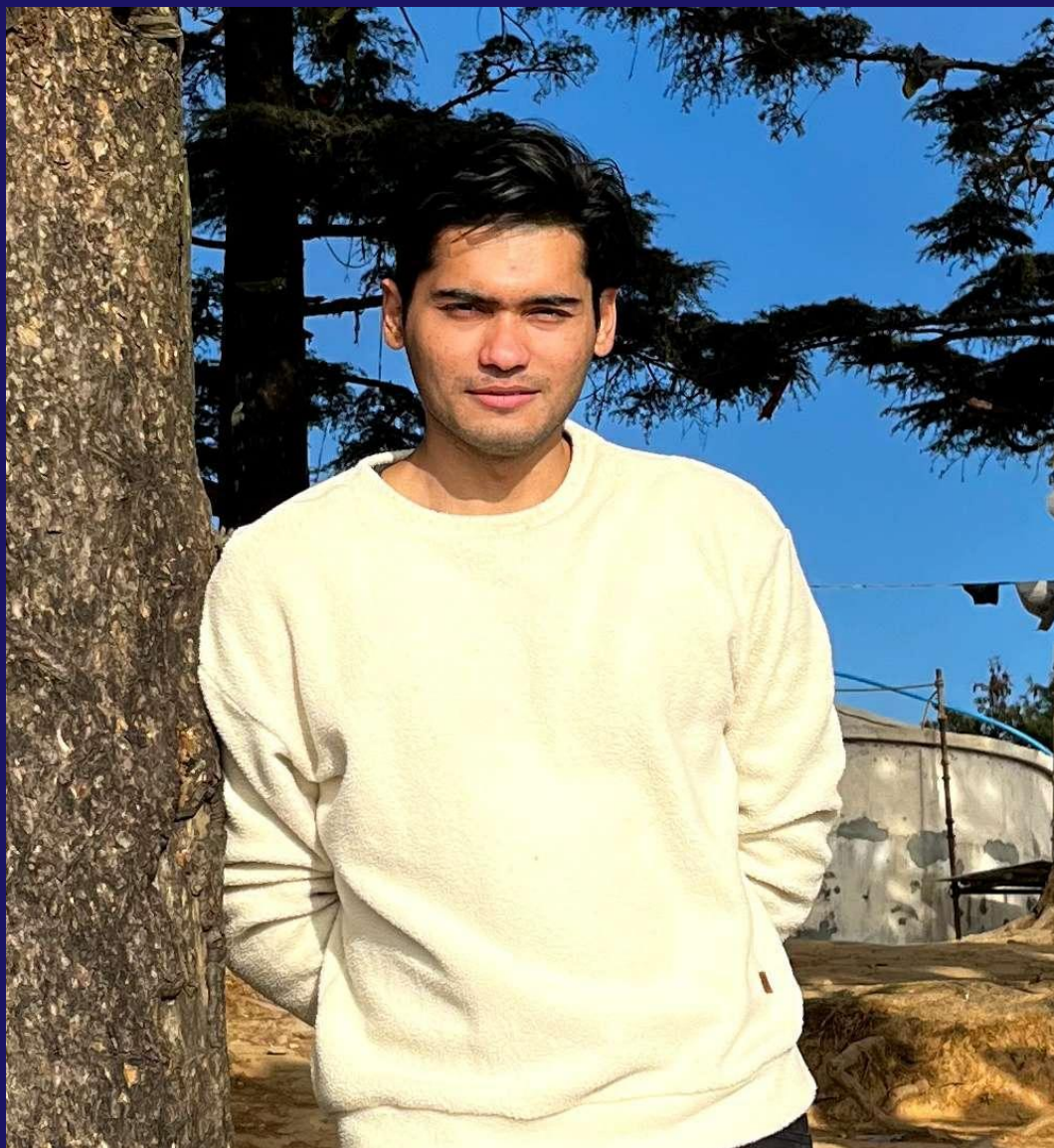


# Beyond Manual Fixes: How Devtron Implements Auto Remediation in Kubernetes





# Hey, I'm Prakash

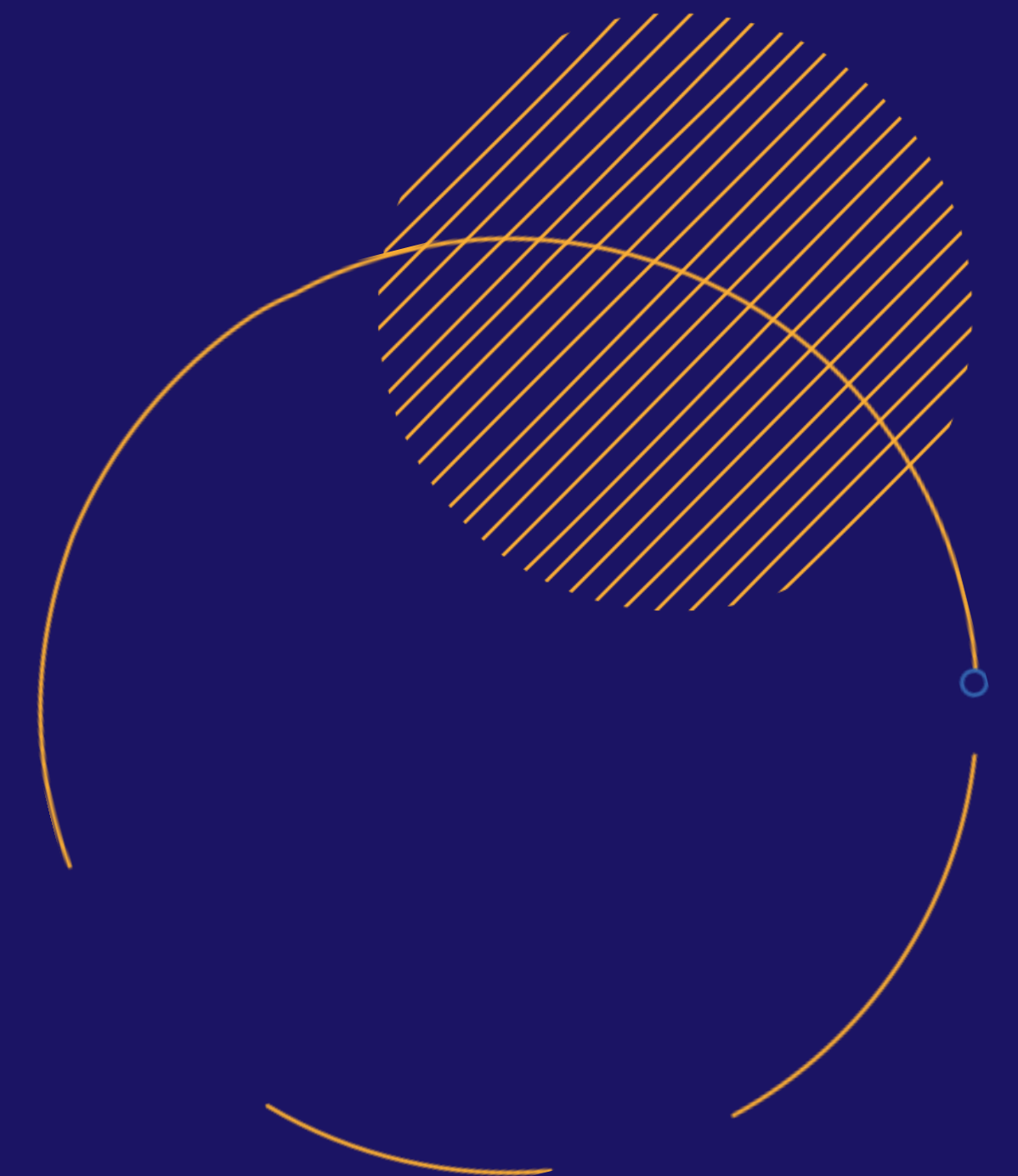


**Software Development Engineer @ Devtron Inc.**

**Let's Connect** 

**GitHub: @prakash100198**

**LinkedIn: [linkedin.com/in/prakash100198](https://www.linkedin.com/in/prakash100198)**



- What is Auto Remediation?
- We'll walkover some problem statements
- Why do we need Auto Remediation?
- Challenges(with examples) in implementing Auto Remediation in any system/software
- A Demo with Devtron's UI on remediating an issue
- A Bird's Eye view of how we implement Auto Remediation with challenges in that approach
- Heated moments with clients and how we resolved those bottlenecks.

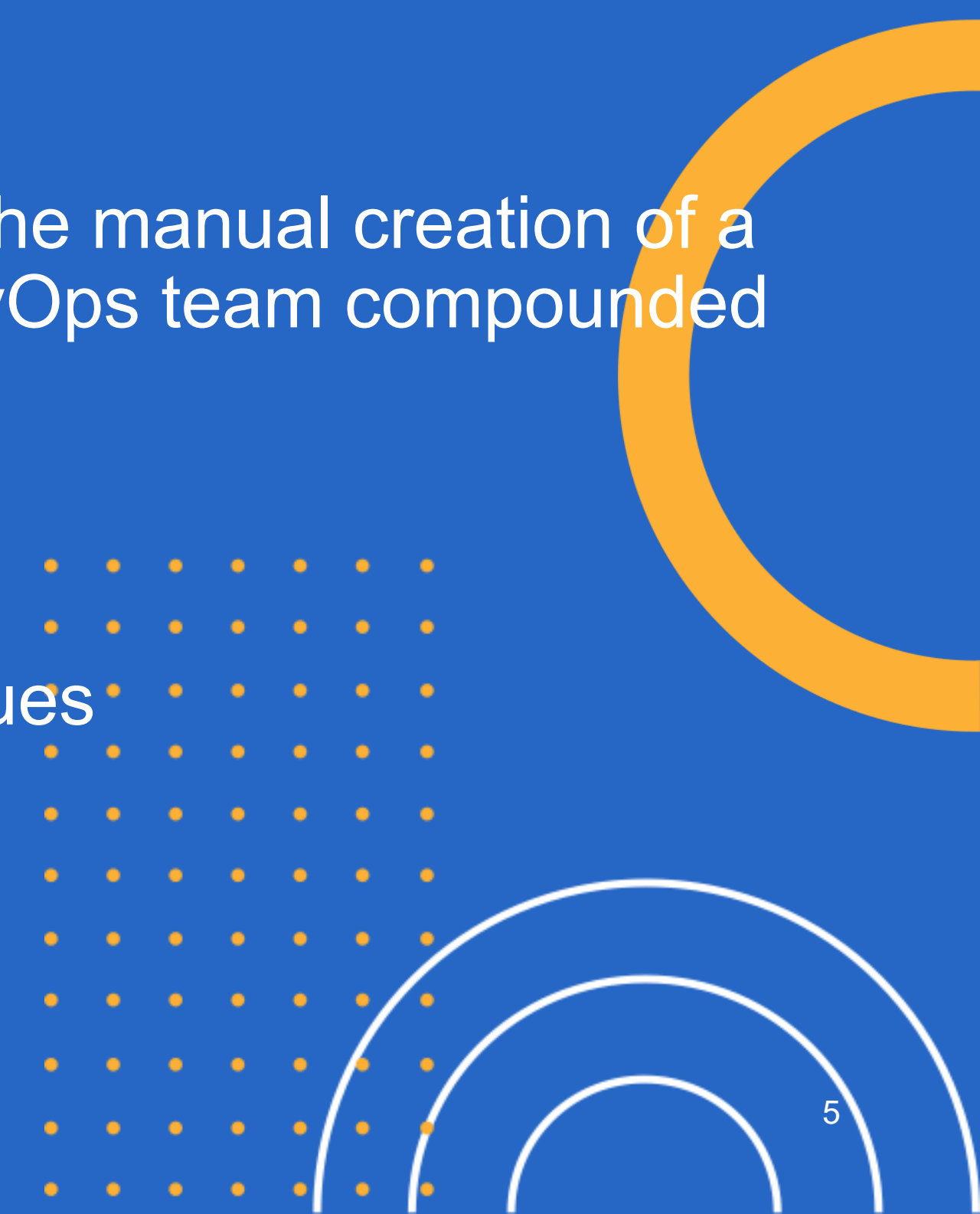
# What is **Auto Remediation**?

- Auto remediation is an approach to automation that responds to events with automations able to fix, or remediate, underlying conditions. It can takes actions in a series of steps where each step completion or failure can be a prerequisite for the next step.

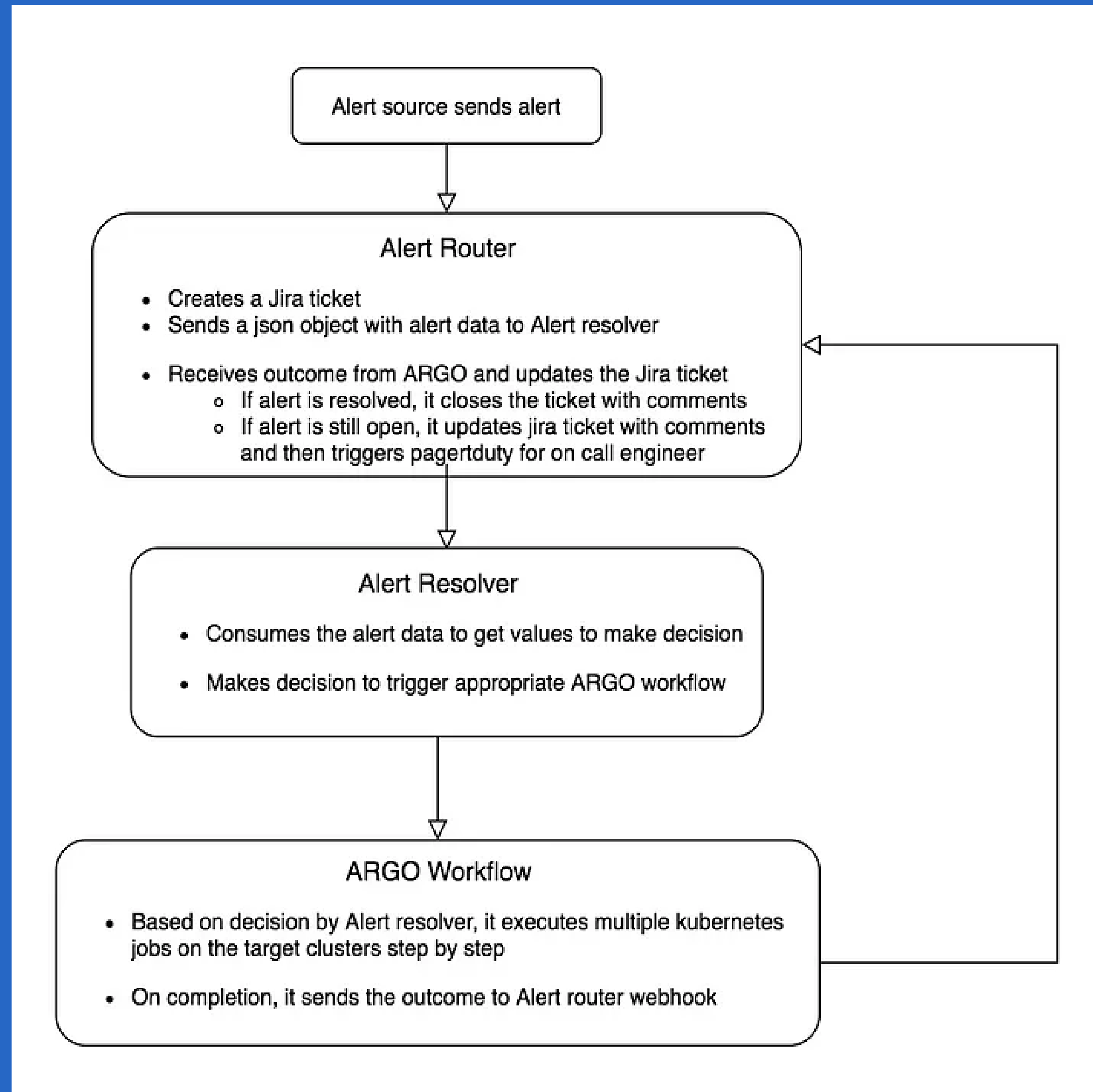


# Problem Statements

A thick, hand-painted style orange brushstroke that underlines the title and extends across the top of the slide.

- Imagine you get an error event in your monitoring stack, you know the fix and you start implementing the solution, but till then you're done with the work that has already impacted your business. The delay in resolving these incidents can lead to prolonged service disruptions, revenue loss, and customer dissatisfaction.
  - It ties up valuable engineering resources that could be better utilized for other tasks.
  - A node that is running out of CPU.
  - In the event of multiple node failures, leveraging an EKS configuration file enables the manual creation of a new EKS cluster to redirect incoming traffic. However, delayed detection by the DevOps team compounded with the 10-minute duration for cluster creation results in substantial downtime.
  - When system resources are overwhelmed due to sudden spikes in traffic.
  - Bottlenecks in real-time, such as database query slowdowns or network latency issues
- etc..
- 
- Decorative elements in the bottom right corner: a large orange arc, a grid of small orange dots, and three concentric white arcs.

Here's a diagram on how industry wide Auto Remediation process looks like.



# Why do we need Auto Remediation?



To address above issues, we need an automated remediation system that can detect and resolve known error events without human intervention. By implementing an automated remediation system, we aim to:

- Reduce the mean time to resolution (MTTR) for known error events, minimizing the impact on business operations and customer experience.
- Free up engineering resources by automating routine remediation tasks, allowing them to focus on more complex issues and strategic initiatives.
- Improve operational efficiency and reduce the potential for human error during incident response.
- Enhance visibility and auditability of incident response processes through detailed logging and reporting.
- By automatically addressing issues as they arise, auto remediation minimizes downtime, ensuring continuous availability and performance for users.

# Challenges in implementing Auto Remediation in any system/software

There are three main elements of the remediation process:

1. What k8s event to listen to?
1. What remediation needs to be taken?
1. Making the process transparent and auditable.





# Challenges in implementing Auto Remediation in any system/software

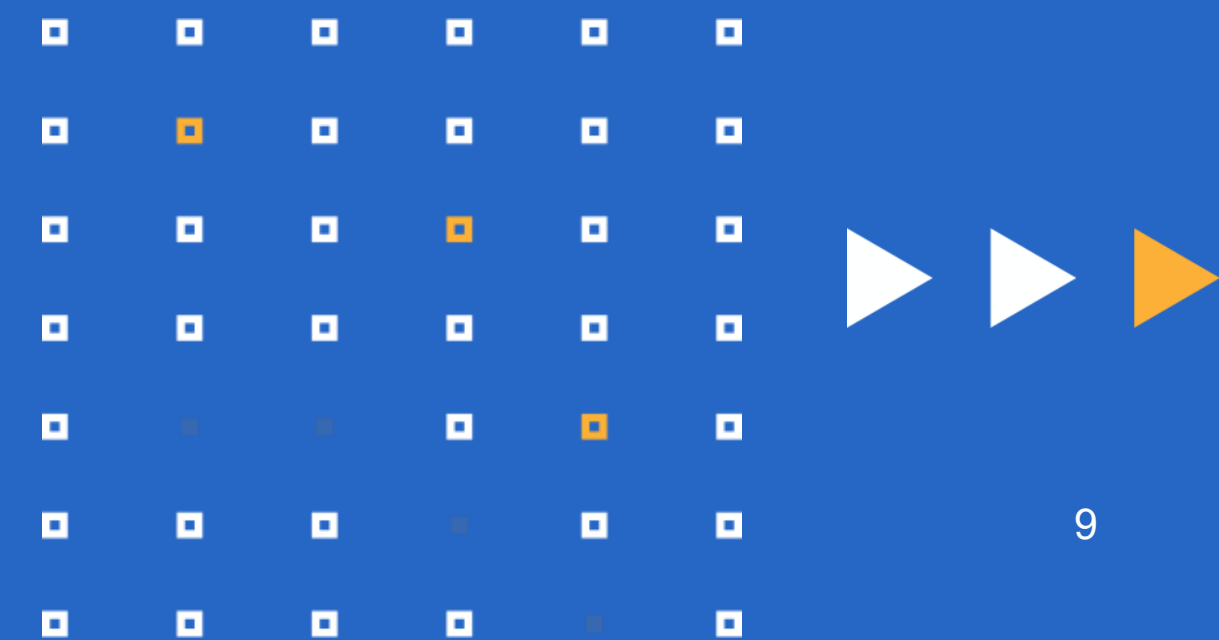
Events could be generated at different lifecycle stages of a kubernetes objects or Devtron, like:

## 1. Kubernetes Events

- Configuration changes (example)
  - ❖ max replica count in HPA increased
  - ❖ Memory increased
- State changes (example)
  - ❖ Failed event
  - ❖ Evicted events

## 1. Devtron events (example)

- CI / CD events
- Vulnerability detected events
- Policy violated events



# Challenges in implementing Auto Remediation in any system/software



All the generated events are not alert or remedy worthy, too many alerts can make your team alert fatigue. In most ideal cases you would like to filter out your events and set up alerts or remedy actions on worthy or critical events only.

For Example: Sending an event whenever a pod is in pending state is a noise as a pod will go in pending state on every deployment. But taking action or alert when a pod was in pending state for the last 5 minutes is worthy. Similarly if your pod is restarting every second, you don't want 600 alerts in the next 10 mins I believe.

A single Kubernetes cluster could include dozens, hundreds or even thousands of nodes, Pods, containers and other objects. Realistically speaking, there is no feasible way of keeping track of all of them and identifying potential issues by hand. Although you can use commands like `kubectl describe` to check the state of various resources manually, that's not a practical means of catching issues when you have a high volume of resources to manage.



# Devtron's UI Demo on Issue Remediation





# A Bird's Eye view of how we implement Auto Remediation with challenges in that approach

- Let's say that the event source is K8s, so previously what we used to do was, we had watchers configured in our code which is always up and always listening to the k8s events, and “we got those events from hitting the k8s api server”.
- Remember the highlighted part this is the major problem that we'll discuss later.
- The picture demonstrates how kubectl get events equivalents to hitting api server for events and what kind of response we get.

```
Terminal — bash — 140x40
sensu $ kubectl get events --field-selector type!=Normal
LAST SEEN   TYPE      REASON   OBJECT                       MESSAGE
28s         Warning   Failed   pod/nginx-6c8997c749-88gj8   Failed to pull image "engine:latest": rpc error: code = Unknown desc = Error response from daemon: pull access denied for engine, repository does not exist or may require 'docker login'
28s         Warning   Failed   pod/nginx-6c8997c749-88gj8   Error: ErrImagePull
44s         Warning   Failed   pod/nginx-6c8997c749-88gj8   Error: ImagePullBackOff
sensu $

sensu $ curl -XGET -s "http://127.0.0.1:8888/api/v1/namespaces/cncf-webinar/events?fieldSelector=type%21%3DNormal" | jq '.items[] | {
>   type: .type,
>   message: .message,
>   reason: .reason,
>   kind: .involvedObject.kind,
>   name: .involvedObject.name
> }'
{
  "type": "Warning",
  "message": "Failed to pull image \"engine:latest\": rpc error: code = Unknown desc = Error response from daemon: pull access denied for e
  nginx, repository does not exist or may require 'docker login'",
  "reason": "Failed",
  "kind": "Pod",
  "name": "nginx-6c8997c749-88gj8"
}
{
  "type": "Warning",
  "message": "Error: ErrImagePull",
  "reason": "Failed",
  "kind": "Pod",
  "name": "nginx-6c8997c749-88gj8"
}
{
  "type": "Warning",
  "message": "Error: ImagePullBackOff",
  "reason": "Failed",
  "kind": "Pod",
  "name": "nginx-6c8997c749-88gj8"
}
sensu $

[0] 0:kube-proxy- 1:webinar* "homelab" 09:27 09-Jul-20
```

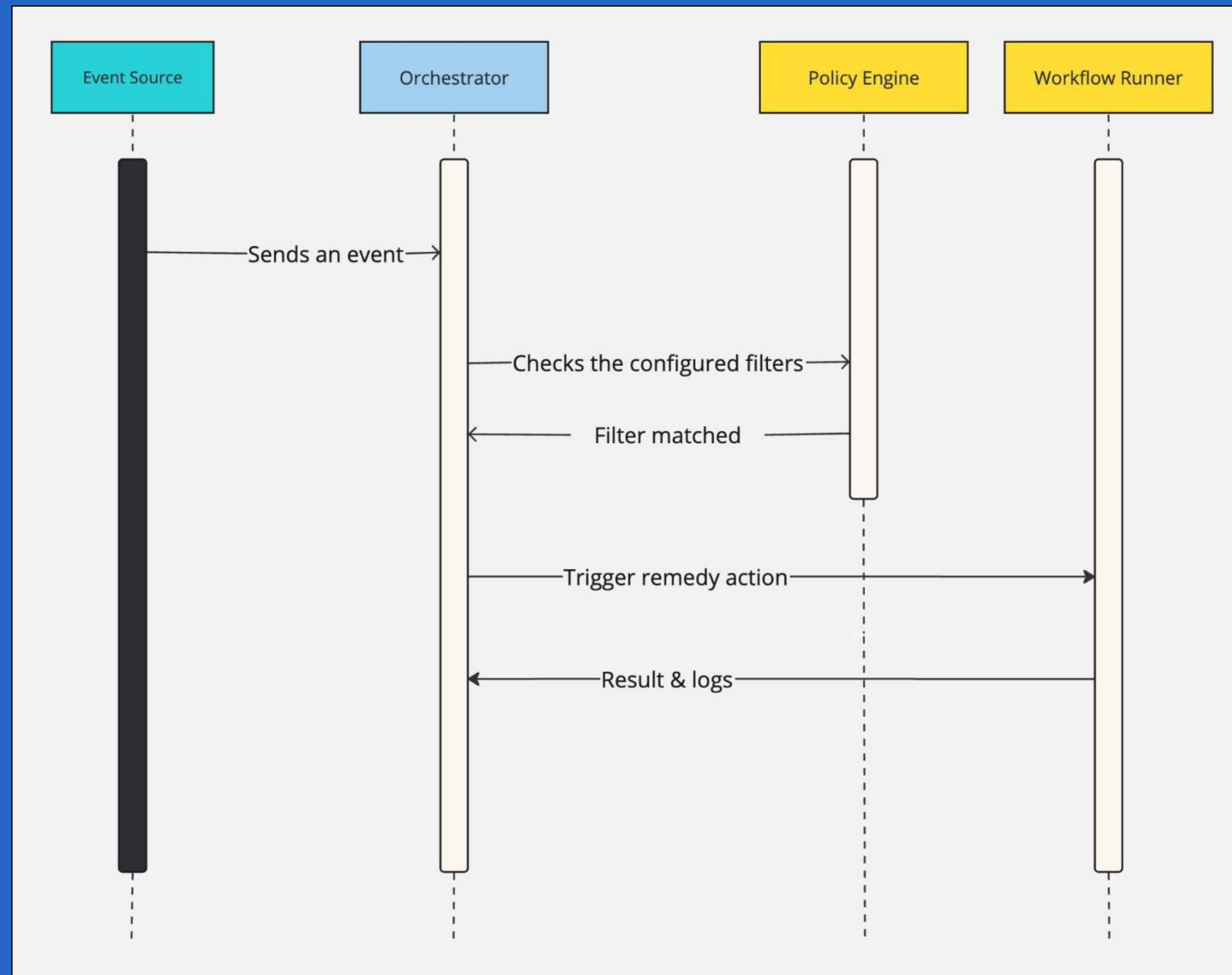


# Approx. number of events are generated from K8s

- Before delving into the architecture let's have a look at the number of events generated on an average from K8s. This will help us gauge the level we are operating at.
- So this prometheus graph shows us 1 day sum of total number of events generated from K8s across 1 week duration.
- You would see a dip in the graph on weekends, but even with the dip the min to max range of events are 6.4M to 6.83M, and this is only for one of our clients, for more than 10 clients imagine the number



# Architecture Diagram



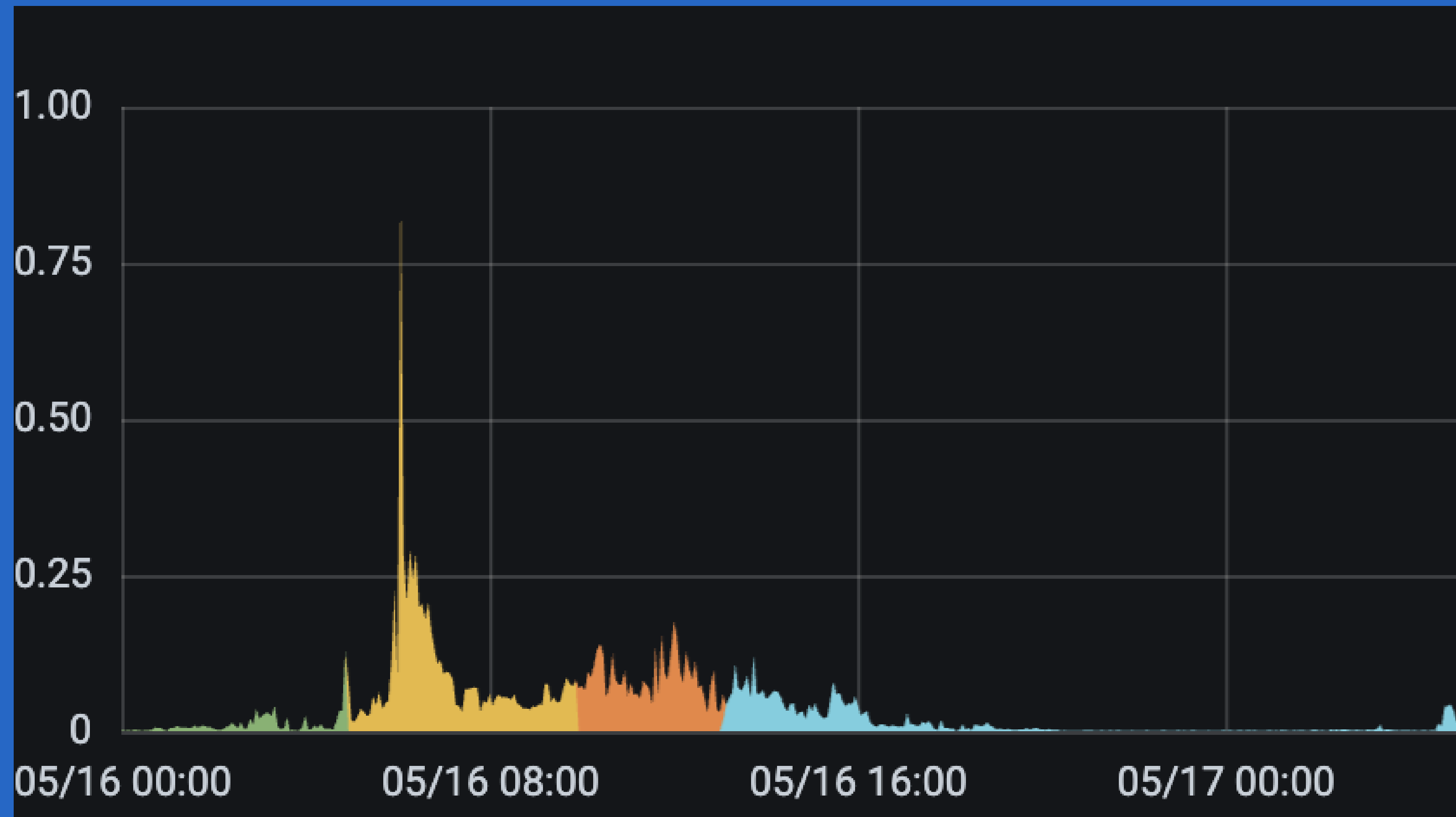
# Major Issue with the approach

- Let's understand the underlined issue using a prometheus graph, demonstrating total number of requests from devtron that hit apiserver.
- Again the dips are in the weekends, and the min to max range we are operating at is btw 5.05M to 5.4M, which is also a huge number considering the product we have.
- So, in auto remediation we show the diff btw. objects that changed, so that we can have initial object manifest and final object manifest to apply filter on. So to fetch the manifests we hit apiServer constantly which resulted in CPU spikes which in turn reflected in the aws bills.
- If you take a look at the sudden spike, this was after the auto remediation feature was enabled at the client.
- Later you could see a sudden dip in the graph, that was when we implemented a proper solution.



# Major Issue with the approach

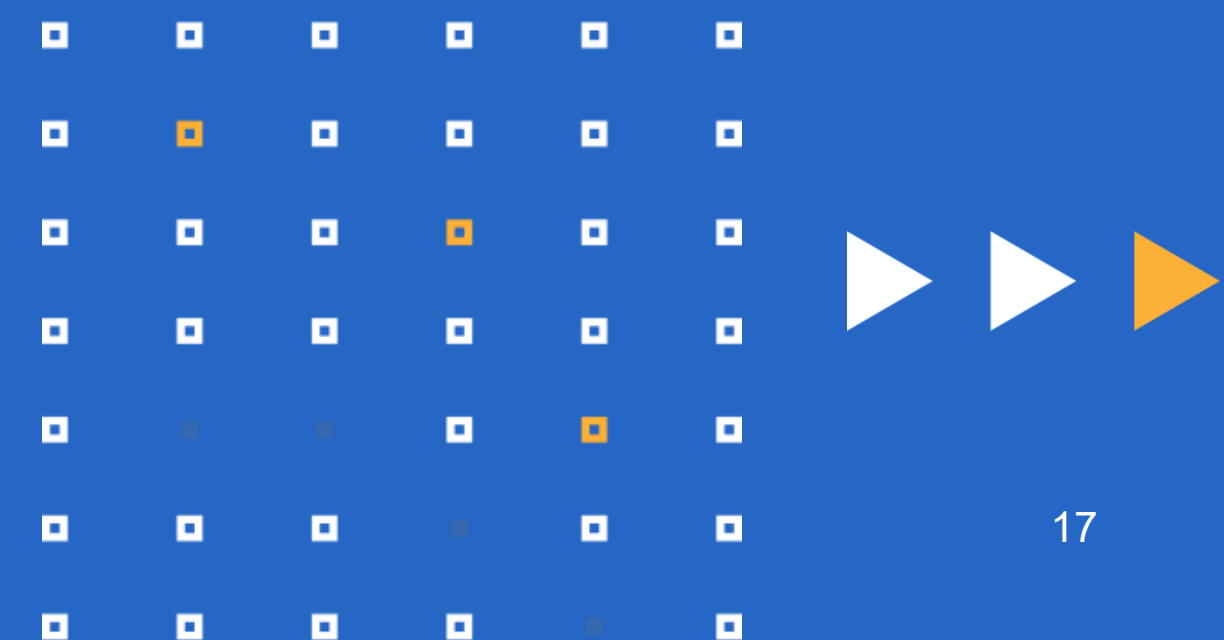
This is what the cpu usage looked like on the client. You can see the sudden spike.



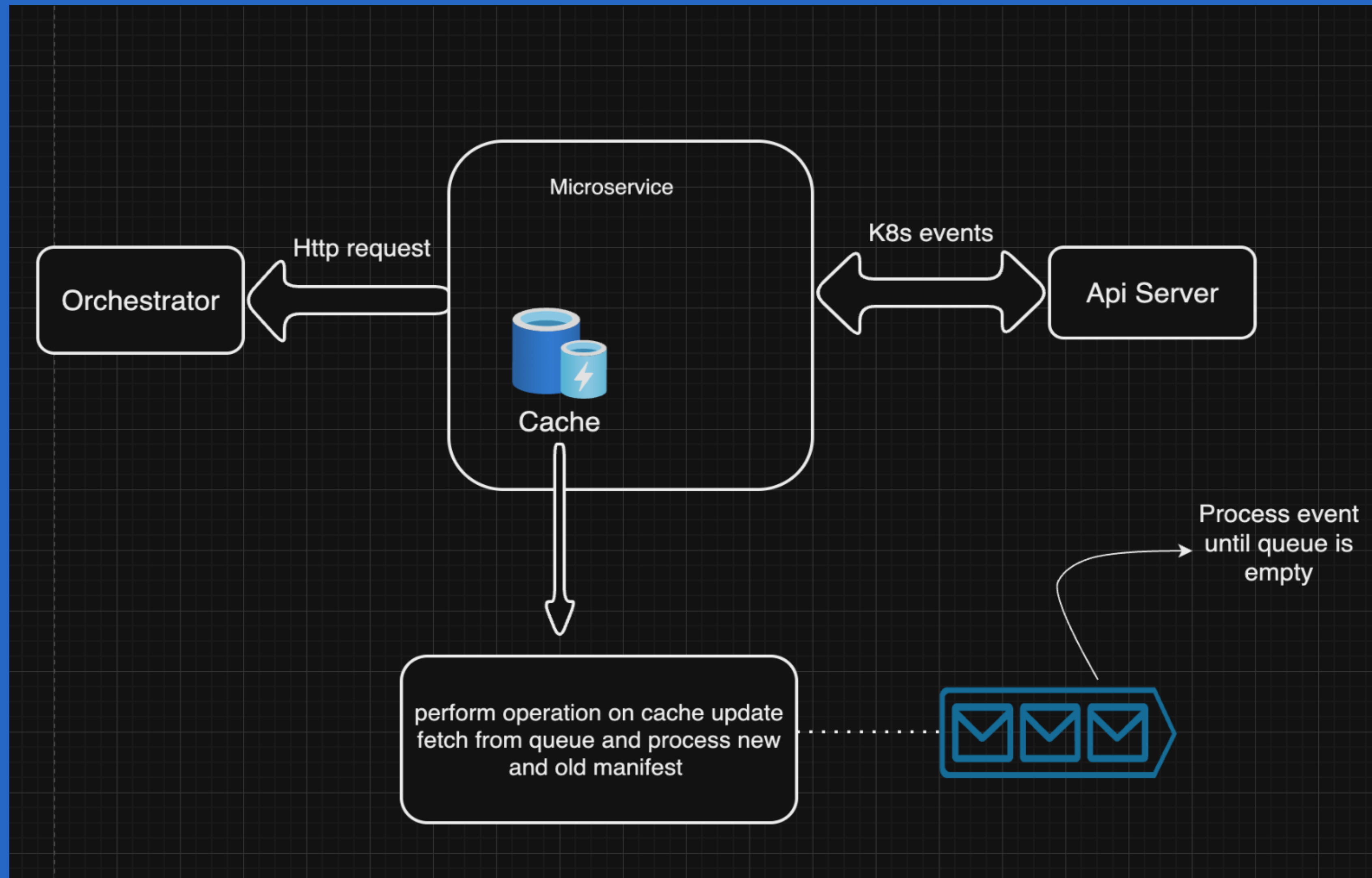


# Heated moments with clients and how we resolved those bottlenecks.

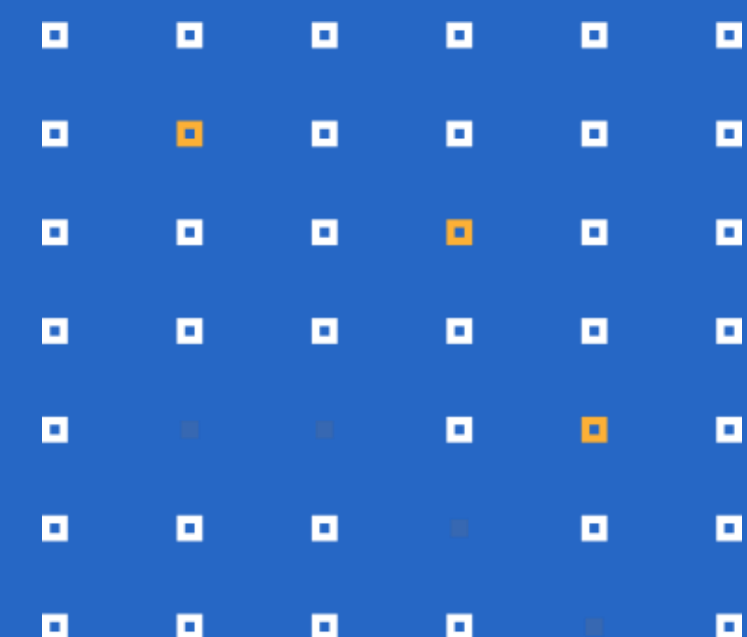
- So this was a major issue that clients reported, which was obviously intolerable. So we refined the approach a little bit.
- Now we are leveraging caches.
- Here, we are storing manifests in cache, okay so we have prepared a mechanism that everytime a manifest is getting updated or created or deleted, then the cache is constantly listening to the events and we are updating the cache with new and old resource manifest.
- And while running everything in go routines(light weight threads) we process everything in queue and process the new vs old resource manifest.
- So we are leveraging the FCFS(First Come First Serve) principal of queues and processing those events first which are coming first from cache, so when you do kubectl get events the ordering in which the output comes should be the order in which we should process the request.



# Basic Architecture



An intriguing and whimsical anecdote unfolded during our storytelling journey, adding a surprising twist that not only increased our bills but also tickled our sense of humor





Website: <https://devtron.ai>

GitHub: <https://github.com/devtron-labs/devtron>

Twitter: <https://twitter.com/Devtron>

Discord Server: <https://discord.devtron.ai>

YOU  
FOR  
YOUR  
PATIENCE  
PATIENCE

