**🟢 First things first: Why do we even need a server/VM?**

* Your Java JAR (order.jar) cannot just “float in the air.”
* It needs a **computer** with:
  + CPU
  + RAM
  + Disk
  + Operating System (Linux/Windows)
* That computer is called a **server** (or VM if it’s in the cloud).

👉 So before Kubernetes existed, every time we wanted to run our JAR, we had to **ask Infra/Cloud team to give us a VM**.

**🟢 Case 1: Running Retail Project Without Kubernetes**

**Example: You want to run Order Service**

1. You ask **cloud/infra team**:

“Please give me 1 Linux VM with 4 CPUs, 8 GB RAM, 100 GB disk.”  
They give you something like:

* + VM name: order-service-01.mycompany.com
  + IP: 10.20.30.40

1. You **login into the VM**:
2. ssh user@10.20.30.40
3. You **copy your app** (order.jar) to that VM:
4. scp order.jar user@10.20.30.40:/opt/apps/
5. You **start the app manually**:
6. cd /opt/apps
7. java -jar order.jar
8. App is now running on **that one VM**.  
   Customers can access it at:
9. http://10.20.30.40:8080/orders

**What happens if traffic grows (like big sale)?**

* 1 VM gets overloaded.
* You go back to cloud team:

“Please give me 2 more VMs.”

* They create:
  + order-service-02.mycompany.com (10.20.30.41)
  + order-service-03.mycompany.com (10.20.30.42)
* You repeat steps: copy JAR → run java -jar.
* Infra team also updates **Load Balancer** so traffic goes to all 3 VMs.

👉 This is **manual, slow, and risky**. If one VM crashes, you must fix it yourself.

**🟢 Case 2: Running Retail Project With Kubernetes**

Now imagine your company already has a **Kubernetes cluster** (say 5 servers/VMs combined together).  
This cluster is already created by:

* Cloud provider (AWS EKS, Azure AKS, GCP GKE), or
* Infra team (bare metal/VMs).

**Steps with Kubernetes:**

1. Instead of giving Kubernetes a JAR, you give it a **Docker image**:
2. mycompany/order-service:1.0
3. You write a **YAML file** (declaration):
4. apiVersion: apps/v1
5. kind: Deployment
6. metadata:
7. name: order-service
8. spec:
9. replicas: 3
10. template:
11. spec:
12. containers:
13. - name: order-container
14. image: mycompany/order-service:1.0
15. ports:
16. - containerPort: 8080
17. You tell Kubernetes to apply it:
18. kubectl apply -f order-deployment.yaml
19. Kubernetes automatically:
    * Chooses which of the 5 servers to run the 3 replicas on.
    * Starts your containers (no need for you to run java -jar).
    * Restarts them if they fail.
    * Balances traffic across replicas.
    * If traffic grows, you just say:
    * kubectl scale deployment order-service --replicas=5

👉 Scaling, restarting, and balancing are **automatic**. You don’t care which VM it’s running on—Kubernetes decides.

**🟢 Why do we need Kubernetes?**

Without Kubernetes:

* You must ask for servers one by one.
* Copy JAR and run it manually.
* Update load balancer manually.
* Fix crashes yourself.

With Kubernetes:

* You don’t ask for new servers every time.
* You don’t run java -jar manually.
* You just say how many copies you want → Kubernetes does the rest.
* Auto restart, auto scaling, auto balancing.

✅ **In one line:**  
Without Kubernetes = you babysit each VM.  
With Kubernetes = you just declare what you want, Kubernetes babysits everythi

**🔹 First, What is a Kubernetes Cluster?**

Think of a **Kubernetes cluster** like a **big shopping mall**:

* The **mall building** = the **cluster**
* The **shops inside the mall** = your **services (Forecast, Ordering, Grid, Catalogue)**
* The **mall manager** = the **control plane** (decides where shops go, handles security, scaling, etc.)
* The **floors/rooms inside the mall** = the **worker nodes** (the physical servers or VMs where services actually run)

👉 So, a cluster is just a **group of servers managed together as one unit**.  
Instead of handling each server (VM) separately, you handle the whole mall (cluster).

**🔹 Why do we need this cluster?**

Let’s compare:

**🟢 Without Cluster (old way)**

* Forecast runs on VM1
* Ordering runs on VM2
* Grid runs on VM3
* Catalogue runs on VM4

⚠️ Problems:

* If VM2 (Ordering) crashes → Ordering is down.
* If Ordering suddenly needs 5 copies → you need 4 more VMs → wait for infra team.
* Each VM may waste resources (VM1 may be half-empty but cannot share with Ordering).

**🟢 With Cluster (Kubernetes way)**

* You have **one big cluster** (say 5 worker nodes).
* All services (Forecast, Ordering, Grid, Catalogue) run **inside this cluster as containers (pods)**.
* Kubernetes (mall manager) decides:
  + Put 2 copies of Ordering on Node1 + Node2
  + Put Forecast on Node3
  + Put Grid on Node4
  + Put Catalogue on Node5

⚡ Benefits:

1. **High Availability** → If Node1 crashes, Kubernetes moves Ordering pods to another node.
2. **Auto Scaling** → If Ordering needs more capacity:
3. kubectl scale deployment order-service --replicas=5

Kubernetes will spread 5 copies across free nodes.

1. **Better Resource Sharing** → If Forecast uses only 20% of Node3, Ordering can also run on Node3.
2. **No Manual Work** → No need to request new VMs, copy JARs, restart manually → Kubernetes handles all.

**🔹 Example in Your Retail Project**

Let’s say:

* You have 4 services: Forecast, Ordering, Grid, Catalogue
* You have a cluster with 3 worker nodes (servers):

Cluster:

- Node1 (8 CPU, 16 GB RAM)

- Node2 (8 CPU, 16 GB RAM)

- Node3 (8 CPU, 16 GB RAM)

Now you deploy:

* Forecast → 2 pods
* Ordering → 3 pods
* Grid → 1 pod
* Catalogue → 2 pods

Kubernetes places them like this:

Node1: Forecast pod, Order pod

Node2: Order pod, Catalogue pod

Node3: Grid pod, Forecast pod, Order pod, Catalogue pod

👉 If Node2 crashes:

* Its Order + Catalogue pods are automatically restarted on Node1 or Node3.

👉 If Ordering needs more copies:

* Kubernetes can spread new pods on whichever node has free CPU/memory.

**🔹 Super Simple Meaning**

👉 A **cluster** is just a **pool of servers acting like one big computer**.  
👉 Kubernetes runs your **retail services (Forecast, Ordering, Grid, Catalogue)** on that pool.  
👉 It automatically **balances, restarts, and scales** them, so you don’t care which server is doing what.

**Step 1: You need a name for your app (Domain)**

* Think of https://retailapp.com as the **name of your shop**.
* This name has to be registered somewhere so that the internet knows it belongs to you.
* You do this by **buying a domain** from services like:
  + GoDaddy, Namecheap, Google Domains

✅ Now you have a **name** for your app.

**Step 2: The signboard that points to your building (DNS)**

* You have a building where your app lives. This building has a **real address** (an IP address, like 52.12.45.90).
* The internet doesn’t know your building’s address by name. It only knows **numbers (IP)**.
* DNS (Domain Name System) is like a **phonebook**:
  + You tell it: retailapp.com → 52.12.45.90
* Now, anyone typing https://retailapp.com will **go to your building** automatically.

**Step 3: You have multiple chefs in the building (Backend Servers)**

* Your retail project has multiple modules:
  + Forecast → Server1, Server2
  + Ordering → Server1, Server2, Server3
  + Catalogue → Server1
* These are like **different chefs in different kitchens**.
* The user doesn’t care which chef cooks the food. They just order through the counter.

**Step 4: The reception desk (Load Balancer)**

* The **Load Balancer** is like the **front desk of your building**:
  1. User opens the browser and clicks Forecast.
  2. Browser sends request to https://retailapp.com.
  3. DNS translates retailapp.com → IP of Load Balancer.
  4. Load Balancer receives request → decides which Forecast server (Server1 or Server2) will handle it.
  5. Backend server processes request → response goes back **through Load Balancer** → user sees data.

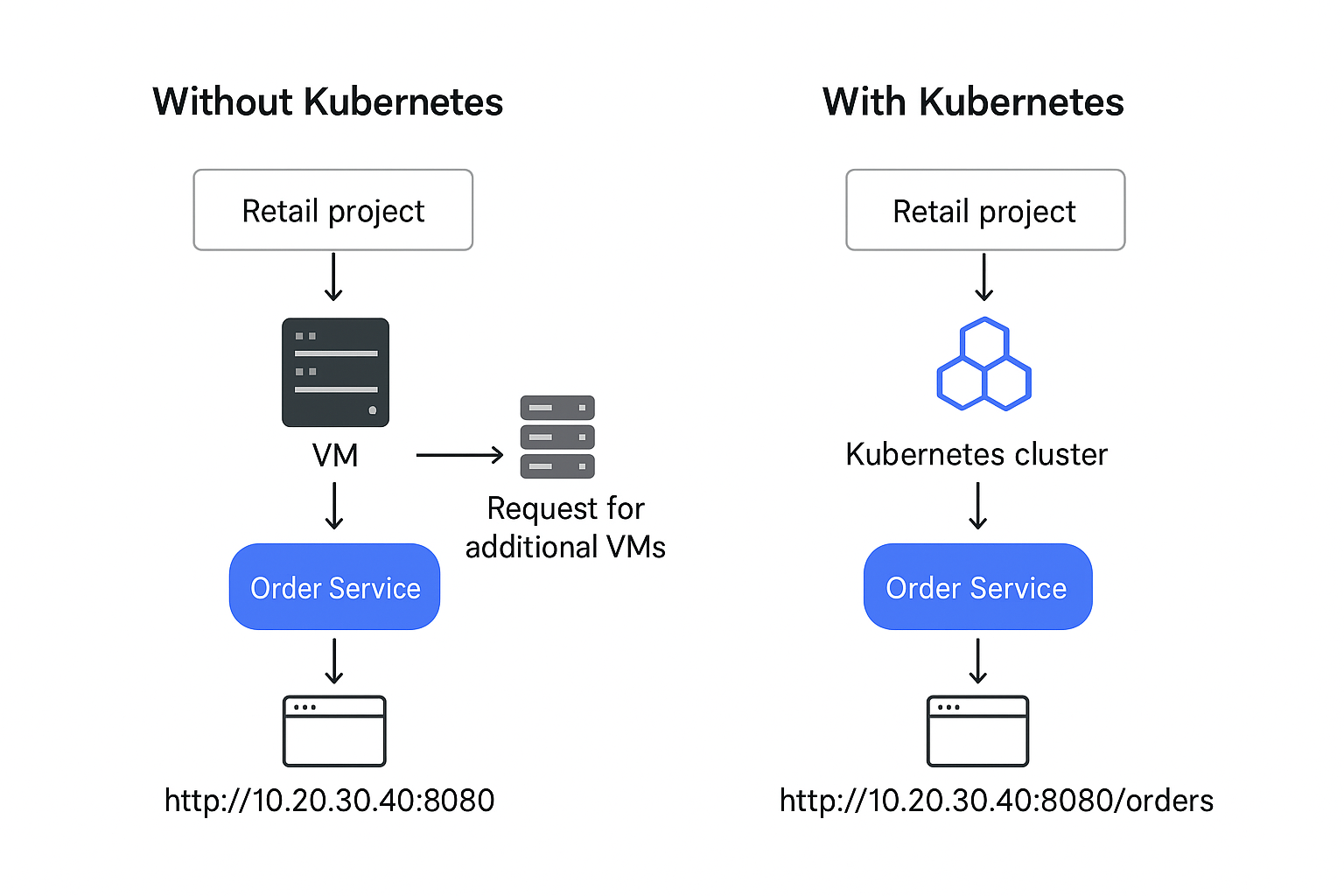
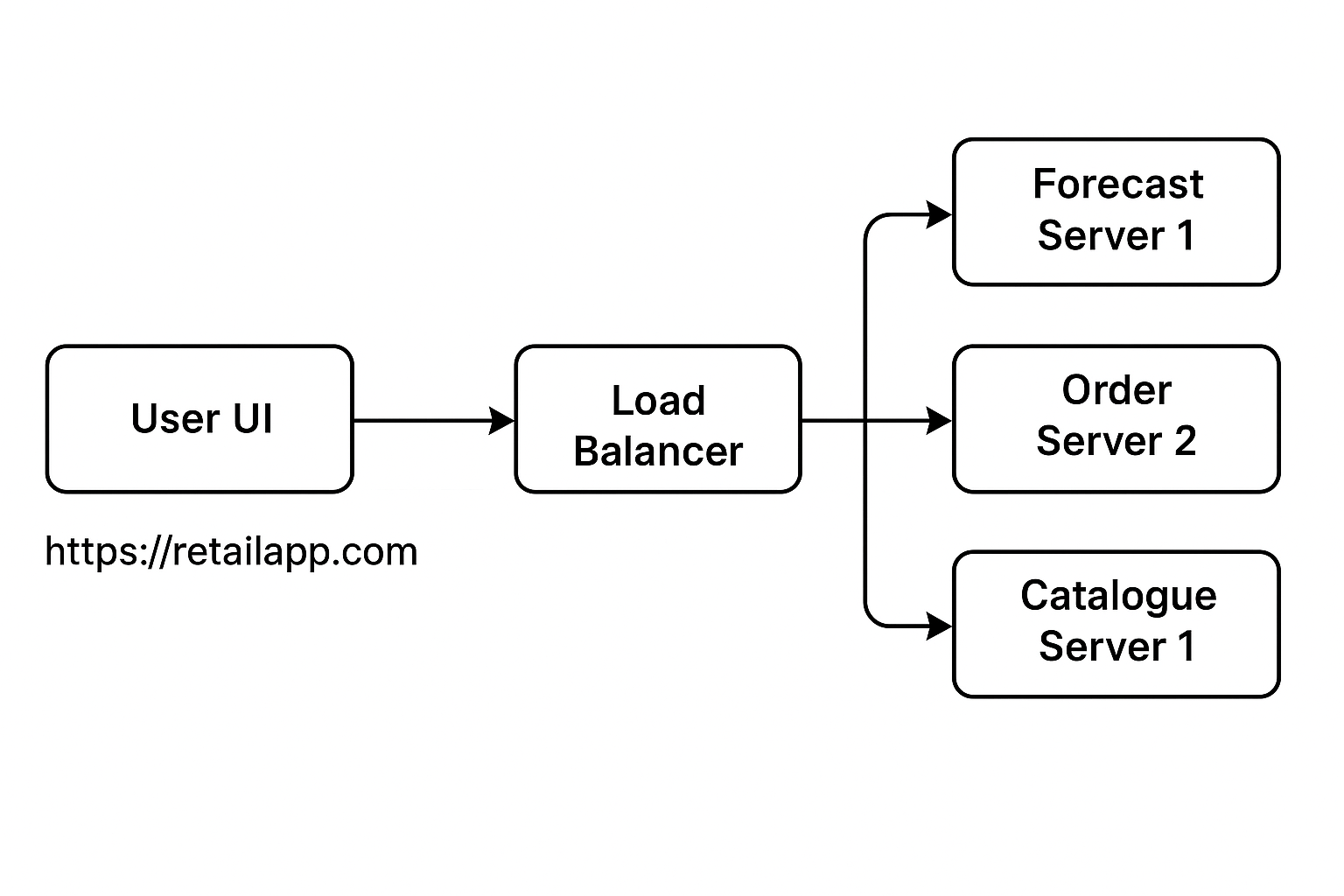
✅ User always sees **one URL**, even though multiple servers are working behind the scenes.

**Step 5: Step-by-step “path” of a request**

1. User types: https://retailapp.com → opens browser
2. Browser asks DNS: “What’s the IP of retailapp.com?”
3. DNS replies: 52.12.45.90 (Load Balancer)
4. Browser sends request to 52.12.45.90
5. Load Balancer receives request:
   * Forecast module? → forwards to **Forecast Server1**
   * Orders module? → forwards to **Order Server2**
6. Backend server responds → goes back through Load Balancer → browser displays data

**Step 6: Super-simple analogy**

| **Thing** | **Analogy** |
| --- | --- |
| URL retailapp.com | Name of your shop |
| DNS | Phonebook translating shop name to building number |
| Load Balancer | Reception desk deciding which chef/kitchen will handle order |
| Backend Servers | Kitchens/chefs that prepare the data |
| UI | Customer seeing the final product |



**scenario**

You have a retail project with services like:

* ordering
* forecast
* grid
* catalogue

Each service is running as a **Kubernetes pod** inside a cluster.

Sometimes, you see pods being **restarted daily during non-business hours**. You want to know **why and how**.

**1️⃣ Why Kubernetes Pods are restarted**

There are several reasons:

**a) Manual restart for updates**

* If you deploy a new version of a service (say, ordering got a bug fix), you might need to **restart pods** so the new image/version is picked up.
* Example: kubectl rollout restart deployment ordering

**b) Health and stability**

* Pods can be automatically restarted if **liveness probes** or **readiness probes** fail.
* Example:
  + Liveness probe checks if your service is alive.
  + If it fails, Kubernetes kills the pod and starts a new one.

**c) Scheduled maintenance / auto-restart**

* Companies often **restart pods daily in non-business hours** for:
  + Clearing memory leaks
  + Refreshing connections (DB, cache, etc.)
  + Applying updated config maps or secrets
  + Ensuring system stability for critical retail operations
* This is usually done with a **cron job** or **CI/CD pipeline**.

**d) Resource optimization**

* If pods consume too much memory or CPU over time, restarting them helps maintain **performance**.

**2️⃣ How pods are restarted**

In Kubernetes, pods are **ephemeral**. You generally don’t restart pods manually like servers. There are a few ways:

**Option 1: kubectl rollout restart**

kubectl rollout restart deployment ordering

kubectl rollout restart deployment forecast

* Restarts all pods in the deployment one by one.
* Ensures zero downtime (thanks to replicas).

**Option 2: Delete pods manually**

kubectl delete pod <pod-name>

* Kubernetes automatically creates a new pod to maintain the desired state.
* Usually not done manually for multiple pods.

**Option 3: CI/CD / CronJob automation**

* A scheduled **CronJob** in Kubernetes or in your CI/CD pipeline can run:

kubectl rollout restart deployment --all

* Typically scheduled during **non-business hours** (like 2 AM) so it doesn’t impact users.

**3️⃣ Why schedule it in non-business hours**

* Retail services handle **orders, forecasting, and inventory**.
* Restarting during business hours could cause:
  + Users unable to place orders
  + Forecast calculations failing
  + Grid/dashboard showing errors
* So, daily automated restarts are scheduled **when traffic is low**.

**4️⃣ Real-world example (retail project)**

Suppose you have **ordering service**:

* Pods: ordering-1, ordering-2, ordering-3
* Deployment: ordering-deployment
* Daily cron job runs at 2 AM:

# pseudo cronjob.yaml

schedule: "0 2 \* \* \*"

job:

steps:

- kubectl rollout restart deployment ordering-deployment

- kubectl rollout restart deployment forecast-deployment

* Outcome:
  + Each pod is restarted one by one
  + New pods pick up latest configs and are healthy
  + No impact on daytime operations

✅ **Summary:**

* **Why restart pods:** updates, memory leaks, config refresh, health
* **How pods are restarted:** manually (kubectl), automatically (CronJob or CI/CD)
* **Why non-business hours:** avoid downtime for critical retail operations

If you want, I can **draw a simple visual diagram showing your retail services and how daily pod restarts happen**. It makes it super easy to understand.

Do you want me to draw that?

**You said:**

some times i used to get pager duty alert for pod over running so we will be asking users to close it or else do action like kill it or something so what are all those

**ChatGPT said:**

Ah! Now it’s clear—you’re talking about **alerts from PagerDuty (or any monitoring tool) when pods are “over running” or misbehaving**, and the actions you take. Let’s break it down **clearly and step by step** for your retail project.

**1️⃣ What “pod over running” or alerts mean**

In Kubernetes:

* Each pod has **resources limits** (CPU, memory) set in the deployment YAML.  
  Example:
* resources:
* requests:
* memory: "512Mi"
* cpu: "500m"
* limits:
* memory: "1Gi"
* cpu: "1"
* **Over running alerts** are triggered when a pod **exceeds its CPU/memory limits** or behaves abnormally.

**Examples of issues that trigger alerts:**

| **Alert Type** | **Meaning** |
| --- | --- |
| CPU usage high | Pod is consuming more CPU than expected; may slow down the service |
| Memory usage high | Pod is consuming too much memory → risk of OOM (Out Of Memory) crash |
| Pod not ready / crash looping | Pod failed liveness or readiness checks multiple times |
| Pod stuck or hung | Service inside the pod is unresponsive |

**2️⃣ What PagerDuty does**

* PagerDuty monitors alerts from tools like **Prometheus, Datadog, Dynatrace**.
* When a pod crosses thresholds (CPU/memory, crash loops), it **sends an alert** to the team.
* Example:
  + “ordering-1 pod CPU at 95% for 10 min”
  + “forecast pod memory at 1.2Gi / 1Gi limit”

**3️⃣ Actions you take when alert comes**

**a) Ask users to close heavy operations**

* Sometimes alerts are because users are performing heavy actions (bulk upload, forecast recalculation).
* You may wait or ask users to **pause/stop those operations** temporarily.

**b) Kill or restart the pod**

* If pod is stuck or overconsuming, you can **force Kubernetes to restart it**:

kubectl delete pod <pod-name>

* Kubernetes automatically creates a **new healthy pod**.

**c) Scale up replicas**

* If load is too high, you might **increase pod replicas**:

kubectl scale deployment ordering-deployment --replicas=5

* This distributes load and reduces CPU/memory pressure.

**d) Check logs**

* Always check **pod logs** to see what caused high usage:

kubectl logs <pod-name>

* If it’s a real bug (memory leak, infinite loop), you need **code fix**.

**e) Investigate root cause**

* Use monitoring dashboards (Prometheus, Grafana, Dynatrace) to see trends.
* Sometimes you need **config changes**: increase memory limit, tune queries, or fix code.

**4️⃣ Flow in your retail project**

1. **Alert triggers** → PagerDuty sends notification
2. **Initial check** → Is it a user-heavy operation? Wait or ask them to stop
3. **Immediate action** → Kill/restart pod if unresponsive
4. **Scaling** → Increase replicas if load is expected to remain high
5. **Root cause analysis** → Fix bug, optimize code, or increase resource limits

✅ **Summary:**

* “Pod over running” = pod CPU/memory exceeded or pod stuck
* PagerDuty alerts you
* You can ask users to stop, kill/restart pod, scale, or fix code
* This is all to **ensure retail services (ordering, forecast, grid) keep running smoothly**