DevOps aus der Sicht eines Architekten

Continuous Lifecycle 2019

Stefan Kühnlein, Senior Solution Architect

Agenda

1. DevOps and ITIL
2. Software Architecture
3. Deployment Strategies
4. Challenge Database Migration
5. Service Mesh

DevOps and ITIL

DevOps life cycle processes

- Treat Operations personnel as first-class stakeholder
- Get their input when devloping requirements

- Small teams
- Limited coordination
- Unit tests

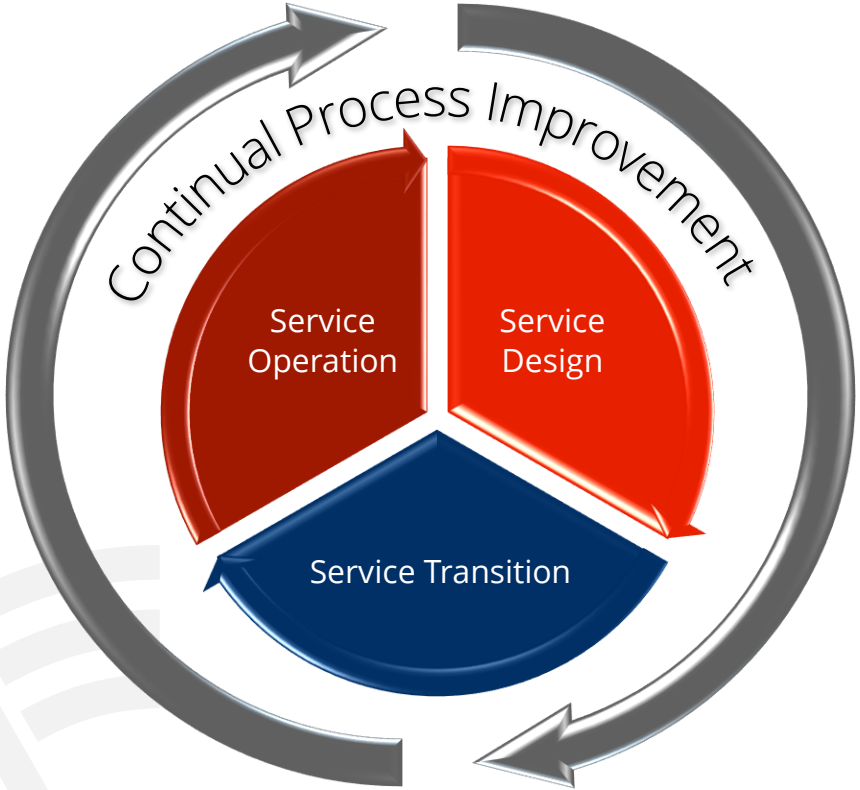
- Build tools
- Supports continuous integration

- Automated testing
- User acceptance testing

- Deployment tools
- Supports continuous deployment

- Monitoring
- Responding to error conditions

ITIL - Service Life Cycle



Service Design

Considerations when designing a service

- What automation is going to be involved as a portion of the service?
- What are the SLAs for the service?
- What are the personnel requirements for the service?
- What are the compliance implications of the service?
- What are the implications for capacity?
- What are the business continuity implications of the service?
- What are the information security implications of the service?

Service Transition

Service transition involves extending the knowledge of the new or revised service to the users and the immediate supporters of that service within operations.

- Are all features of the old version supported in the new version?
- Which new feature are introduced? How will the scripts for the deployment tool be modified, and who is responsible for that modifications?
- Will the new version require or support a different configuration of servers, which includes both testing/staging and production servers?

Service Operation

During operation, events are defined ty ITIL, as “any detectable or discernible occurrence that has significance for the management of the IT infrastructure of the delivery of IT service and evaluation of the impact a deviation might cause to the service.”

- Events of interest during operation include
  - Status information from systems and infrastructure
  - Environmental conditions, such as smoke detectors
  - Software license usage
  - Security information (e.g., from intrusion detection)
  - Normal activity, such as performance metrics from servers and applications

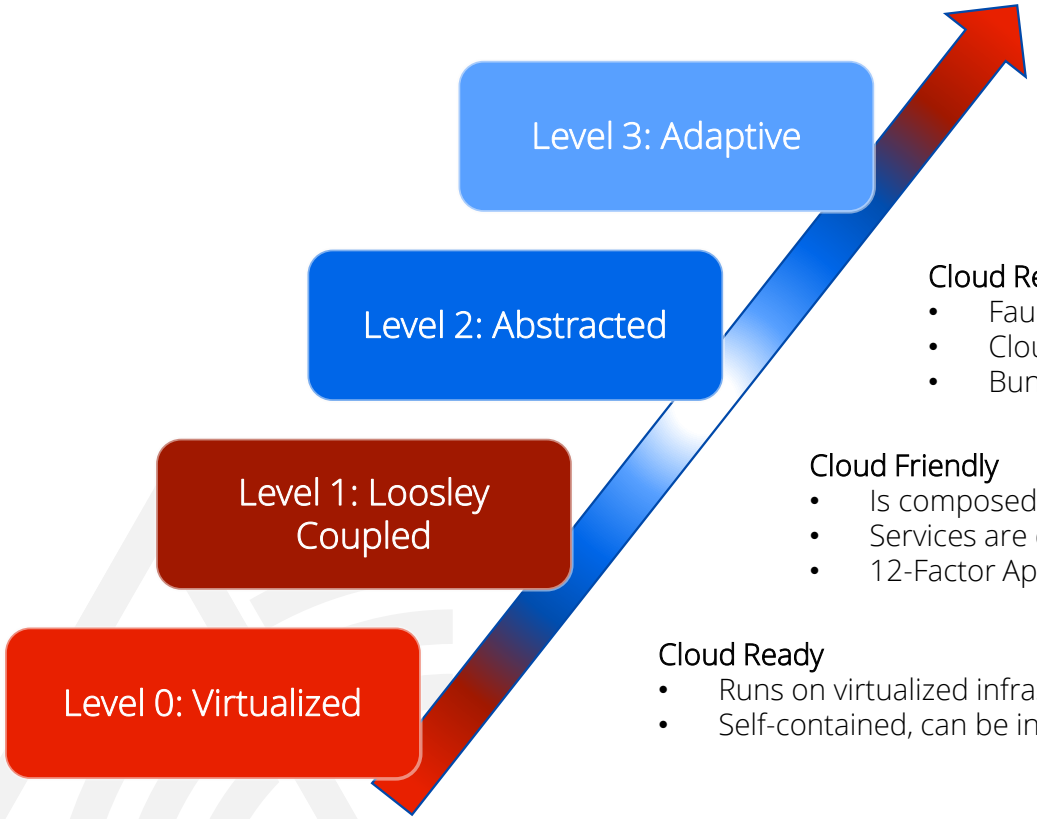
Service Operation – Incident Management

- Core activities of incident management are
  - Logging the incident
  - Categorization and prioritization
  - Initial diagnosis
  - Escalation to appropriately skilled or authorized staff, if needed
  - Investigation and diagnosis, including an analysis of the impact and scope of the incident
  - Resolution and recovery, either through the user under guidance from support staff, through the support staff directly, or through internal or external specialists
  - Incident closure, including recategorization if appropriate, user satisfaction survey, documentation and determination if the incident is likely to recur

Incident management is one of the areas where DevOps is changing the traditional operations activities

Software Architecture

Maturity Model of native Cloud-Applications



Cloud Native
- Microservice architecture
- API-first design

Cloud Resilient
- Fault-tolerant and resilient design
- Cloud-agnostic runtime implementation
- Bundled metrics and monitoring

Cloud Friendly
- Is composed of loosely coupled services
- Services are discoverable by name
- 12-Factor App Principles

Cloud Ready
- Runs on virtualized infrastructure
- Self-contained, can be instantiated from an image or script

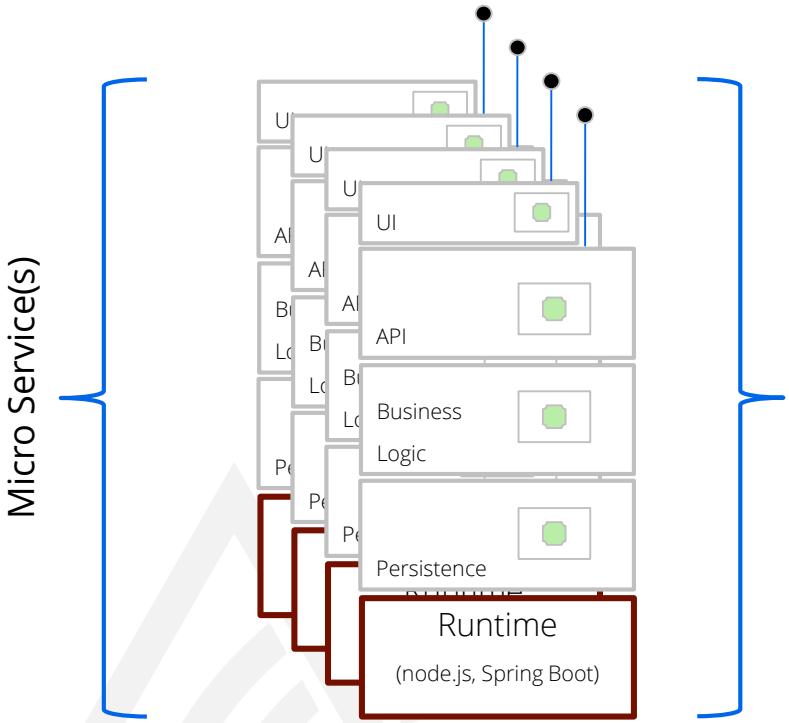
12 Factor App

1. Codebase
2. Dependencies
3. Configuration
4. Backing Services
5. Build, Release Run
6. Processes
7. Port Binding
8. Concurrency
9. Disposability
10. Development-Production Parity
11. Logs
12. Admin Processes

Design Principles

- Design for Distribution: Containers; microservices; API driven development
- Design for Configuration: One image; multiple environments
- Design for Resiliency: Fault-tolerant and self-healing
- Design for Elasticity: Scales dynamically
- Design for Delivery: Short roundtrips and automated provisioning
- Design for Performance: Responsive; concurrent; resource efficient
- Design for Automation: Automated Dev & Ops tasks
- Design for Diagnosable: Cluster-wide logs, metrics and traces

Design for Resiliency

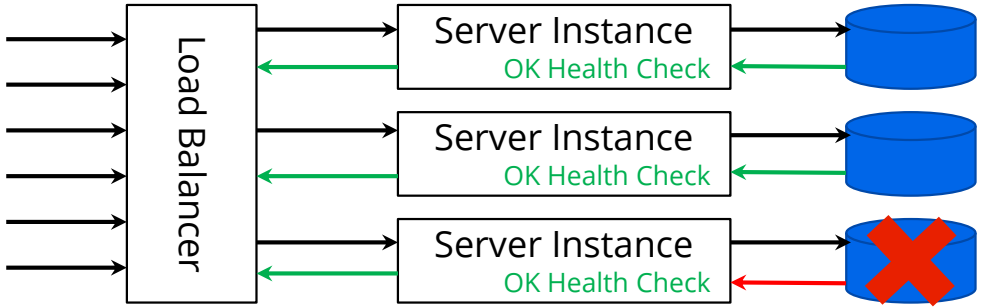


Micro Service Resilience Pattern

Failure Prevention • Fail-fast • Circuit Breaker (Prevent cascading failures) • Isolation • Loose Coupling	**Failure Detection** • Circuit Breaker (Monitoring connections) • TimeOut • Monitoring
Failure Mitigation • Circuit Breaker (Fallback-Default value) • Shed Load • Error Handler	**Recovery** • Retry

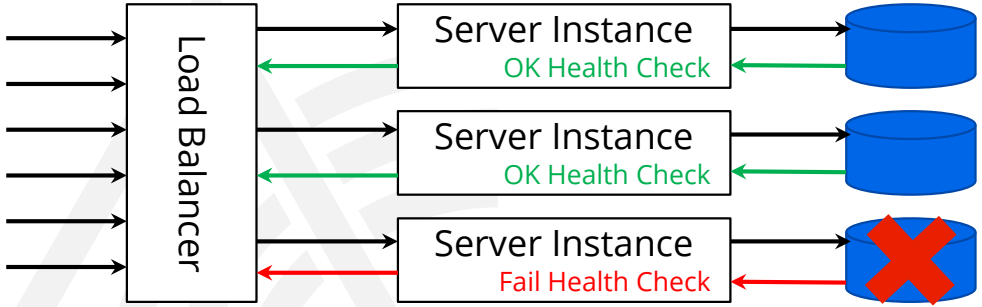
Shallow and Deep Health Checks

Shallow Health Check



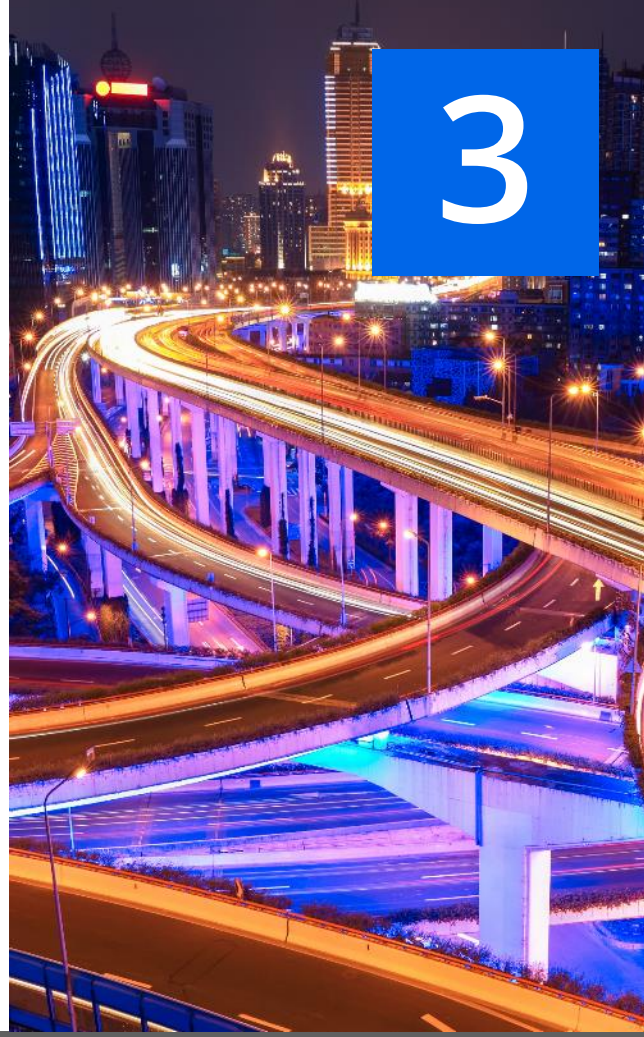
- Sample „Ping“
- Doesn‘t tell anything about the logical dependencies

Deep Health Check



- Gives more information about the health
- Harder to implement
- Health Checks are expensive

Deployment Strategies

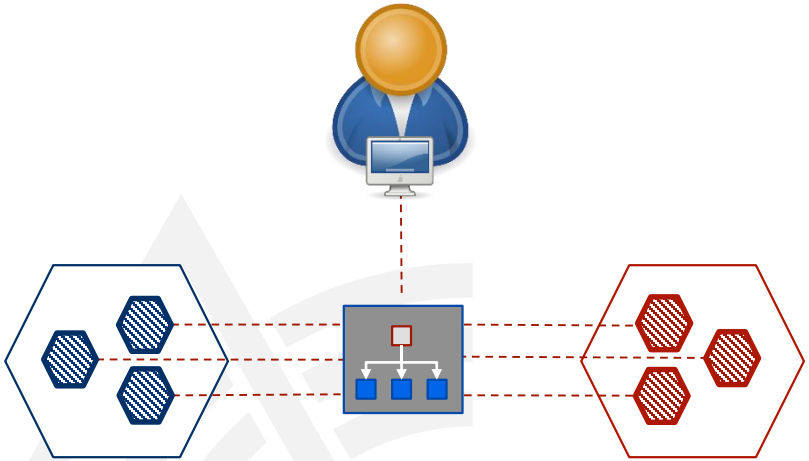


Overview Deployment Strategies

- **Recreate**
  Version A is terminated then version B is rolled out.
- **Ramped (Rolling Update)**
  Version B is slowly rolled out and replacing version A.
- **Blue- Green Deployment**
  Version B is released alongside version A, then the traffic is switched to version B.
- **Canary Deployment**
  Version B is released to a subset of users, then proceed to a full rollout.
- **A/B testing**
  Version B is released to a subset of users, then proceed to a full rollout.
- **Shadow**
  Version B is released to a subset of users under specific condition.

Recreate Strategy

The recreate strategy consists of shutting down version A and deploying version B after version A is down.
This strategy implies downtime of the service.

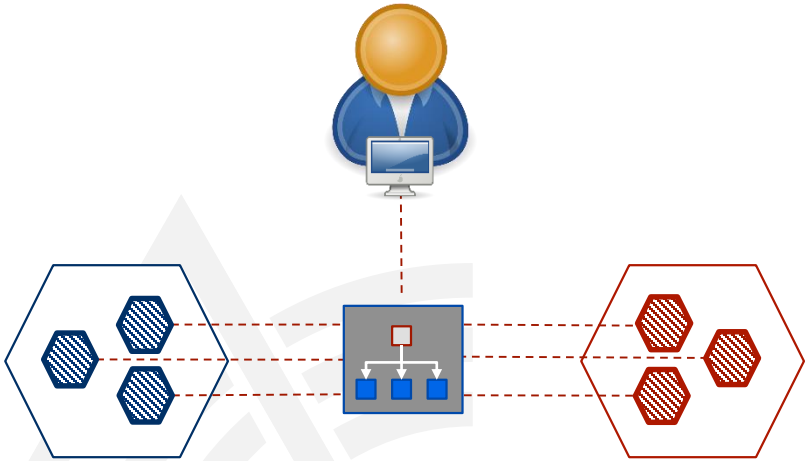


- Pros
  - Easy to setup
  - Application state entirely renewed
- Cons
  - High impact on the user

Ramped (Rolling Update) Strategy

The ramped deployment strategy consists of slowly out a version of an application by replacing instances one after the other.
The following parameters can increase the deployment time:

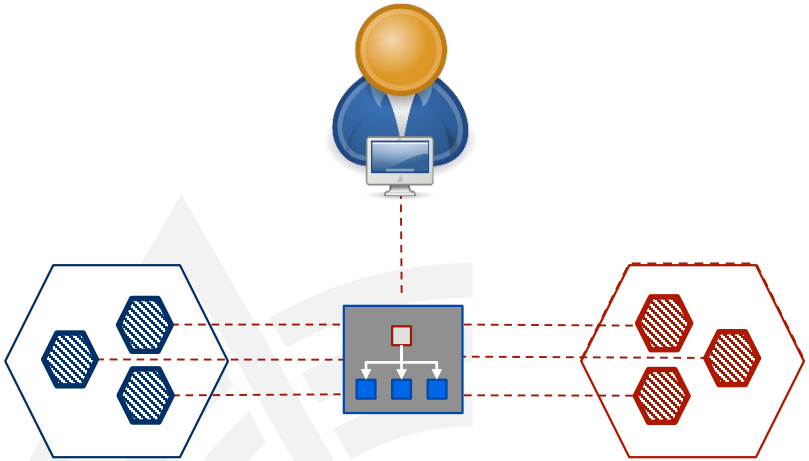
- Parallelism: Number of current instances to roll out
- Max. surge: How many instanced to add in addition
- Max. unavailable: Number of unavailable instances during rollout



- Pros
  - Easy to set up
  - Version is slowly released across instances
  - Convenient for stateful applications
- Cons
  - Rollout/rollback can take time
  - Supporting multiple APIs is hard
  - No control over traffic

Blue- Green Deployment

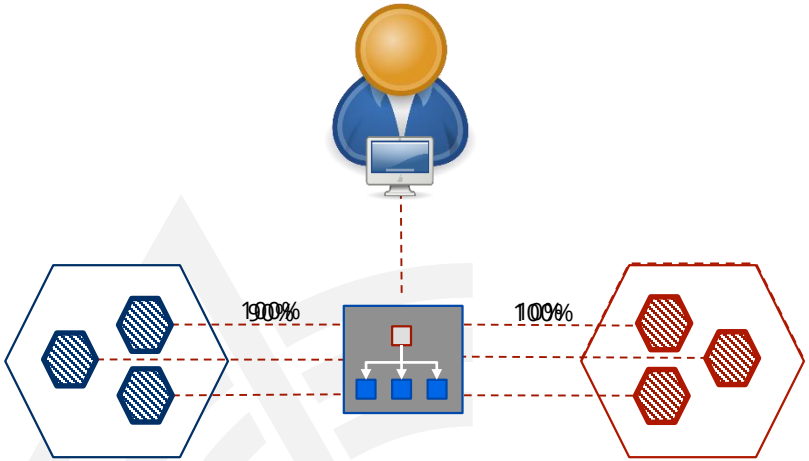
- The blue/green deployment strategy differs from a ramped deployment, version B (green) is deployed alongside version A (blue) with exactly the same amount of instances. After testing that the new version meets all the requirements the traffic is switched from version A to version B at the load balancer level.



- Pros
  - Instant rollout/rollback
  - Avoid versioning issue, the entire application state is changed in one go
- Cons
  - Expensive at it requires double the resources
  - Proper test of the entire platform should be done before releasing
  - Handling stateful applications can be hard

Canary Deployment

A canary deployment consists of gradually shifting production traffic from version A to version B. Usually the traffic is split based on weight. For example, 90 percent of the requests go to version A, 10 percent go to version B.



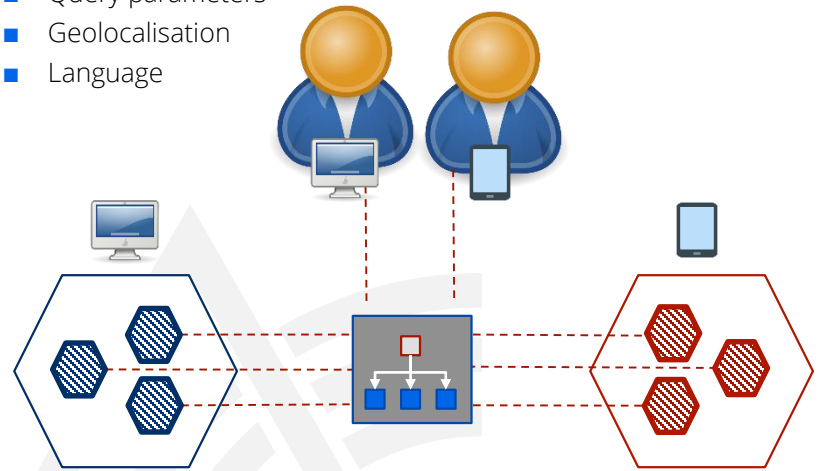
- Pros
  - Version released for a subset of user
  - Convenient for error rate and performance monitoring
  - Fast rollback
- Cons
  - Slow rollout

A/B Testing

A/B testing deployments consists of routing a subset of users to a new functionality under specific conditions.

This technique is widely used to test conversion of a given feature and only roll-out the version that converts the most.

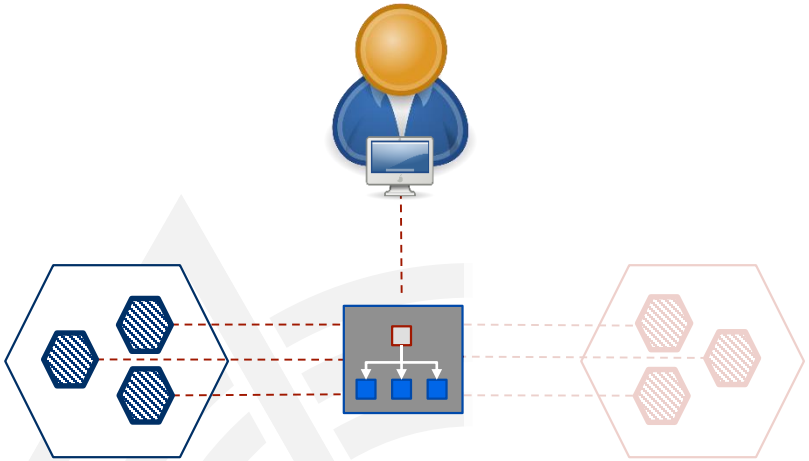
- Technology support: browser version, screen size, operating system, etc
- Query parameters
- Geolocalisation
- Language



- Pros
  - Several versions run in parallel
  - Full control over the traffic distribution
- Cons
  - Requires intelligent load balancers
  - Hard to troubleshoot errors for a given session, distributed tracing becomes mandatory

Shadow

A shadow deployment consists of releasing version B alongside version A, fork version A's incoming requests and send them to version B as well without impacting production traffic. This is particularly useful to test production load on a new feature.



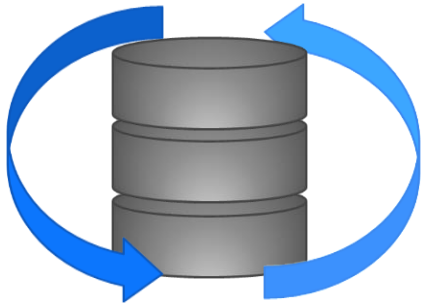
- Pros
  - Performance testing of the application with production traffic
  - No impact on the user
  - No rollout until stability and performance o f the application meet the requirements
- Cons
  - Expensive as it requires double the resources
  - Not a true user test and can be misleading
  - Complex to setup
  - Requires mocking service for certain cases

Challenge Database Migration

Challenge: Database Migration & Continuous Delivery

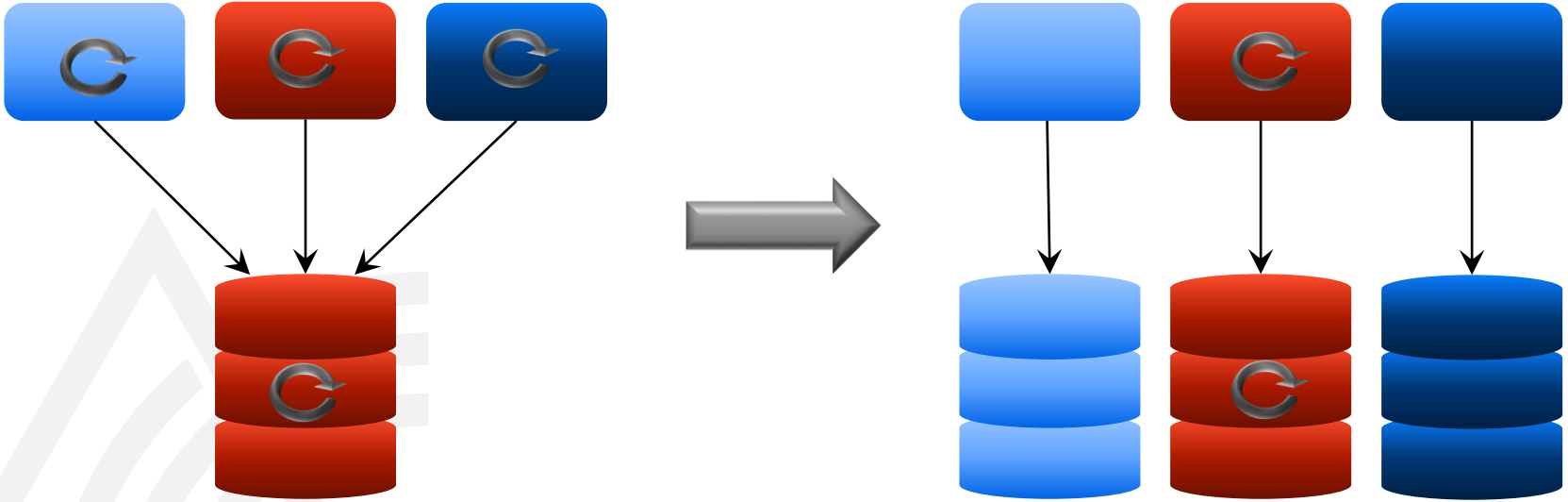
The problems with Database Updates in DevOps Environment

- We have to adapt existing data. This can be very difficult if there is a huge amount of data
- It's hard to rollback changes done in the database in case of critical error
- Database changes are difficult to test, because we would need a database similar to the production database base



Separated Data Sources

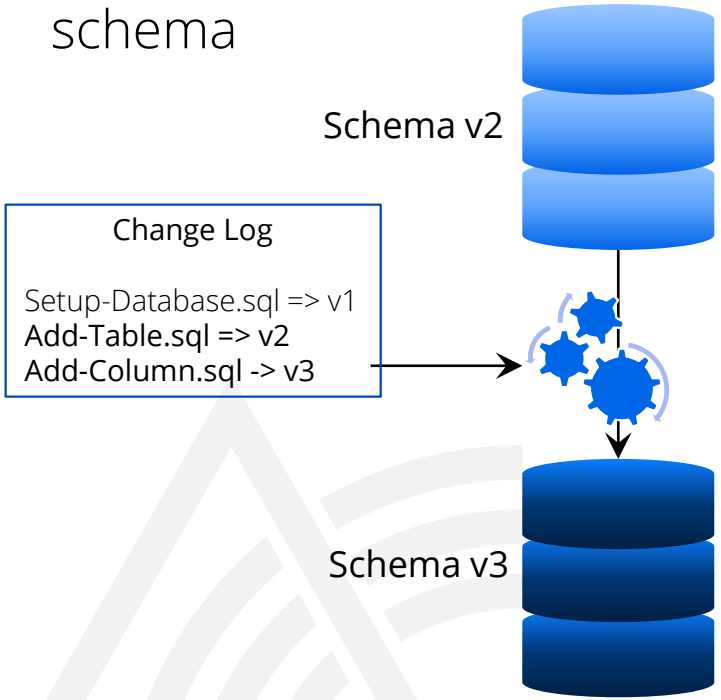
- Shared Database
  - If the database changed, we have to update all accessing components
- Separated Data Sources
  - It's only necessary to coordinate the database with one application



Schema Update and Data Migration

Relational Databases

- Relational Database have a fix schema



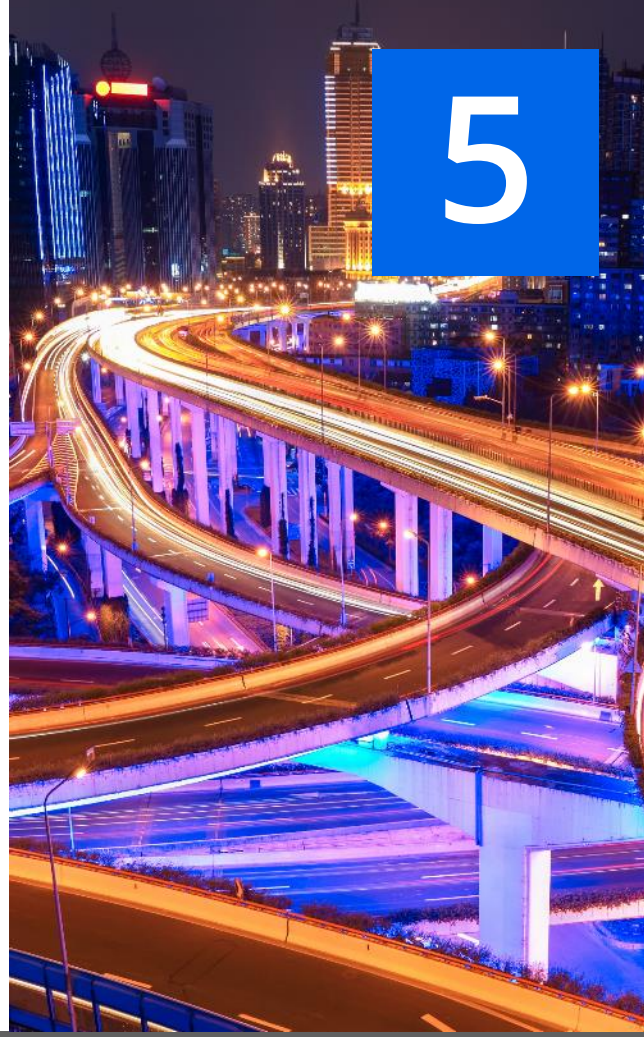
NoSQL Databases

- NoSQL databases typically don‘t have a restricted schema
- Data with the old and the new structure can exists at the same time
- We have make sure, that our application handles variable data structure including data migration

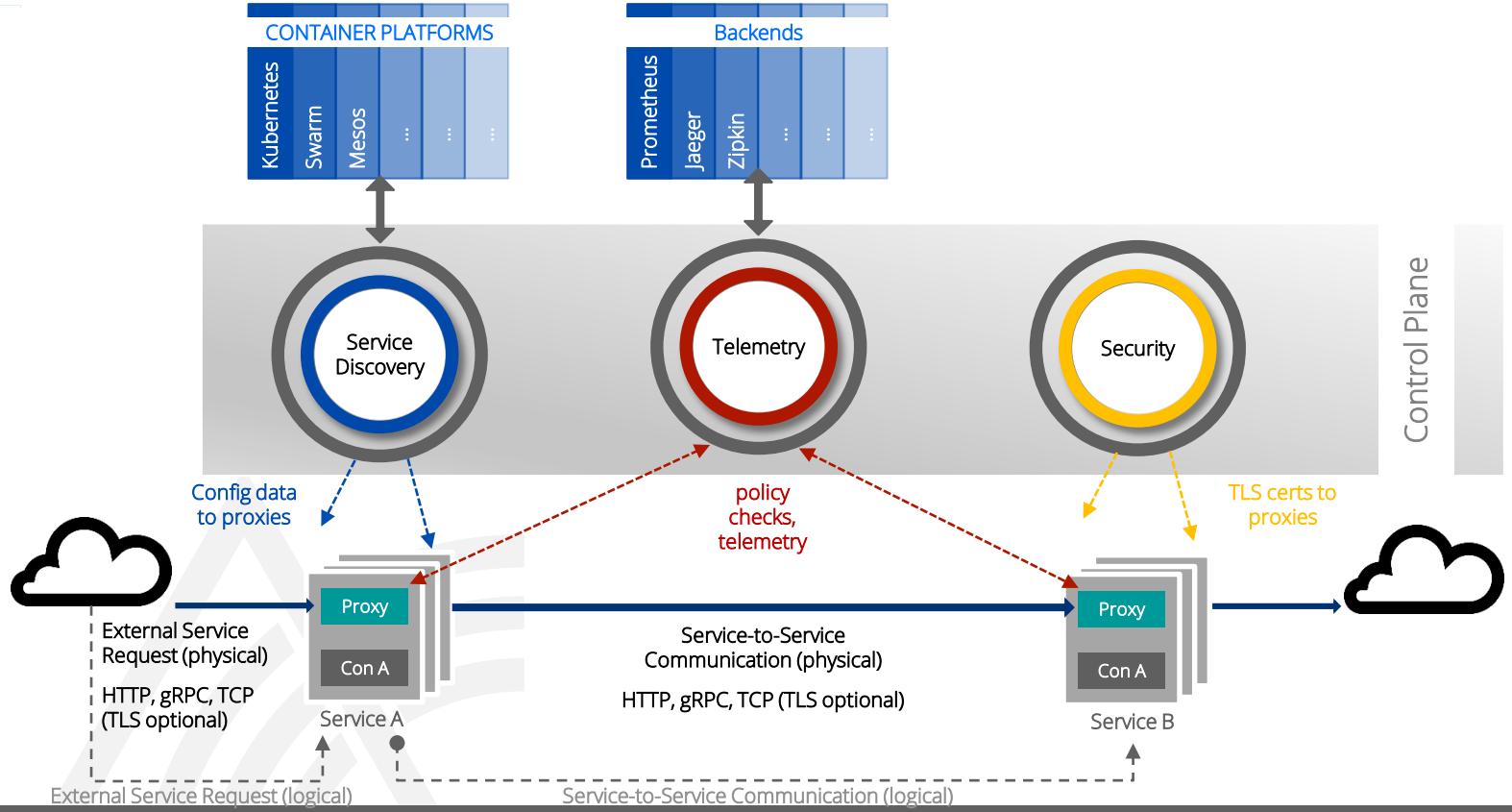
Summary

- Try to make the schema changes backwards compatible
- Separated data sources for each deployment unit
- Schema update and data migration
  - Relational databases: Use a migration tool to track changes and to automate updating the database
  - No SQL databases: No database update necessary as long as the application handles the different data structures
- Continuous deployment
  - It‘s easy without high availability constraint
  - It‘s hard when high availability is required -> Multiple intermediate versions of the application and update application instances step by step

Service Mesh



Architecture of a Service Mesh



Features of a Service Mesh

Traffic Management	Resiliency	Security	Observability
▪ Request Routing ▪ Load Balancing ▪ Traffic Shifting ▪ Traffic Mirroring ▪ Service Discovery ▪ Ingress, Egress ▪ API Specification ▪ Multicluster Mesh	▪ Timeouts ▪ Circuit Breaker ▪ Health Checks ▪ Retries ▪ Rate Limiting ▪ Delay & Fault Injection ▪ Connection Pooling	▪ mTLS ▪ Role-Based Access Control ▪ Workload Identity ▪ Authentication Policies ▪ CORS Handling ▪ TLS Termination	▪ Metrics ▪ Logs ▪ Traces

Weitere Fragen & Antworten

DevOps Best Practices

- Treat Ops as first-class citizens from the point of view of requirements.
- Made Dev more responsible for relevant incident handling
- Enforce the deployment process used by all, including Dev and Ops personnel
- Use continuous deployment
- Develop infrastructure code, such as deployment scripts, with the same set of practices as application code

Stefan Kühnlein

Senior Solution Architect

Weltenburger Str. 4
81677 München

stefan.kuehnlein@opitz-consulting.com
+49 173 7279307

WWW.OPITZ-CONSULTING.COM

@OC_WIRE

OPITZCONSULTING

opitzconsulting

opitz-consulting-bcb8-1009116