
DevOps
& Cloud Native
Concepts

DevOps & Cloud Native Concepts for High Availability

Overview

Canary Release	Zero-Downtime Deployment	Zero-Downtime DB Migration
Resilience	Monitoring & Alerting	Product Metrics

Summary

DevOps & Cloud Native Concepts for High Availability

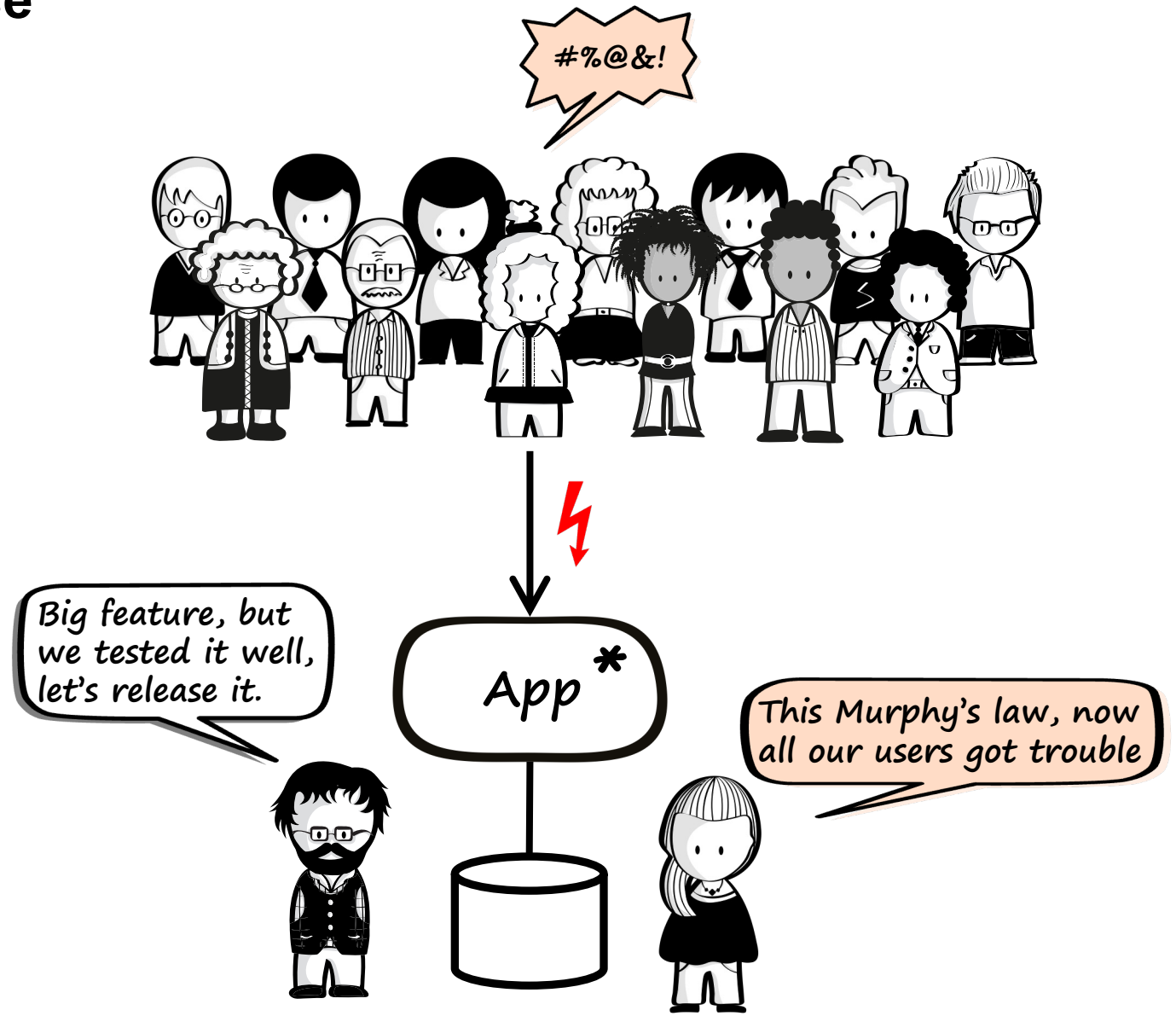
Overview

Canary Release	**Zero-Downtime Deployment**	**Zero-Downtime DB Migration**
Resilience	**Monitoring & Alerting**	**Product Metrics**

Summary

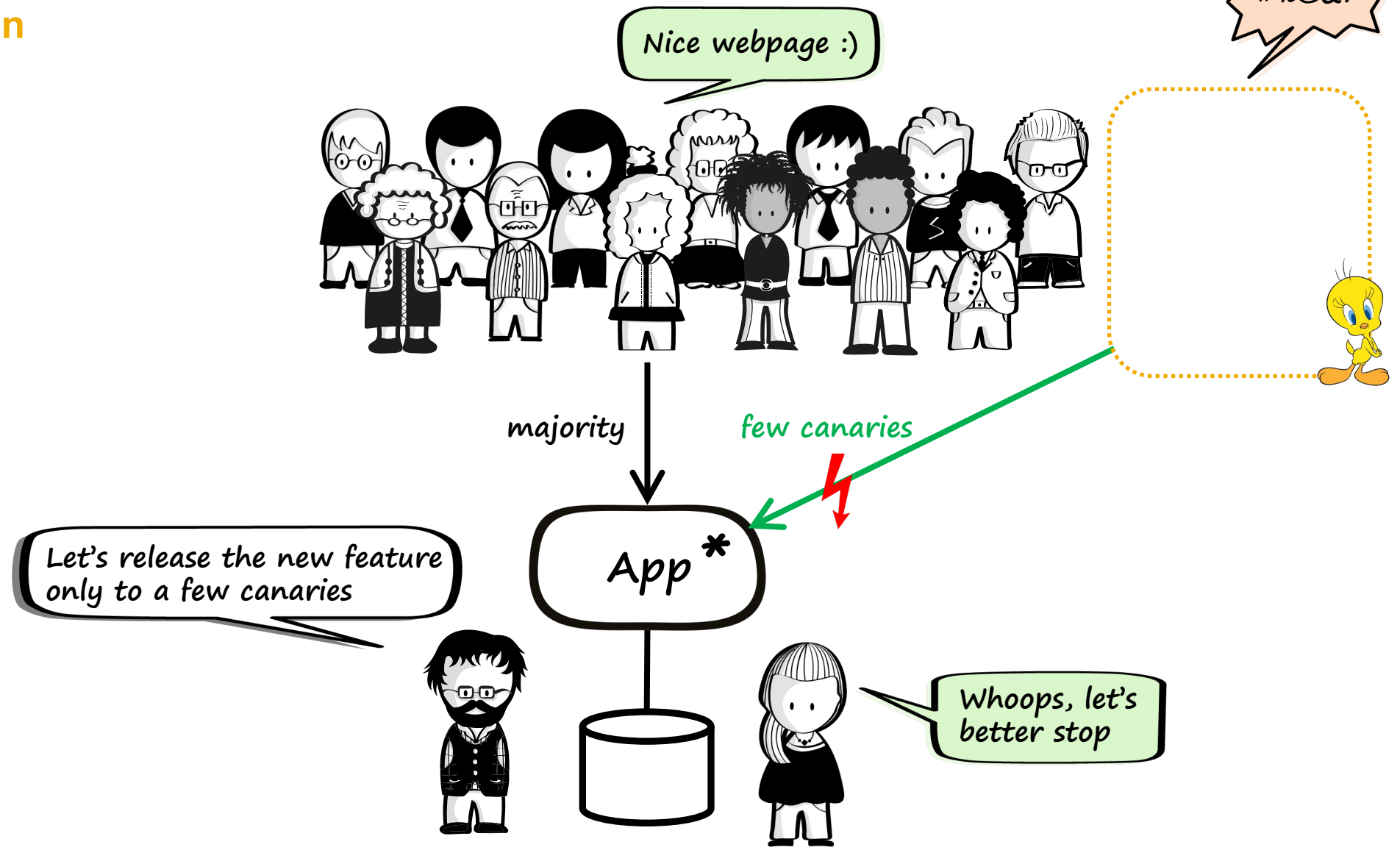
Canary Release

Problem



Canary Release

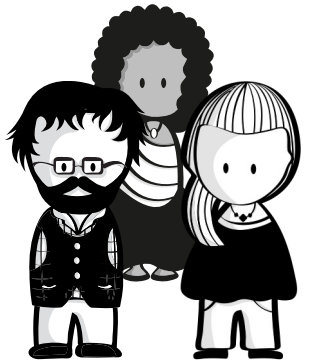
Solution



Canary Release
Reflection

- What different approaches of a canary release can you imagine?
- What knowledge/skills would the different approaches require?
- Which other strategies help to mitigate the risk of severe errors? Is canary your first choice?
- How would you even get to know that something is going wrong?
- Is canary release only preventing a disaster from a functional perspective?
- Are there cheap ways to prevent a disaster earlier?

Canary Release
Conclusion



Need diverse skills:

- Networking, routing, load balancing, feature toggles, side-by-side deployment
- Identifying risks in release from engineering, user & business perspective
- Tools & technology: app router, nginx, proxy, toggle frameworks/services, DNS

Impact of a “bad microservice design”:

- Canary approach will not be local to a service, multiple teams need to align on the strategy
- Making a canary approach feature-specific will be much more expensive & time-consuming
- Risks are much harder to identify in a distributed, yet tightly coupled system

DevOps & Cloud Native Concepts for High Availability

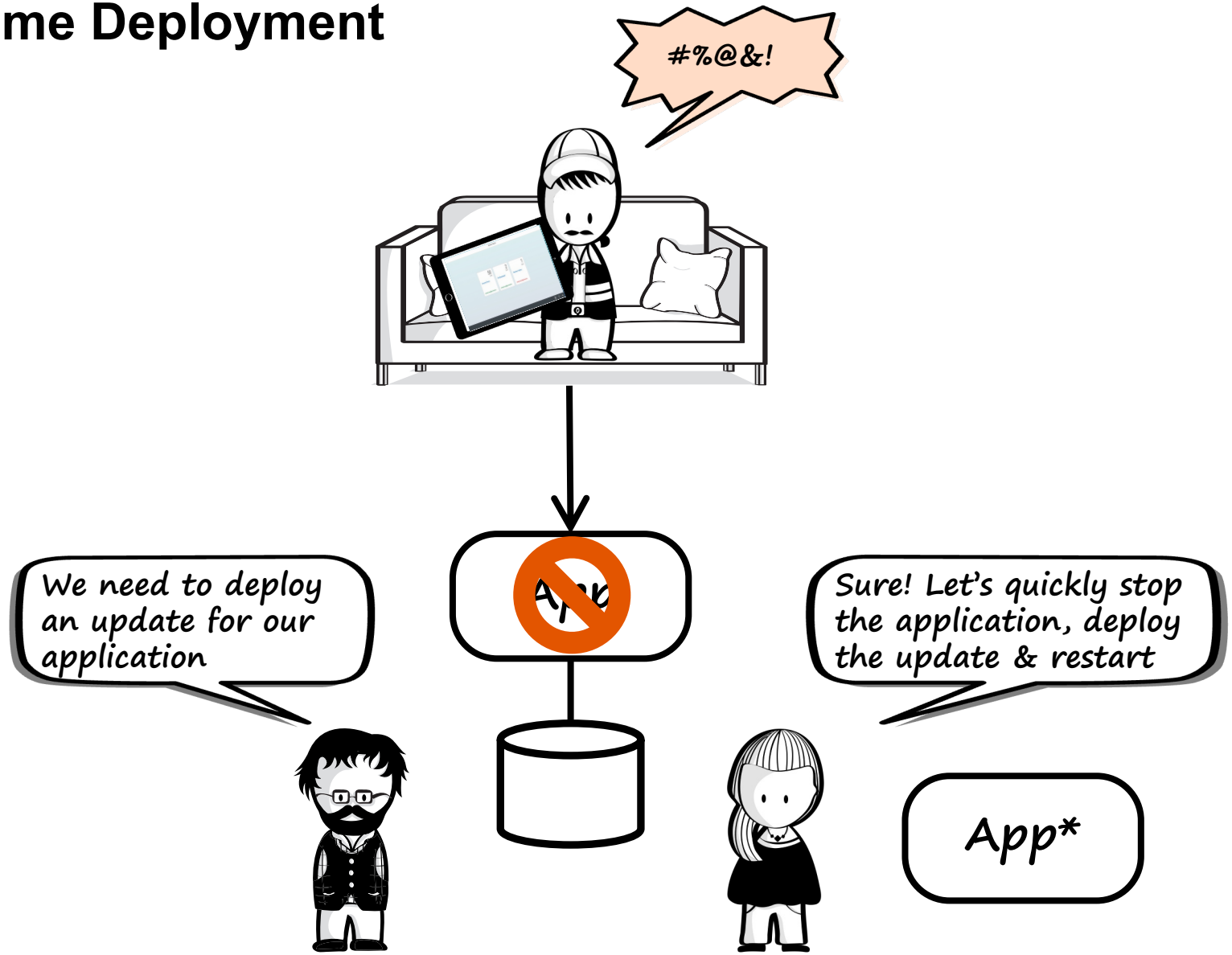
Overview

Canary Release	Zero-Downtime Deployment	Zero-Downtime DB Migration
Resilience	Monitoring & Alerting	Product Metrics

Summary

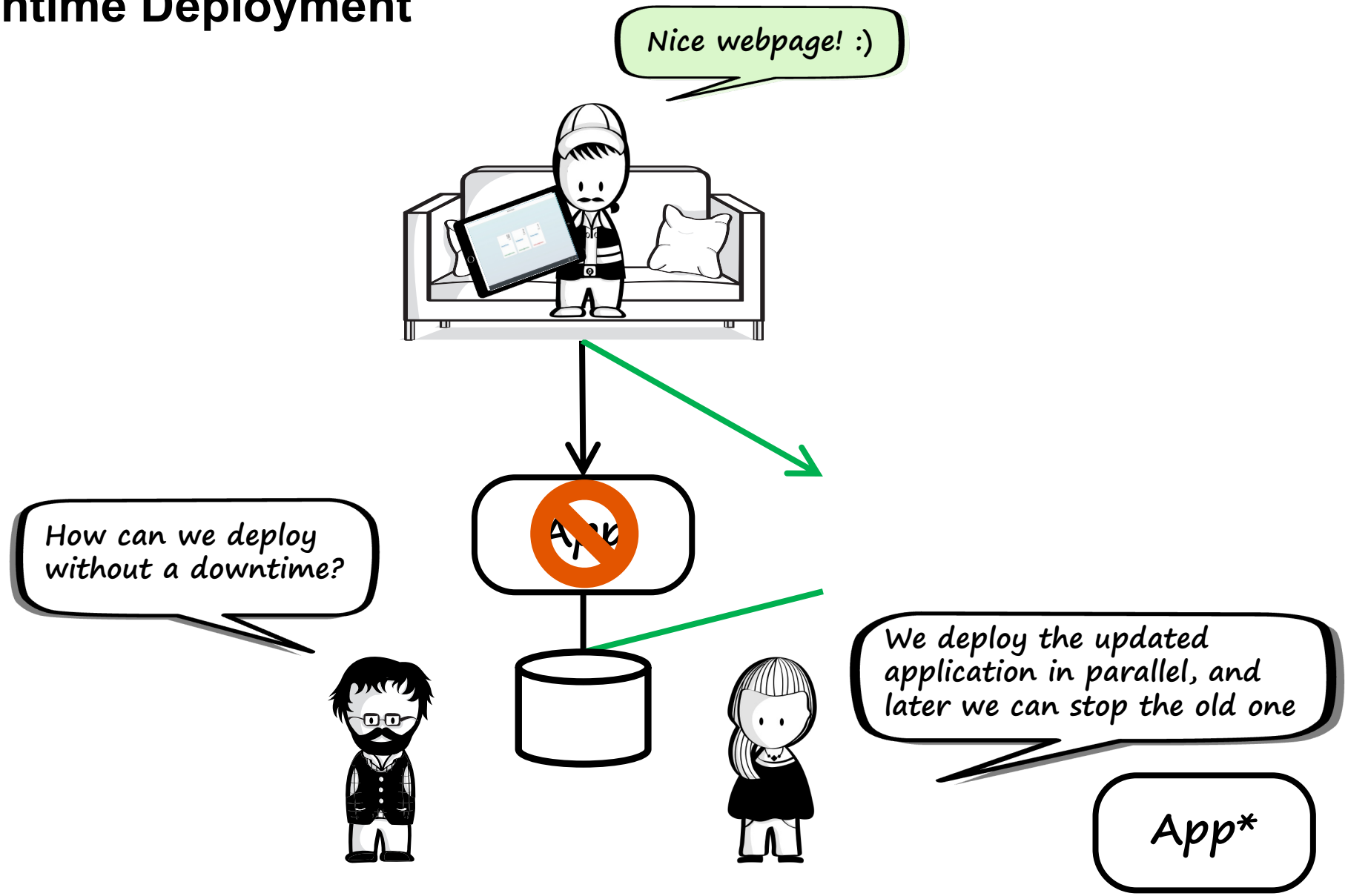
Zero-Downtime Deployment

Problem



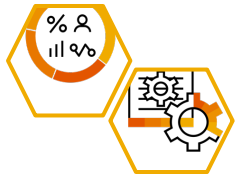
Zero-Downtime Deployment

Solution



Zero-Downtime Deployment

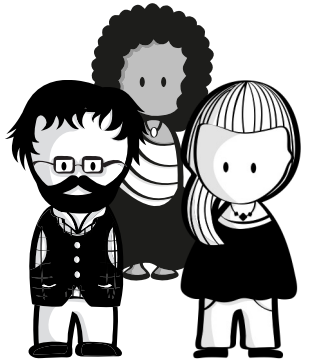
Reflection



- What if we have some application state, does it have any impact?
- How to do this if you have multiple instances of your application?
- Can we make any changes to API endpoints or DB schema without any problems?
- What happens with open connections when the old instance gets shut down?
- How do we know that we can safely proceed?

Zero-Downtime Deployment

Conclusion

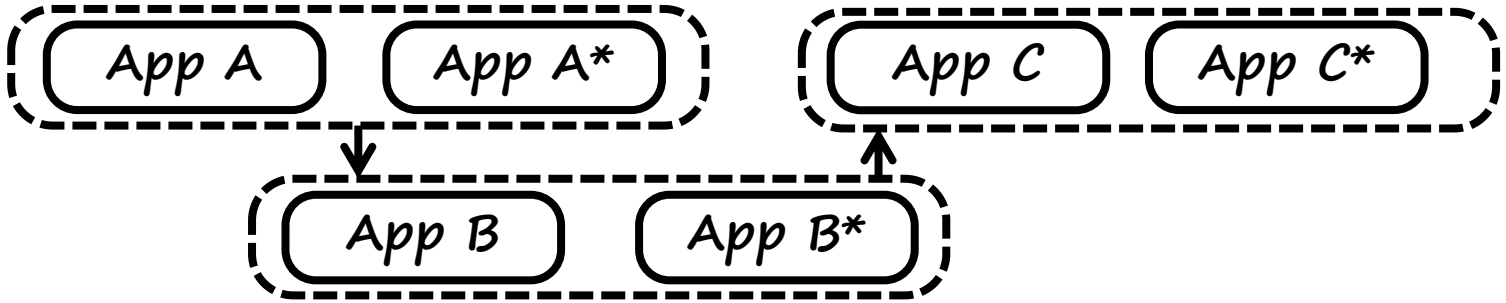


Need diverse skills:

- Make sure application is stateless or deals with state properly and can shutdown gracefully
- Configure load balancing / routing, starting / stopping of application, monitoring / smoke testing
- Tools & Technology: CF routes, K8s ingress/load balancing/rolling update, DNS, Redis, …

Impact of a “bad microservice design”:

- New features “smeared” across multiple services; need to ensure correct orchestration



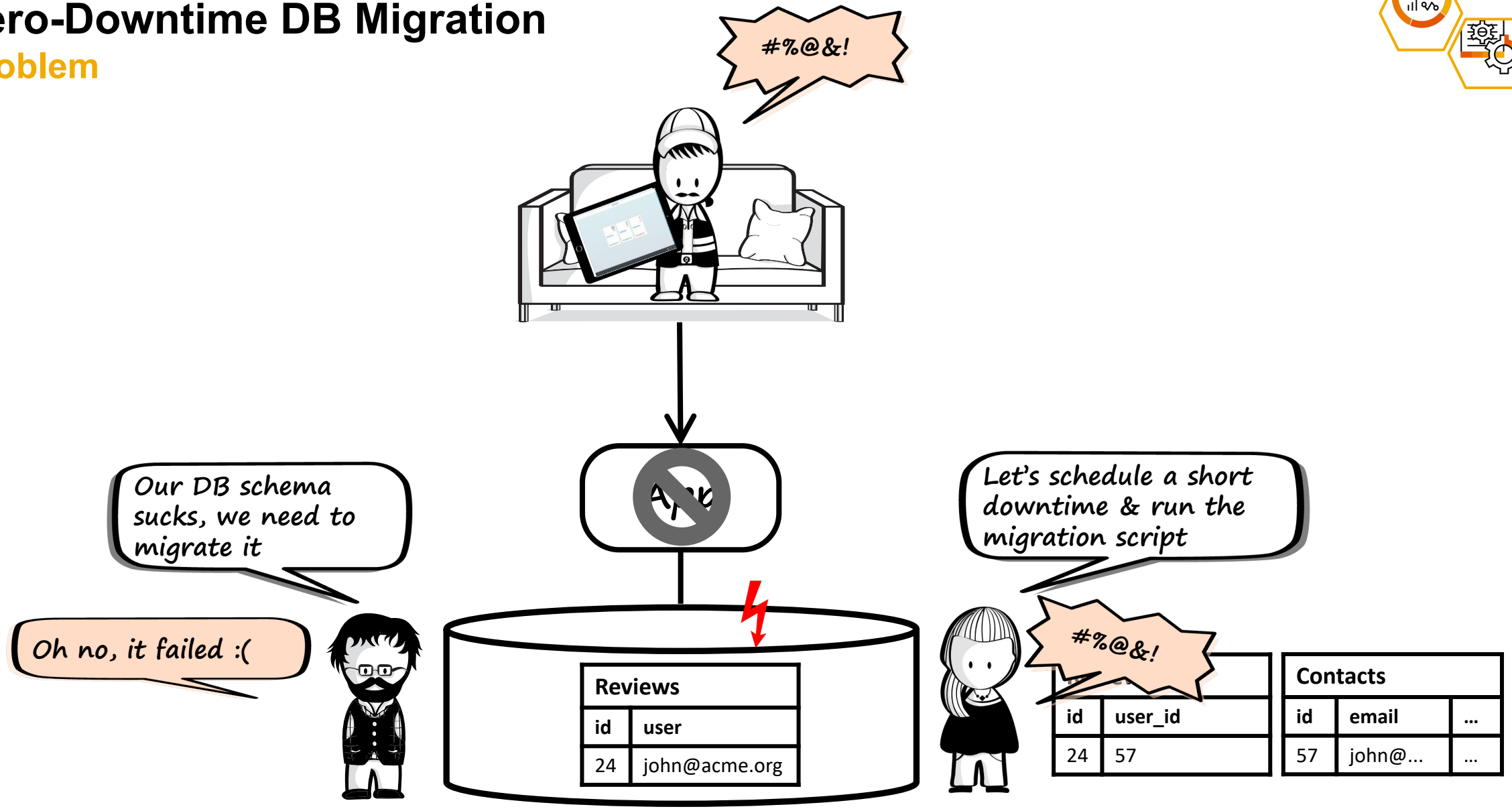
DevOps & Cloud Native Concepts for High Availability

Overview

Summary

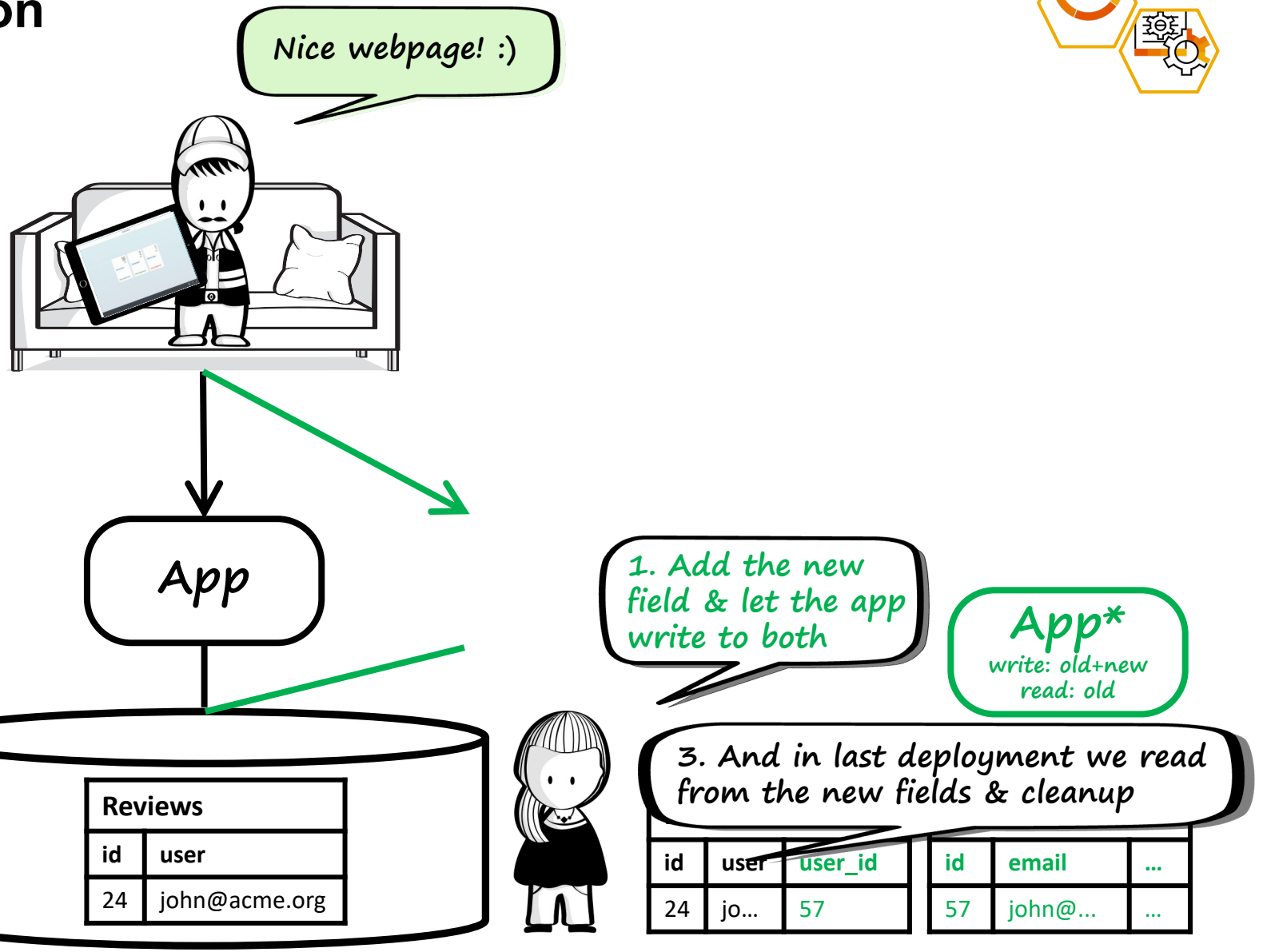
Zero-Downtime DB Migration

Problem



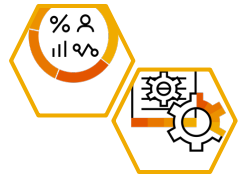
Zero-Downtime DB Migration

Solution



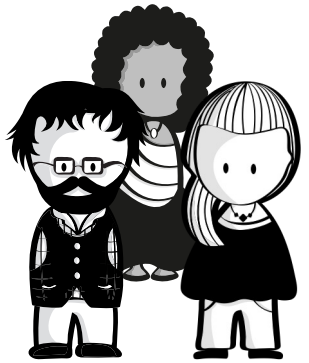
Zero-Downtime DB Migration

Reflection



- What can make it harder to do a migration in the zero-downtime way?
- How to test that your migration steps are fine, especially during the transitions?
- When would you do this – what are pros and cons from team and from business perspective?
- How slow or how fast should you go through a migration?
- DBs in different regions might be in different versions, how to assure correct schema setup?

Zero-Downtime DB Migration
Conclusion



Required skills – DevOps:

- Adjust queries / code on persistence layer
- Change DB data structures, deployment
- Tools & technology: Flyway, Liquibase, SQL, JPA, DBA tools, …

Impact of a “bad microservice design”:

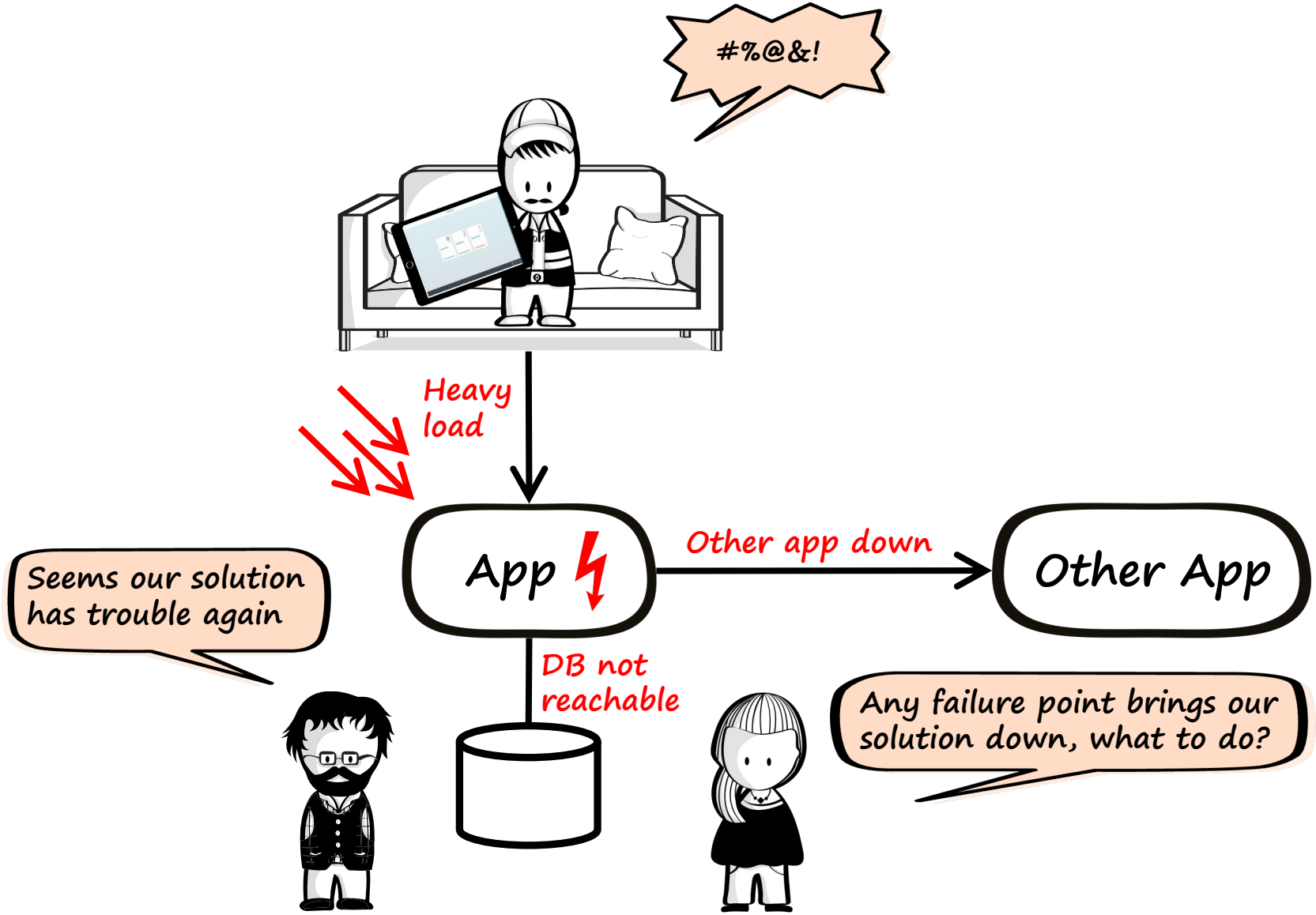
- Don’t know or own all code that accesses the structures to be changed
- Different goals and priorities of the different teams owning the structures
- DB schema will likely not be owned / controlled by the service team

DevOps & Cloud Native Concepts for High Availability

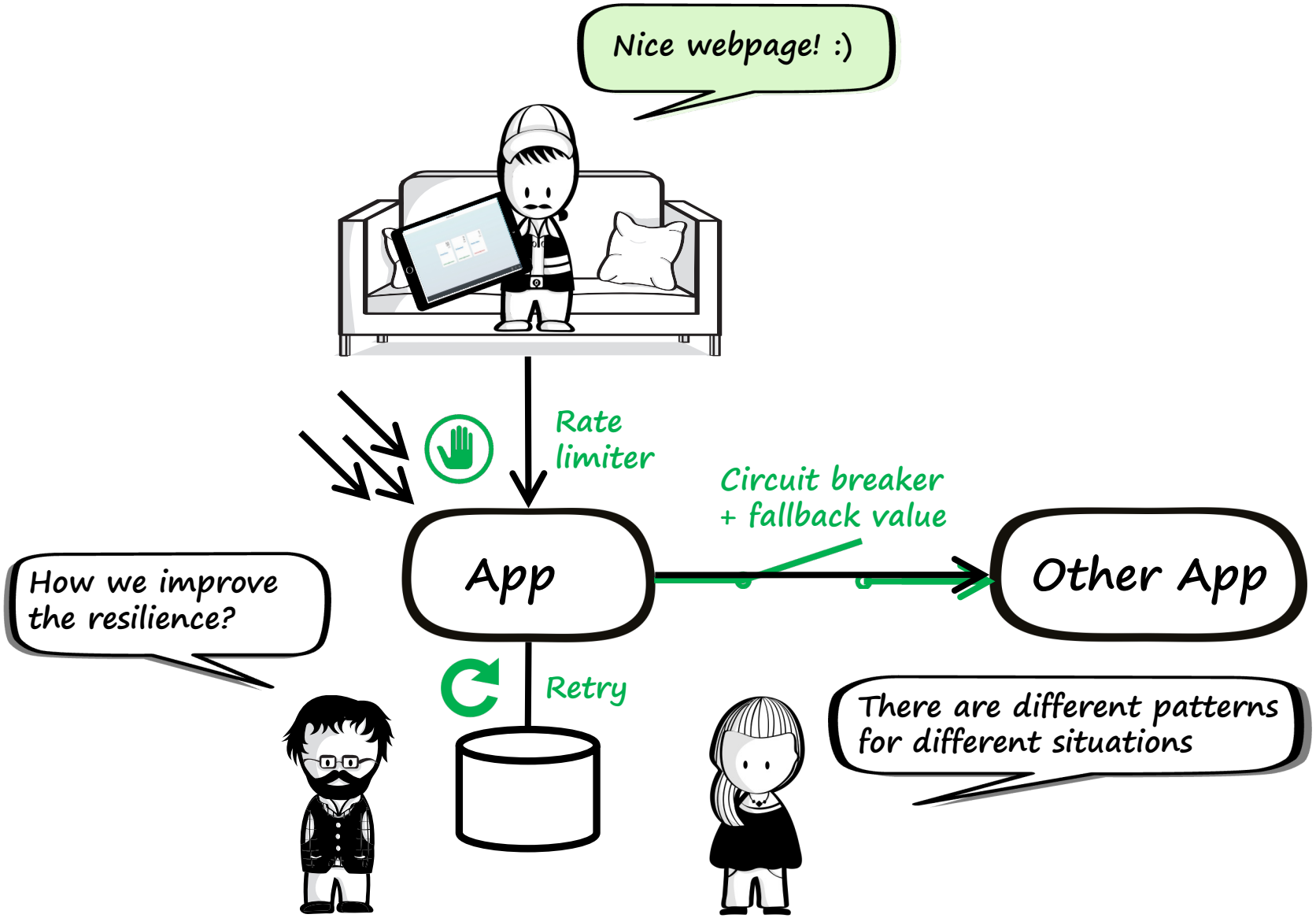
Overview

Summary

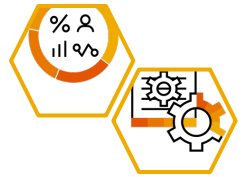
Resilience
Problem



Resilience
Solution

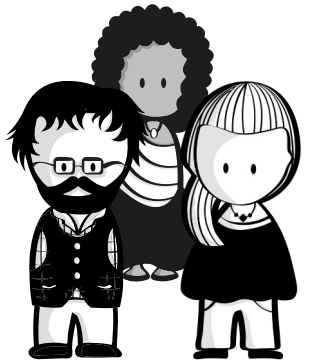


Resilience
Reflection



- How is resilience impacted by coupling of services?
- What are meaningful strategies, and what are acceptable “degradations of service”?
- How far should you take resilience?
- Resilience through code vs. infrastructure - pros and cons?
- How to test for resilience?

Resilience
Conclusion



Need diverse skills:

- Implement resilience in code or infrastructure, decoupling of services
- Testing for resilience, configuring thresholds (for circuit breakers, retry, …), monitoring
- Tools & technology: Istio, Resilience4J, Spring Cloud, Opossum, …

Impact of a “bad microservice design”:

- Services less self-sustained => more failure points; loosely coupled system is naturally resilient
- Points of failure much harder to address in tightly coupled system, e.g. meaningful fallback
- True story: complexity of resilience got so high that there was a “resilience for the resilience”

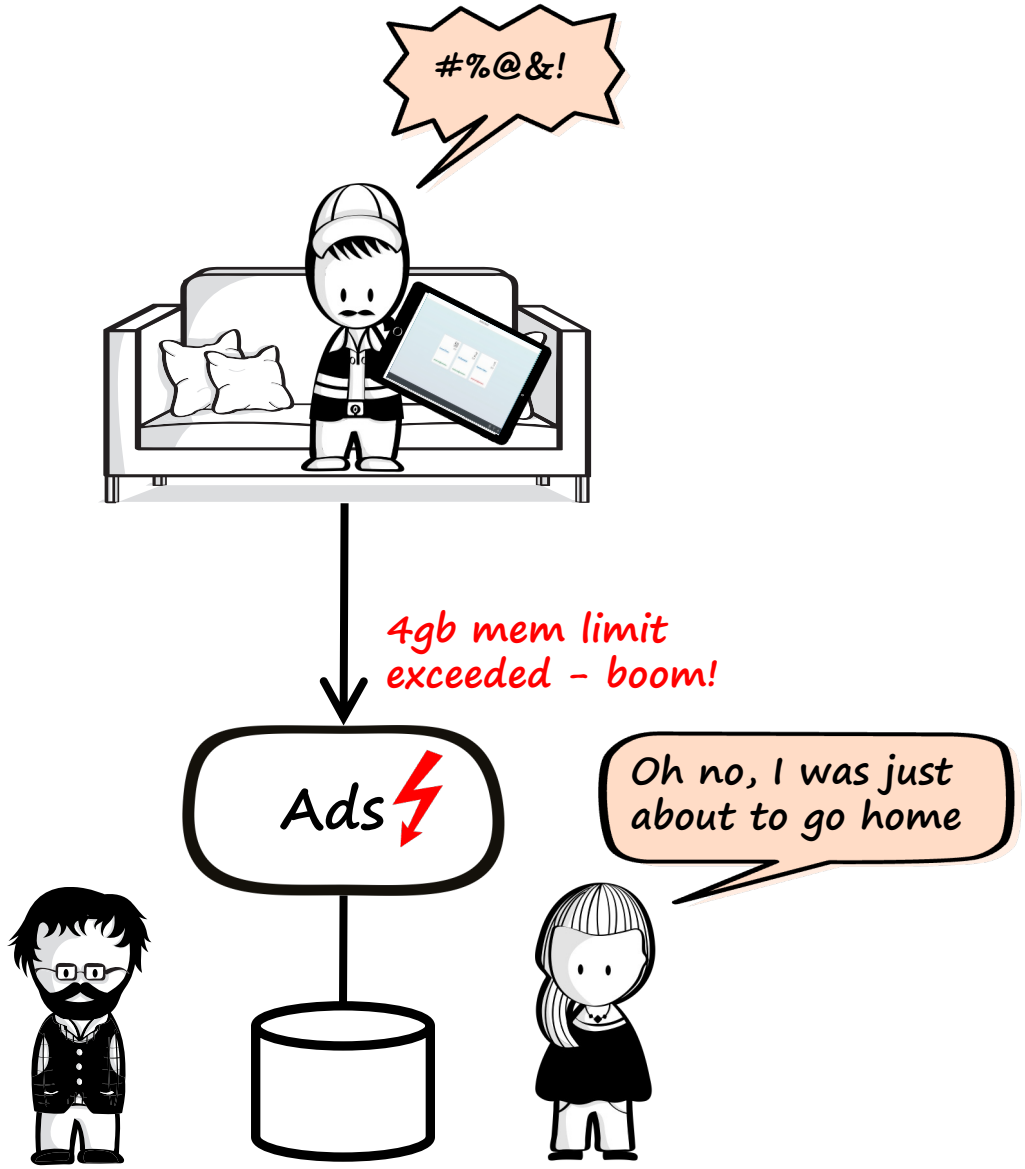
DevOps & Cloud Native Concepts for High Availability

Overview

Summary

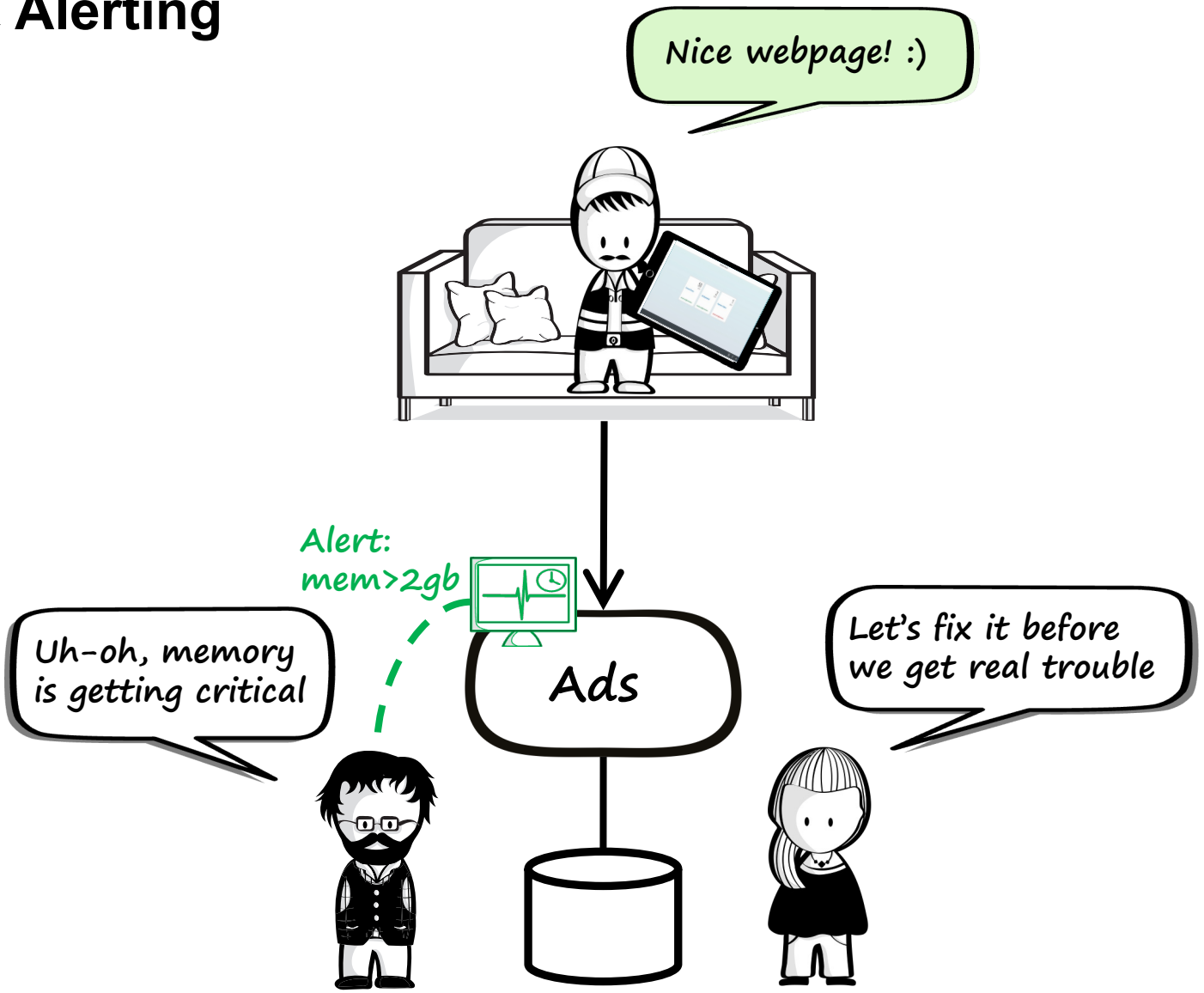
Monitoring & Alerting

Problem



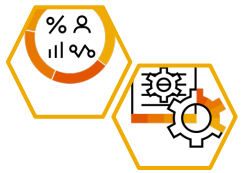
Monitoring & Alerting

Solution



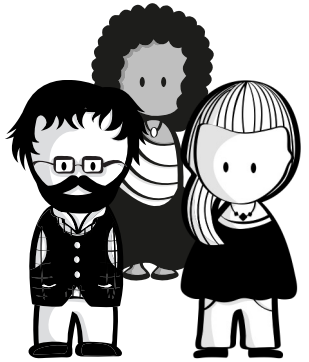
Monitoring & Alerting

Reflection



- Which metrics are helpful and how to set the thresholds right?
- Can all metrics, monitoring and alerting be done with one single approach?
- In which cases to notify the team, and how to notify the team?
- Should the team also be notified during the night?
- What if alerts come in (too) frequently?

Monitoring & Alerting
Conclusion



Need diverse skills:

- Logging in code, correlating logs cross services, log retention, monitors, alert triggers
- Identifying & refining which metrics are critical from a technical & business perspective
- Tools & technology: Kibana, Grafana, Istio, Dynatrace, Uptime, Pingdom, query languages, …

Impact of a “bad microservice design”:

- Need to monitor *much* more aggressively as issues will cascade & amplify, increases complexity
- Likely needs centralization due to complexity, losing ability to monitor service-specific metrics
- Response time tends to be slower due to the increased complexity

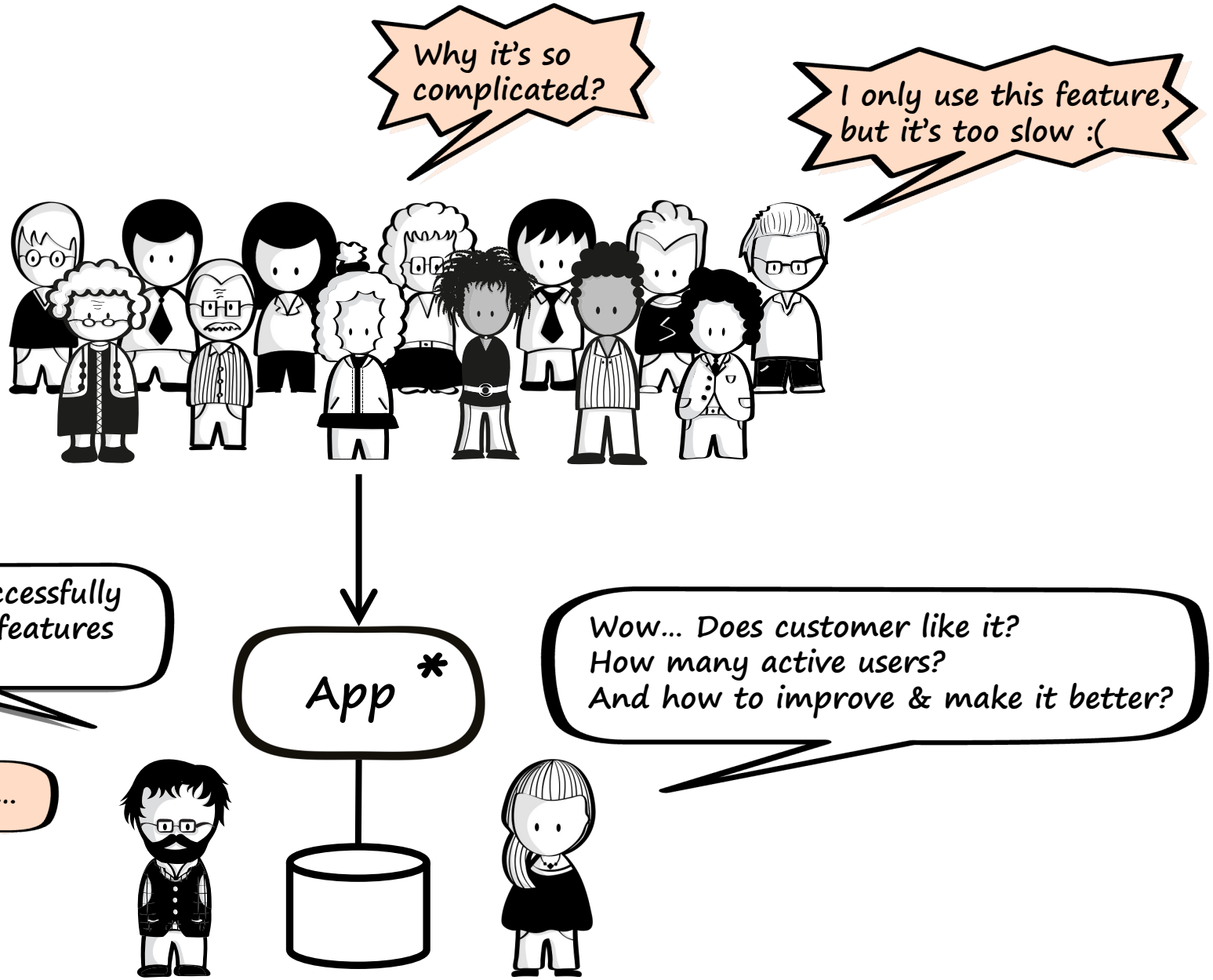
DevOps & Cloud Native Concepts for High Availability

Overview

Summary

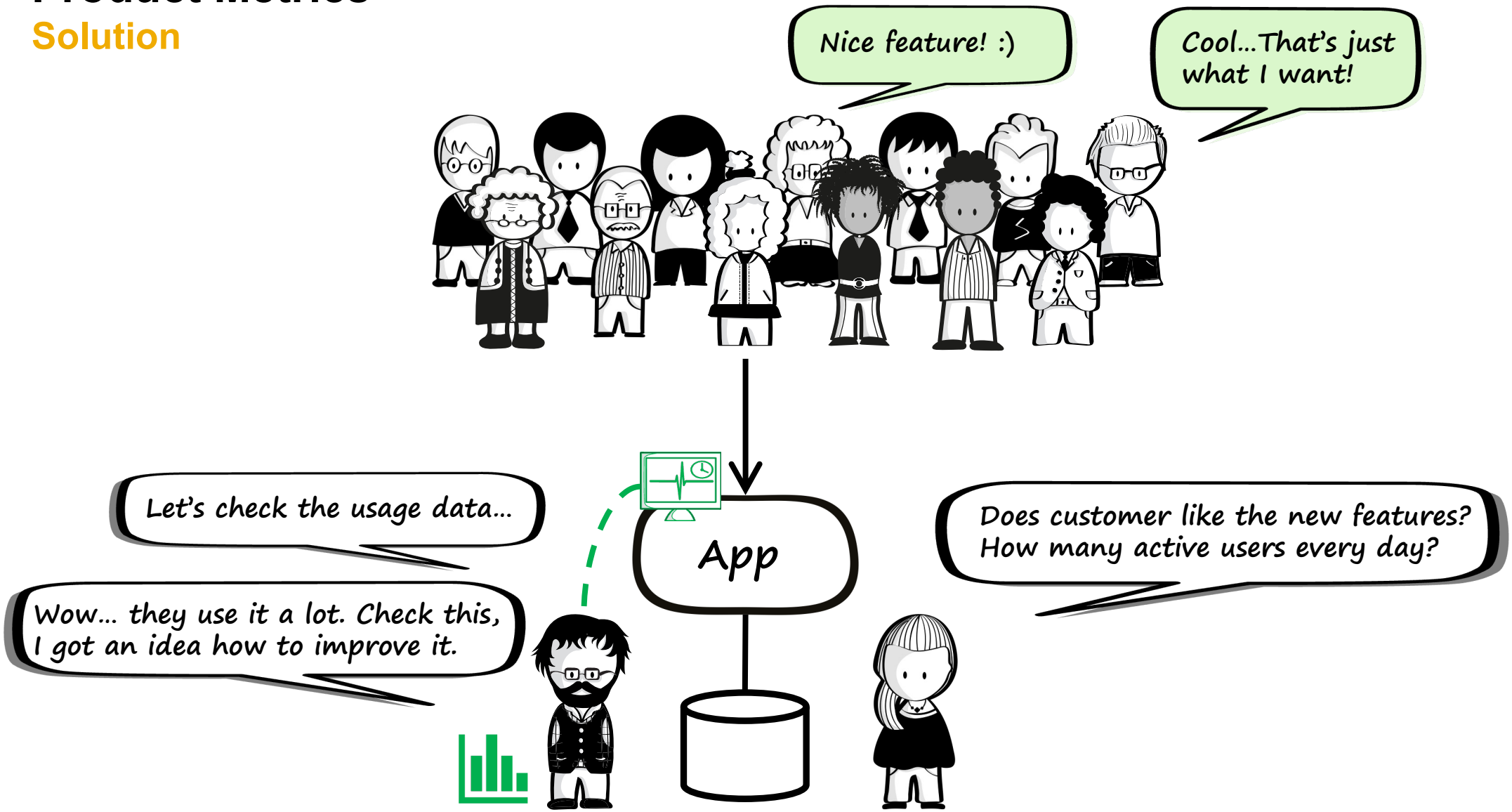
Product Metrics

Problem



Product Metrics

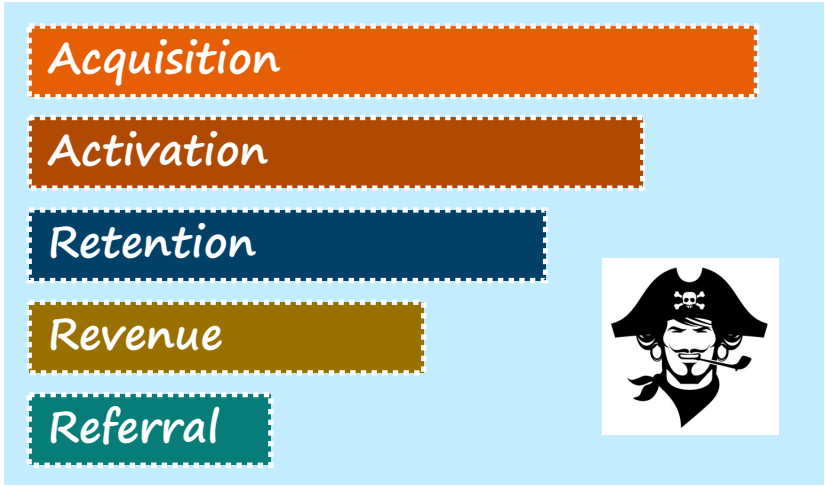
Solution

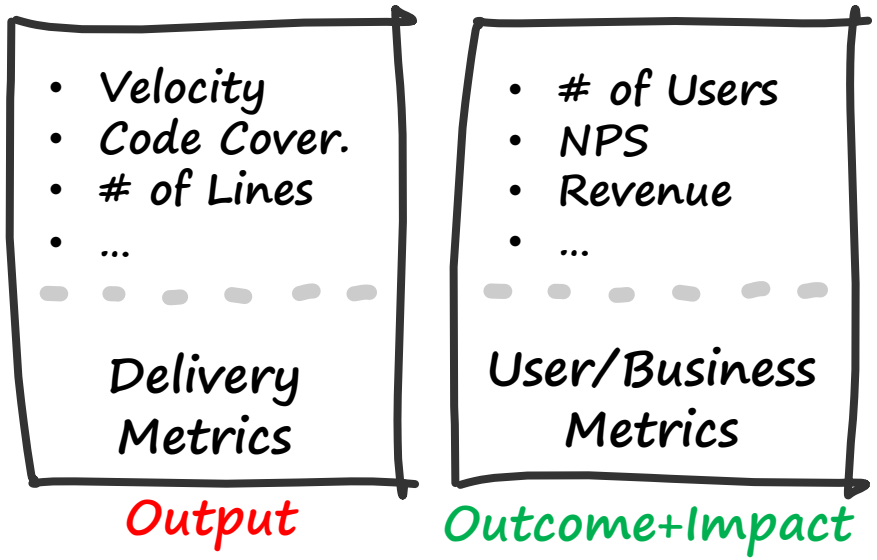


Product Metrics

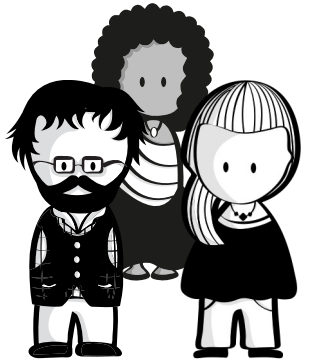
Reflection

- What can you measure and how to measure?
- Who decides on the metrics?
- Anything you need to consider when collecting metrics?
- Are there other kinds of metrics than product metrics?
- Is there some relation between those metrics?





Product Metrics
Conclusion



Need diverse skills:

- Implement usage measurement: UI/server/DB; setup & configure repository & dashboard
- Define metrics + criteria (standard / feature-specific), consider regulations (data privacy)
- Tools & technology: SAC, HTTP, SQL/analytics tools, …

Impact of a “bad microservice design”:

- Harder to collect meaningful metrics as features tend to be scattered
- Infrastructure tends to be managed centrally in tightly coupled systems, putting restrictions
- Need to align across multiple teams on metrics and granularity / collection strategy

DevOps & Cloud Native Concepts for High Availability

Overview

Canary Release	Zero-Downtime Deployment	Zero-Downtime DB Migration
Resilience	Monitoring & Alerting	Product Metrics

Summary

Q&A

…and Open Discussion