  chart: ./charts/dashboard
  values:
    - image: 3bb181628867137ed2e50ce3e5a3cf7b6888ae72
    - devstack_label: {{ .Values.devstack_label }}
    - ttl: {{ .Values.ttl }}
    - user: {{$user}}

- name: api-{{ .Values.devstack_label }}
  namespace: api
  chart: ./charts/api
  values:
    - image: 9632a363e602d0c2f01d43b1512004813760aa4a
    - devstack_label: {{ .Values.devstack_label }}
    - ttl: {{ .Values.ttl }}
    - base_env: beta
    - user: {{$user}}
```

helmfile

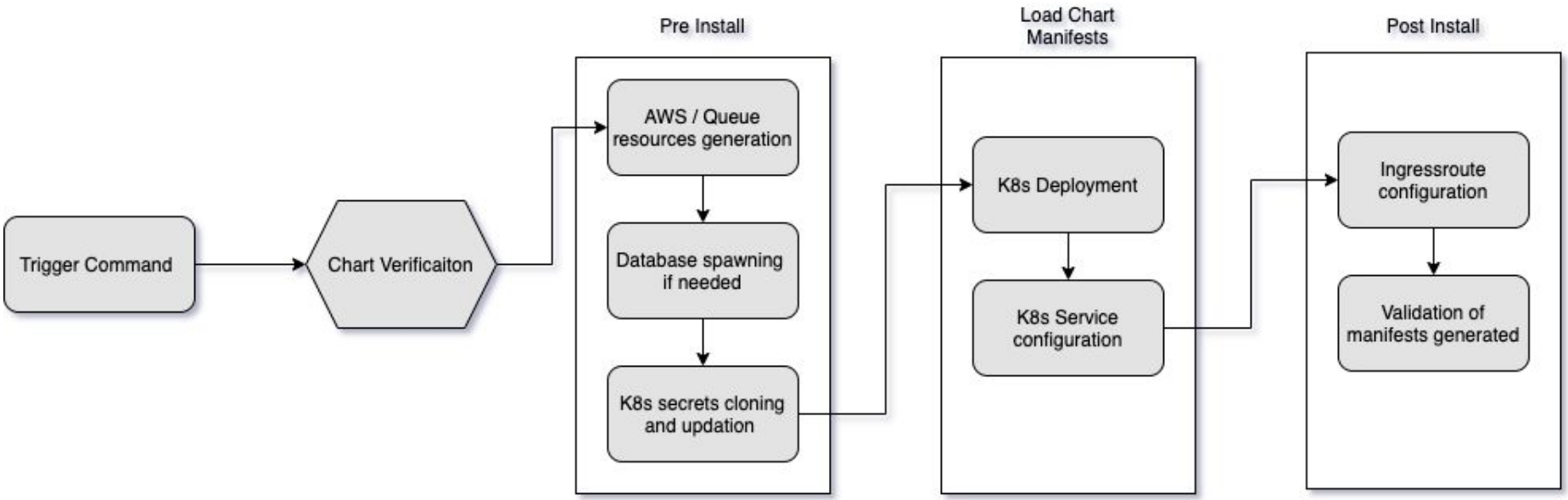
Q5: How do we provision ephemeral infrastructure resources

Need: Reduce infrastructure provisioning overhead for Cloud Components

Solution: Helm Hooks(Provisioning) and Localstack (mocking AWS components)

- Hooks provide a plug and play model to maintain the dynamics of applications auxiliary requirements like databases , AWS resources , Kafka queues etc
- Application helm template just had to specify the requirements based on which the corresponding hooks where triggered to provide ephemeral stack
- Management of AWS resources is done through localstack without depending on actual AWS cloud resources and terraform

Helmfile Workflow :



Q6: How do we handle routing?

Need: Route traffic to the right user service

Solution: Header based routing - Traefik 2.0

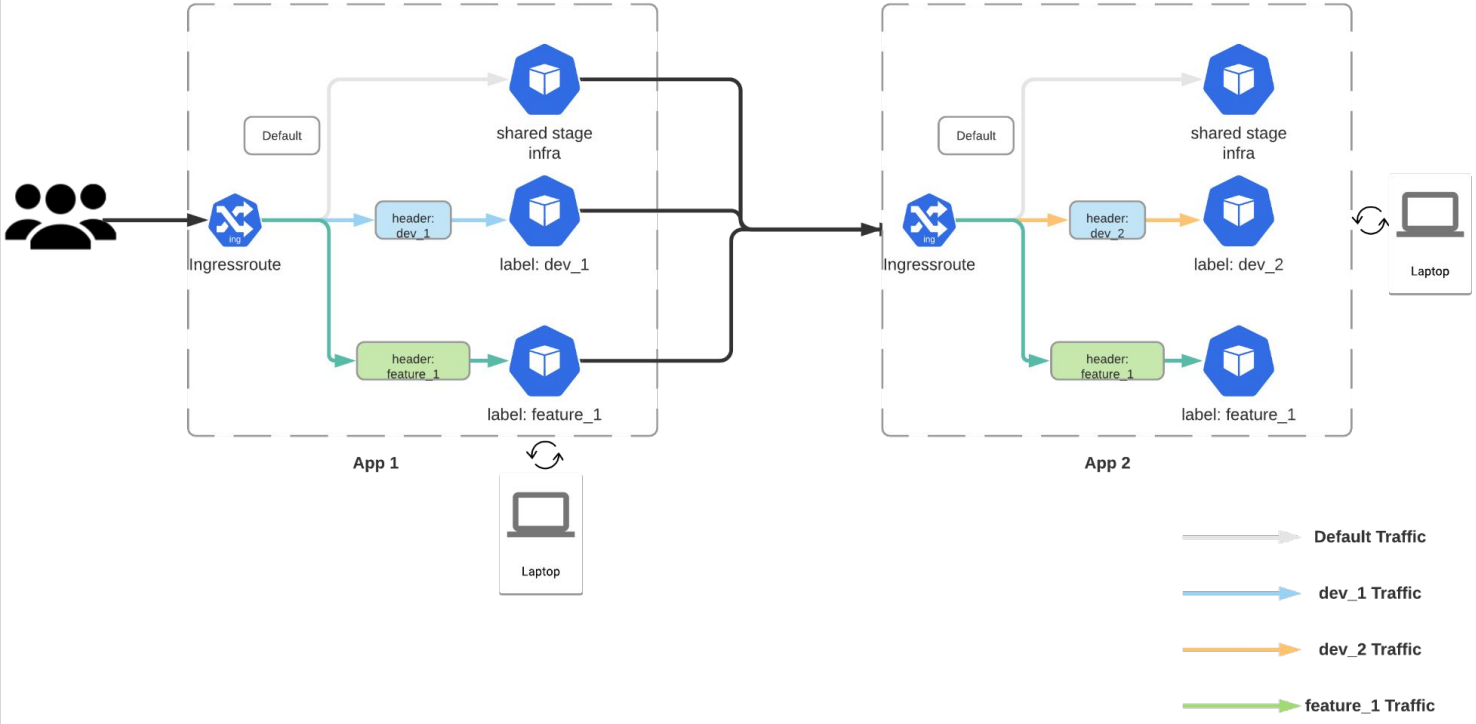
- Each user supplies a specific independent header for their “service fleet”
- The services can be accessed by just adding the header to the request
- Configuration via helm hooks to update the ingressroute object

Need: Route traffic to the right upstream service

Solution: Header Propagation - Opentracing to the rescue

- Piggybacking on the baggage headers of distributed tracing to ensure header based routing is intact across the ecosystem
- Helped in application onboarding without changing a line of code in the application

Routing Overview



Demo

Additional Features

Preview URL

- Dedicated ingressroute / URL per service based on the devstack label
- A traefik middleware that injects the appropriate header for right upstream communication
- All of these bundled as a part of helmhook by default for every application

```yaml
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  annotations:
    janitor/ttl: 6h
  name: api-srinidhi
  namespace: srinidhi
spec:
  entryPoints:
  - http
  routes:
  - kind: Rule
    match: Host(`api-web-srinidhi.dev.razorpay.in`)
    middlewares:
    - name: injectheader-srinidhi
    services:
    - name: api-srinidhi
      port: 80
---
apiVersion: traefik.containo.us/v1alpha1
kind: Middleware
metadata:
  annotations:
    janitor/ttl: 6h
  name: injectheader-srinidhi
  namespace: srinidhi
spec:
  headers:
    customRequestHeaders:
      rzpctx-dev-serve-user: srinidhi
```

ingress.yaml

Ephemeral Databases

- Support for different types of database config
  - Local database : Database running in developer local machine with DB requests reverse proxied
  - Ephemeral database: Database spawned per devstack instance with preseeded data , schema being synced and the specific dev migrations run
  - Persistent database : Databases that are pre-existing and maintained via regular dataops flow
- The workflow for ephemeral database being :
  - Copy of a stable stage environment periodically backed up in S3 acting as base
  - Data container based on this base with migrations run isolated from others

```
version: v1beta10
dev:
  ports:
    - labelSelector:
        app: ${APP_NAME}-${DEVSTACK_LABEL}
      reverseForward:
      - port: 3306
        remotePort: 3306
vars:
  - name: DEVSTACK_LABEL
    value: "srinidhi-db"
  - name: NAMESPACE
    value: "srinidhi"
```

devspace.yaml

Cost ... really looking into that ?

KubeJanitor

- Taking care of the cleaning up of resources
- Janitor based on annotation cleans up the resources periodically
- Dev configurable ttl with default set to 6 hours

```
metadata:
  annotations:
    janitor/ttl: 6h
```

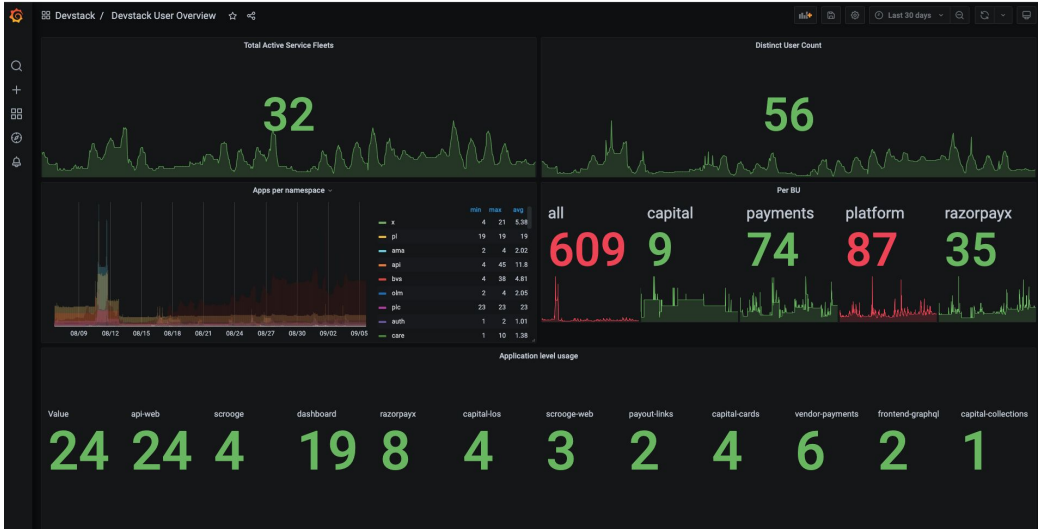
Deployment annotation

Cluster Auto-Scaler

- Spot nodes for reducing overall cost with labelling per business unit
- CA(+HPA) to spin up nodes on demand based on pod lifecycle

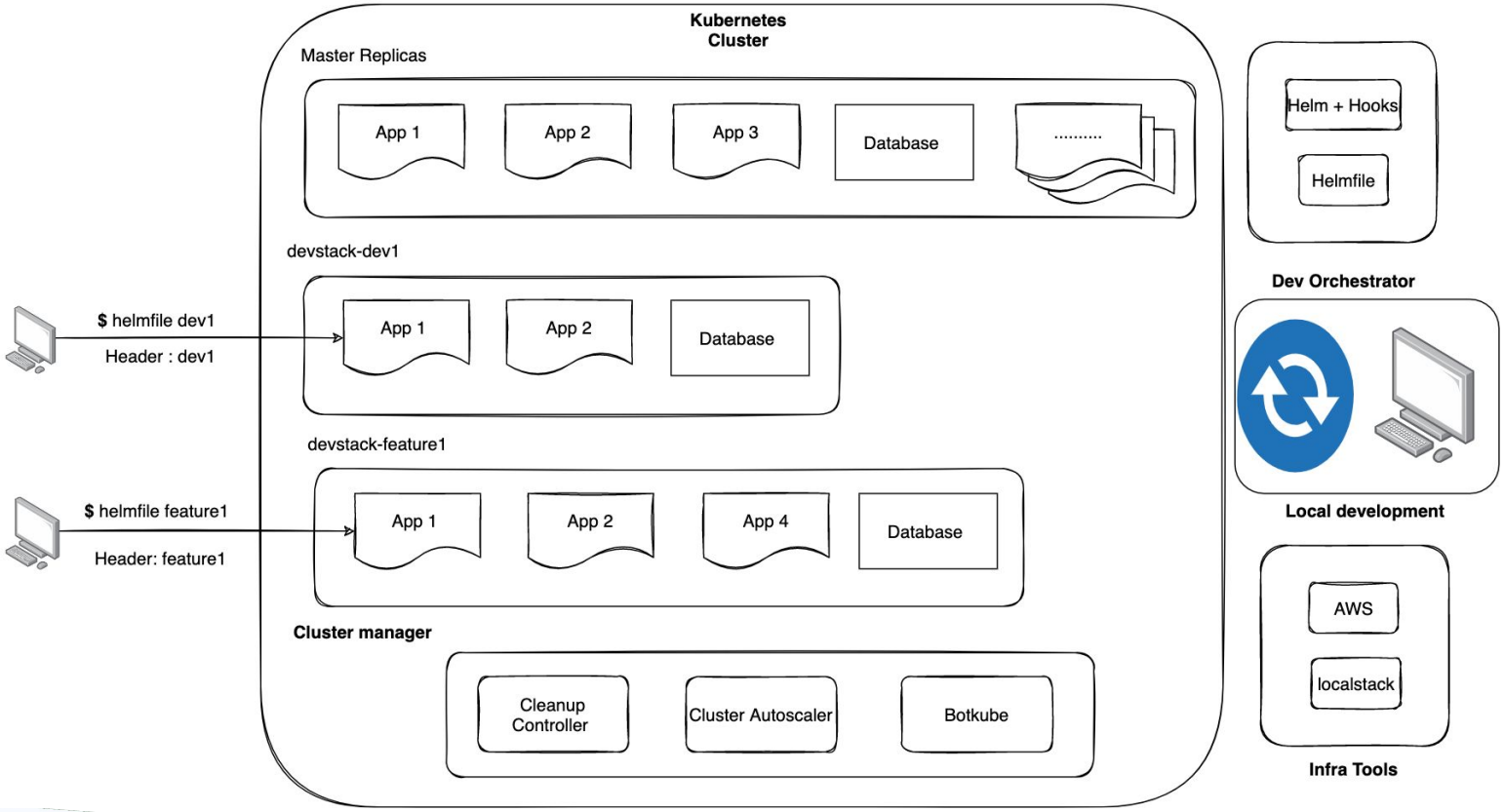
Prometheus Monitoring

- Using labels to measure the overall usage across the apps , BU's , users
- Getting insightful data on the reservations times , feature release time etc

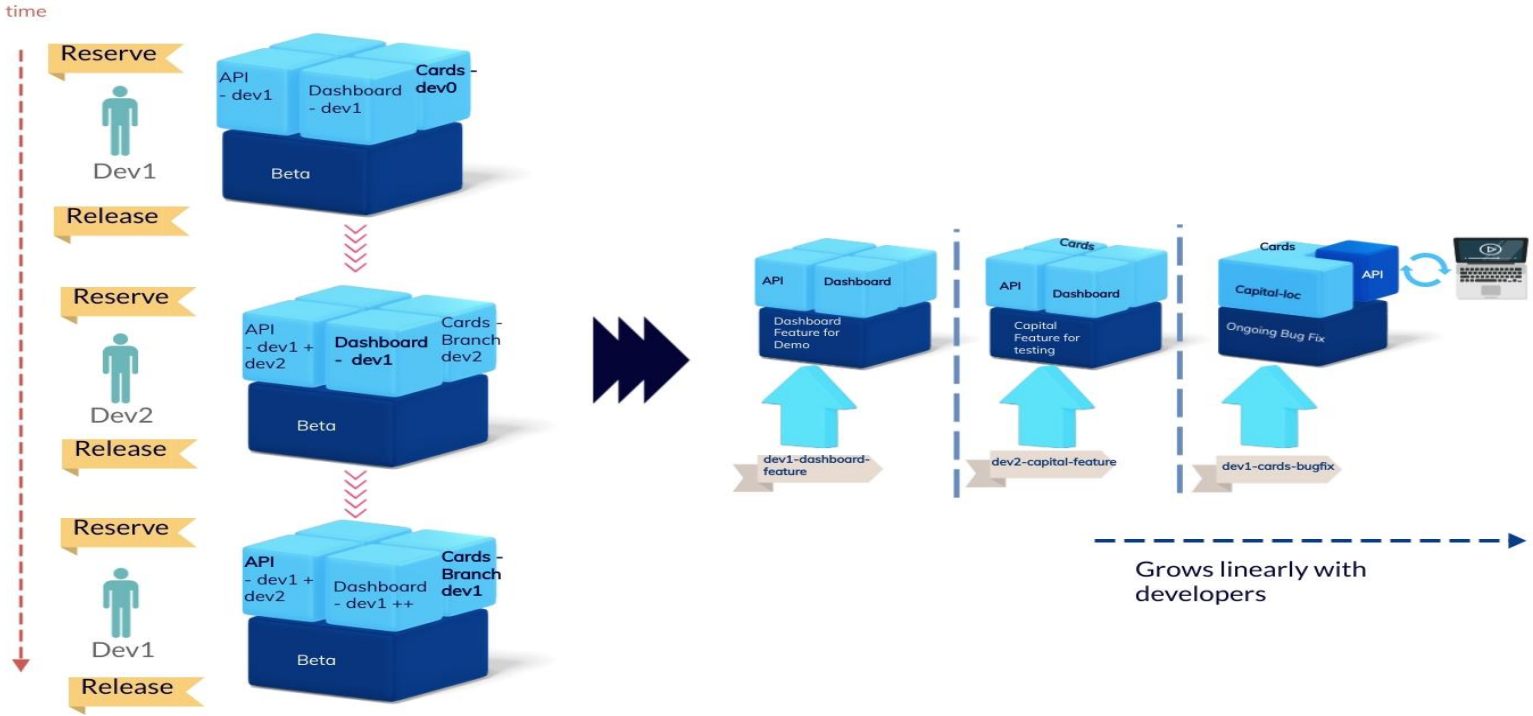


Grafana Dashboard

Solution Overview



Current vs DevStack



Impact on Dev productivity

10-15X reduction in time to take feature live

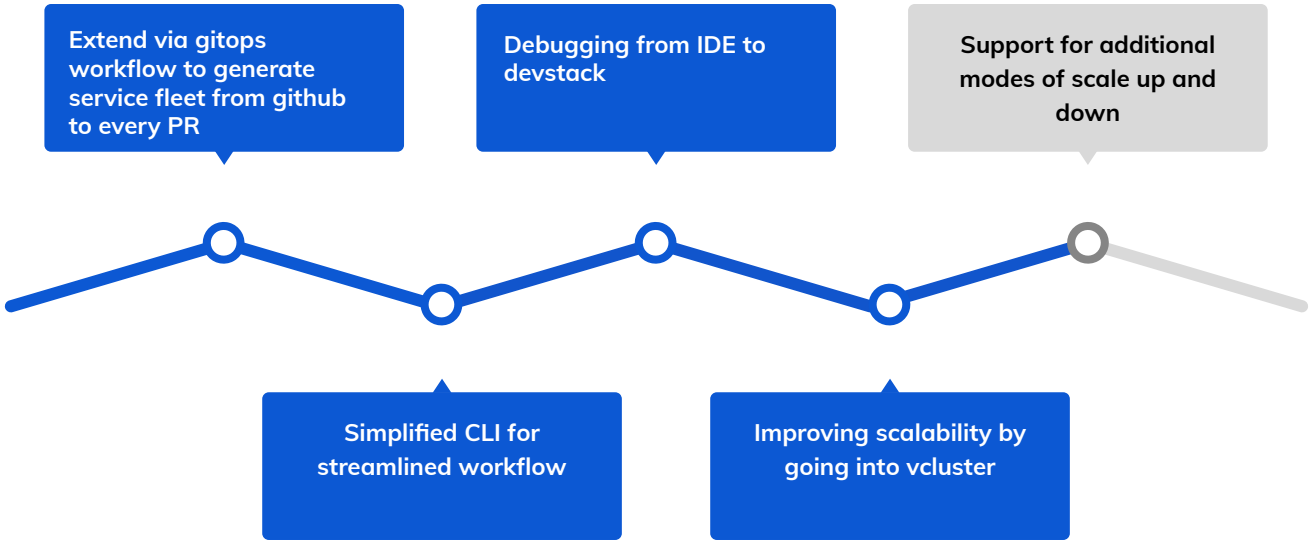
Per Iteration

- 2 min vs 20-40 min earlier
- Devstack: App Hot reloading time - 2mins
- Current: Container build(5min), Deployment (5min), debugging (20min)

Per feature (8/10 iterations)

- 30 mins vs 5 Hours

Road Ahead


Extend via gitops workflow to generate service fleet from github to every PR
Debugging from IDE to devstack
Support for additional modes of scale up and down
Simplified CLI for streamlined workflow
Improving scalability by going into vcluster

Open Source?

Devstack Repo : github.com/razorpay/devstack

Thank you !!!

Questions ?