Architektur-Dokument für das Monitoring-Projekt

Einleitung und Überblick

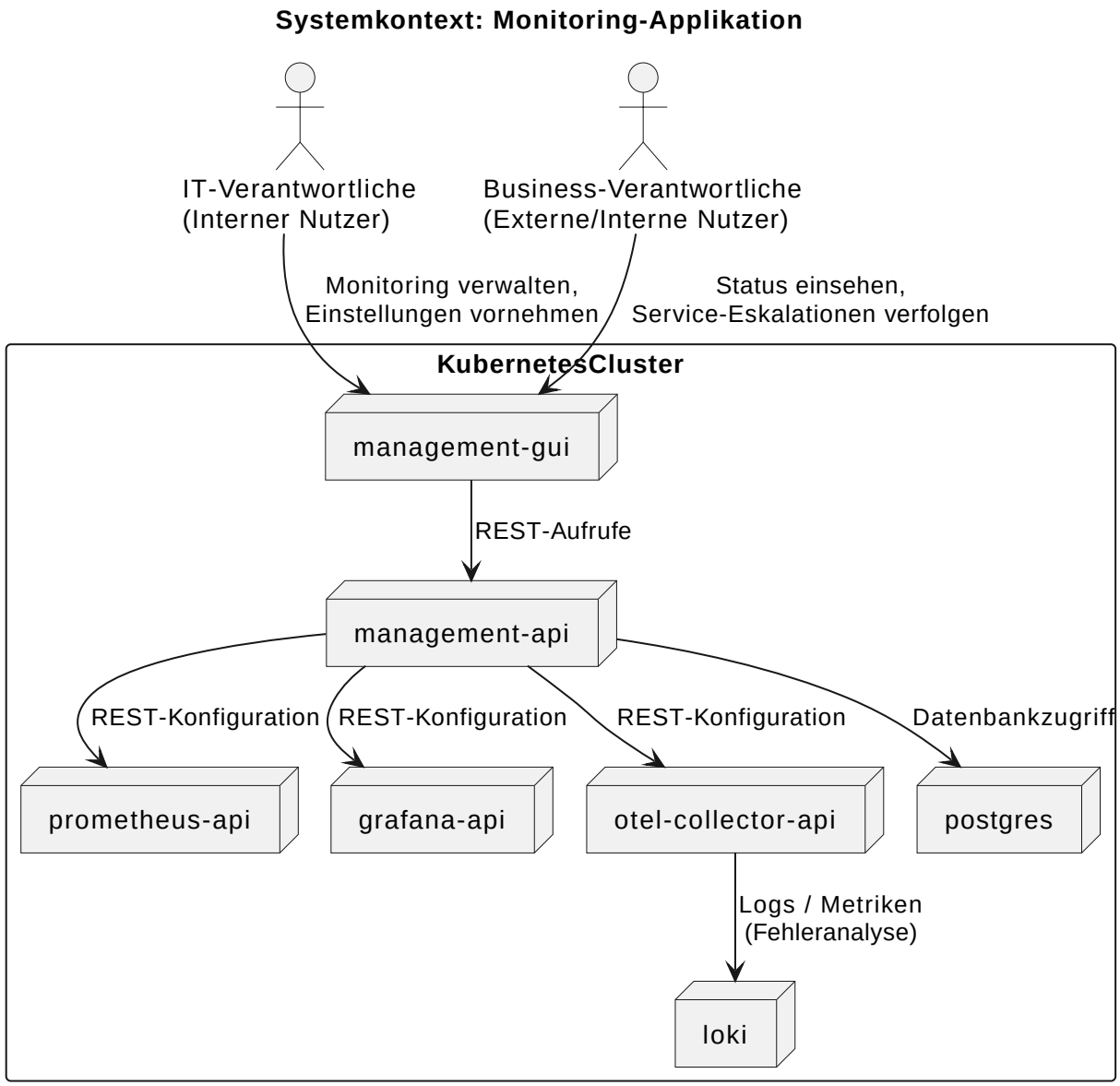
Notiz: Die im Dokument integrierten Diagramme werden von Github leider nicht gerendert.

Dieses Dokument beschreibt die Architektur der Monitoring-Applikation, welche die Verfügbarkeit, Performance und den Status von verschiedenen Services (Ports, HTTP-/HTTPS-Endpunkte usw.) überwacht. Die Lösung basiert auf einer Kubernetes-Installation mit den Containern:

- management-gui (Port 8091)
- management-api (Port 8092)
- prometheus-api (Port 8093) / prometheus (Port 9090)
- grafana-api (Port 8094) / grafana (Port 3000)
- otel-collector-api (Port 8095) / otel-collector (Port 4317/4318)
- postgres (Port 5432)
- loki (Port 3100)

Alle Container laufen in einem Namespace: "monitor". Der Zugriff von extern geschieht über einen NodePort in Kombination mit unserem Nginx Proxy. Die **prometheus-api**, **grafana-api** und **otel-collector-api** sind als "Sidecar"-Komponenten konzipiert, die REST-Endpunkte zur Laufzeitkonfiguration anbieten.

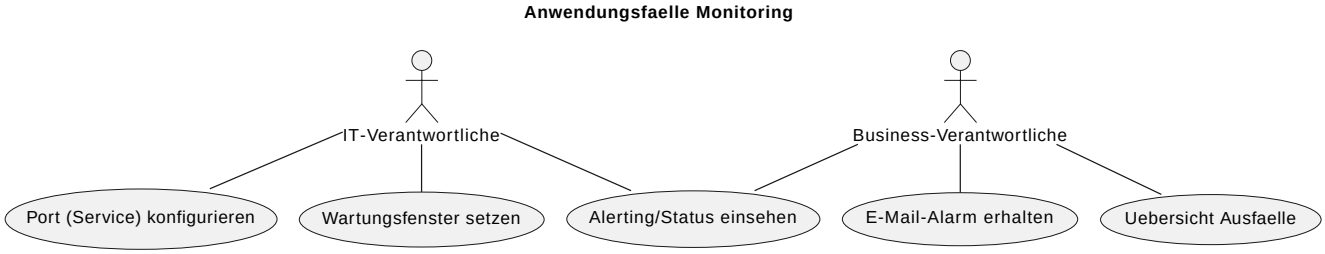
Systemkontext (vereinfacht)



Das Diagramm verdeutlicht die wichtigsten Akteure, ihre Zugriffe auf das System sowie die Datenflüsse zwischen den Containern. Neu hinzugefügt ist der Datenstrom von **otel-collector-api** zu **Loki**, um Log- und Telemetriedaten zu sammeln und für die Fehleranalyse bereitzustellen.

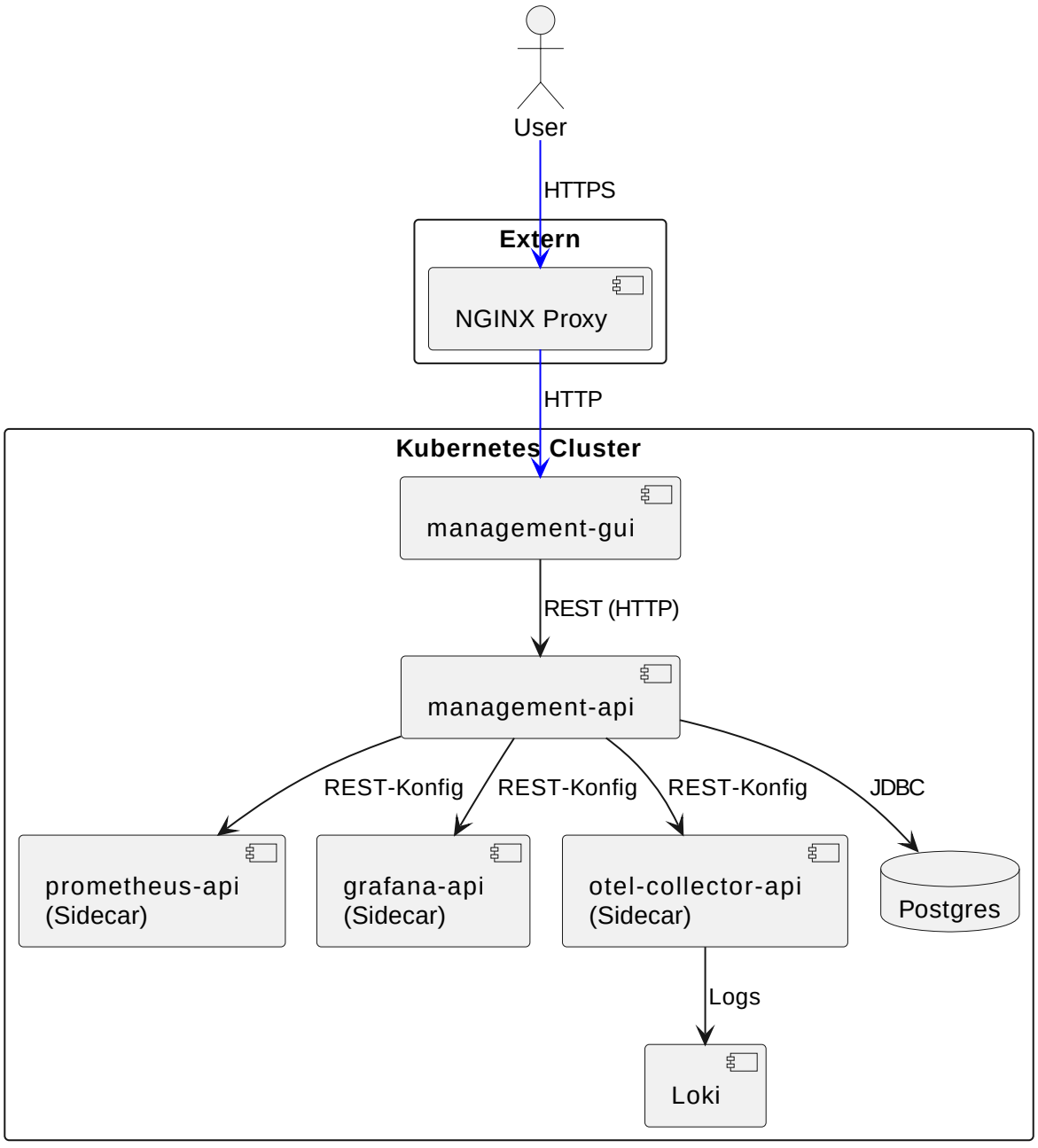
Anwendungsfalldiagramm

Nachfolgend ein Beispiel-Anwendungsfalldiagramm mit den zentralen Use-Cases, welche von IT-Verantwortlichen und Business-Verantwortlichen genutzt werden.



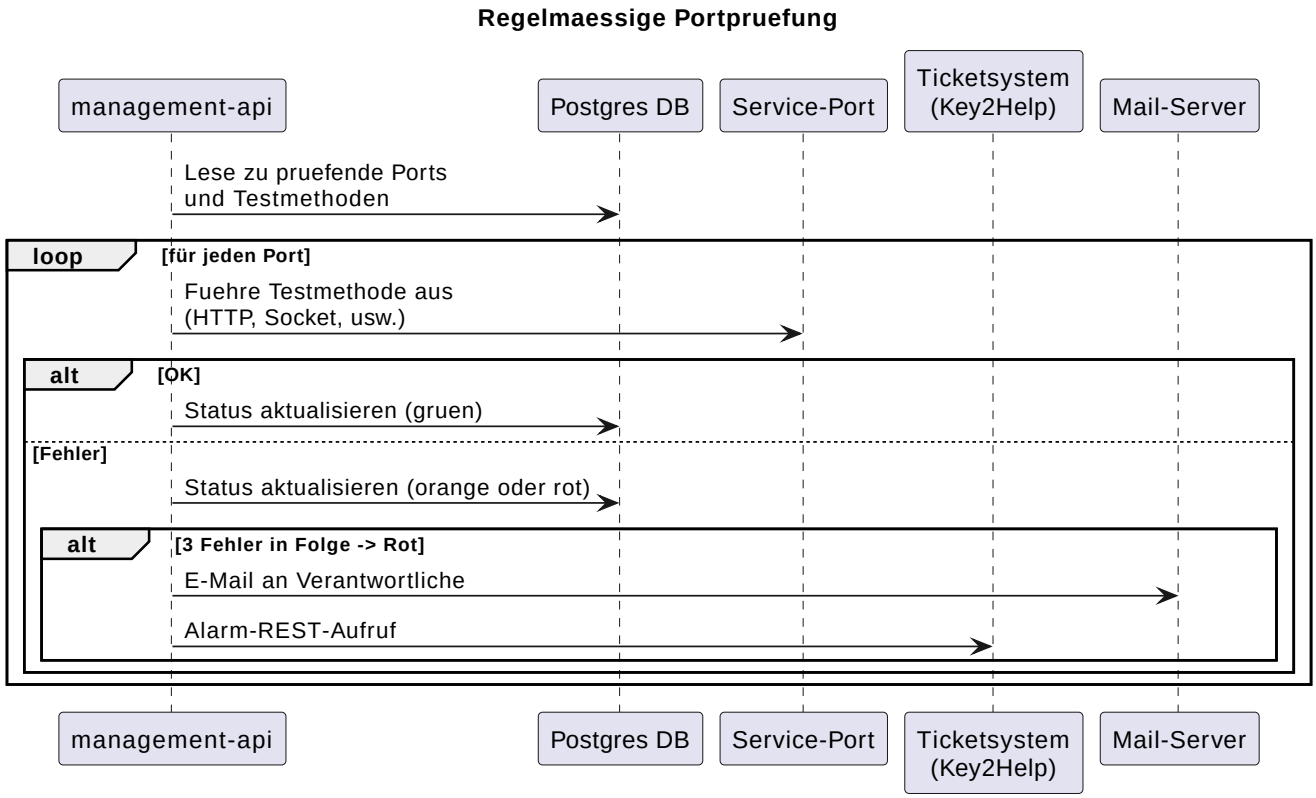
High-Level Komponentendiagramm

High-Level Komponentendiagramm (mit NGINX Proxy und Kubernetes-Rahmen)

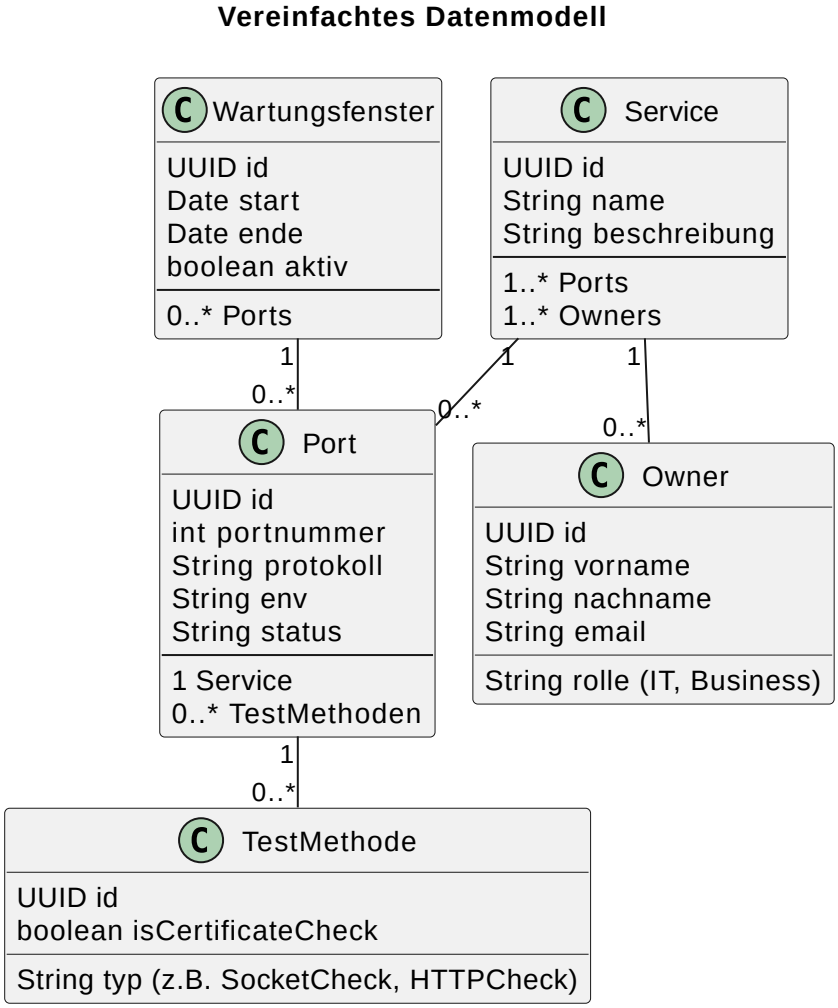


Die **Sidecar-APIs** (prometheus-api, grafana-api, otel-collector-api) sind jeweils zuständig für die Konfiguration und Datenaufnahme bzw. -weiterleitung des Hauptsystems (Prometheus, Grafana, OpenTelemetry Protokoll). Sie laufen zwar als eigenständige Container, müssen aber zusammen mit der Hauptanwendung gestartet und verwaltet werden.

Beispiel-Sequenzdiagramm: Regelmässige Port prüfung



Datenmodell (vereinfacht)



Infrastruktur und Deployment

1. **Kubernetes Single-Instanz**:
   - DEV, TEST, PROD getrennt, je eine VM (oder später mehr).
2. **GitLab-CI/CD**:
   - Automatisiertes Deployment, Container Images werden gebaut und über Terraform.
   - Leicht skalierbar dank Kubernetes.
3. **Storage & Backup**:
   - Daten in Postgres (Container oder externer DB-Host).
   - VMware-Snapshots für schnelle Wiederherstellung.
4. **Security**:
   - Zugriff intern via HTTP im gleichen Namespace.
   - Nach aussen via HTTPS über Nginx.
   - Admin-Zugang und Standardberechtigungsverfahren für Alarm-Empfänger und Systemrechte.

Sidecar-Konzept

prometheus-api, **grafana-api** und **otel-collector-api** sind jeweils Sidecar-Dienste bzw. separate Container. Sie werden in Kombination mit den Hauptdiensten (Prometheus, Grafana, Otel Collector) deployed und ermöglichen:

- Laufzeitkonfiguration (z.B. per REST-API Befehle für Dashboards oder Prometheus-Abfragen)
- mögliche Erweiterungen/Plugin-Funktionalität, ohne den Hauptdienst direkt anzupassen
- Eigenständige Versionierung und Wartung

Oft sind Sidecars nah an ihren Hauptcontainern gekapselt. Dadurch kann man die Logik (z.B. Authentifizierung, Zieldatenverarbeitung) flexibel anpassen.

Qualitätsanforderungen

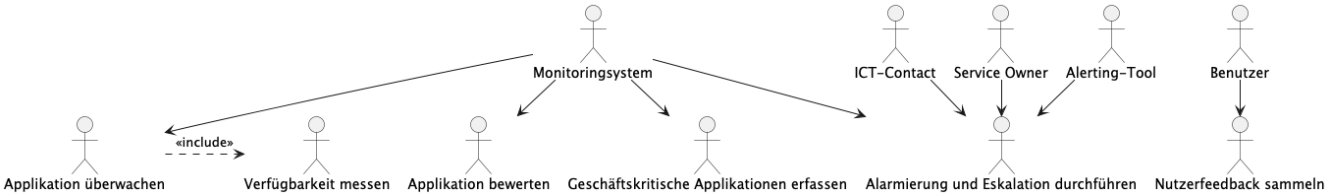
- **Performance**: Durch Containerisierung und Skalierung kann man bei Bedarf weitere Instanzen hochfahren.
- **Verfügbarkeit**: Ein (zukünftig) vollwertiges Kubernetes-Cluster ermöglicht Redundanz, je nach Ausbaustufe.
- **Wartbarkeit**: Alle Definitionen liegen in Gitlab, Container kännen lokal getestet werden, automatisierte Rollouts, klare Module.
- **Sicherheit**: Keine Personendaten, interne E-Mail-Benachrichtigungen, generelle Hardening-Massnahmen (Images, Netzwerk).

Zusammenfassung

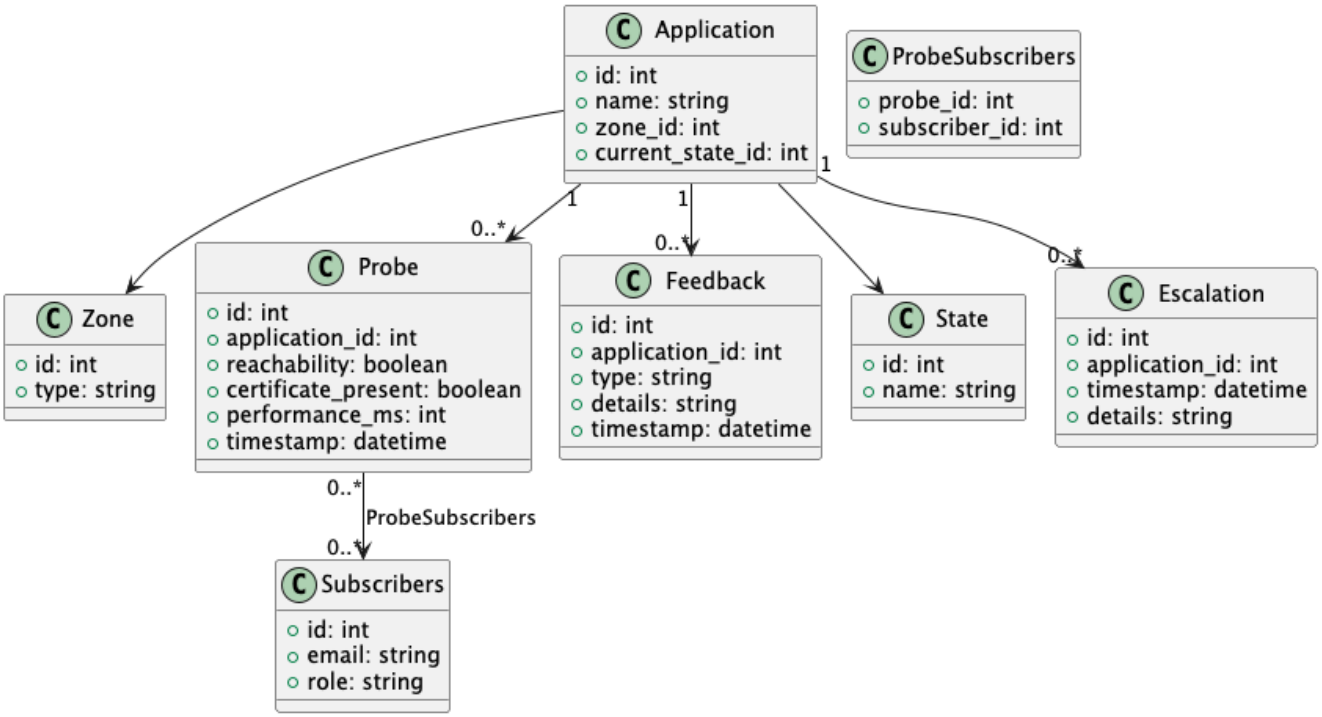
Diese Architektur deckt die Anforderungen ab: regelmässige Prüfungen von Ports mit verschiedenen Testmethoden, Alerting bei Fehlern, flexible Konfiguration per GUI, Datenhaltung in Postgres und Visualisierung/Monitoring in Grafana/Prometheus. Die **otel-collector-api** sendet zur Fehleranalyse Telemetrie an Loki. Sämtliche Module laufen in einem Kubernetes-Cluster und sind dank CI/CD in wenigen Minuten ausrollbar oder aktualisierbar.

Verfeinerte Modelle

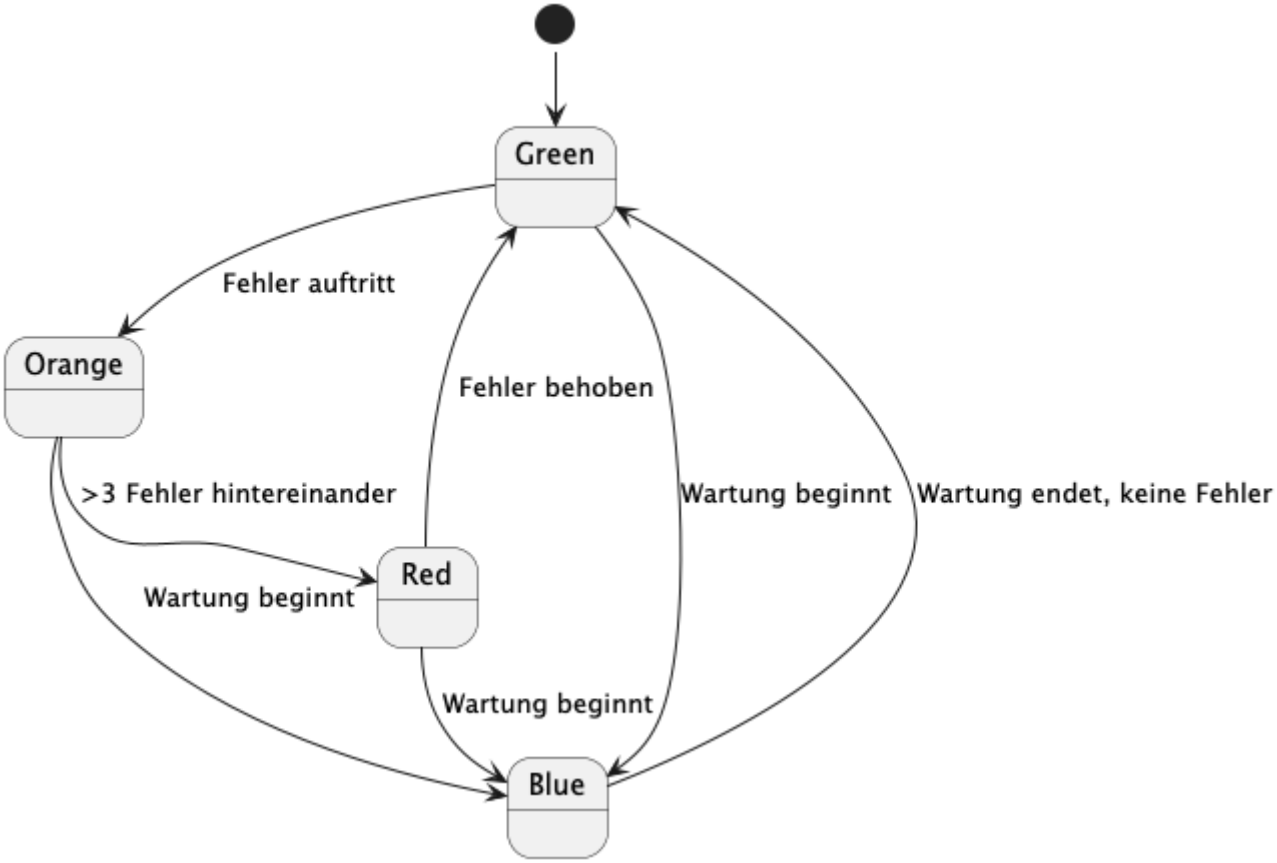
Anwendungsfalldiagramm



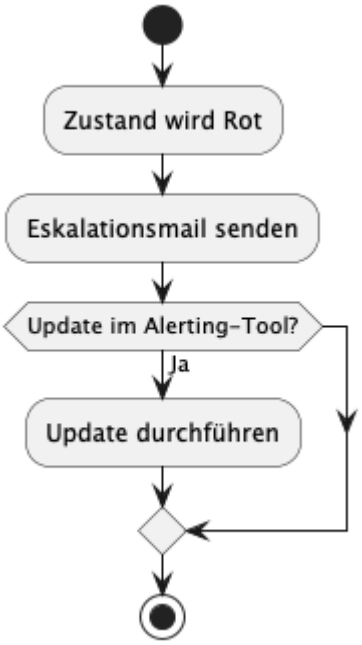
Klassendiagramm (nicht final)



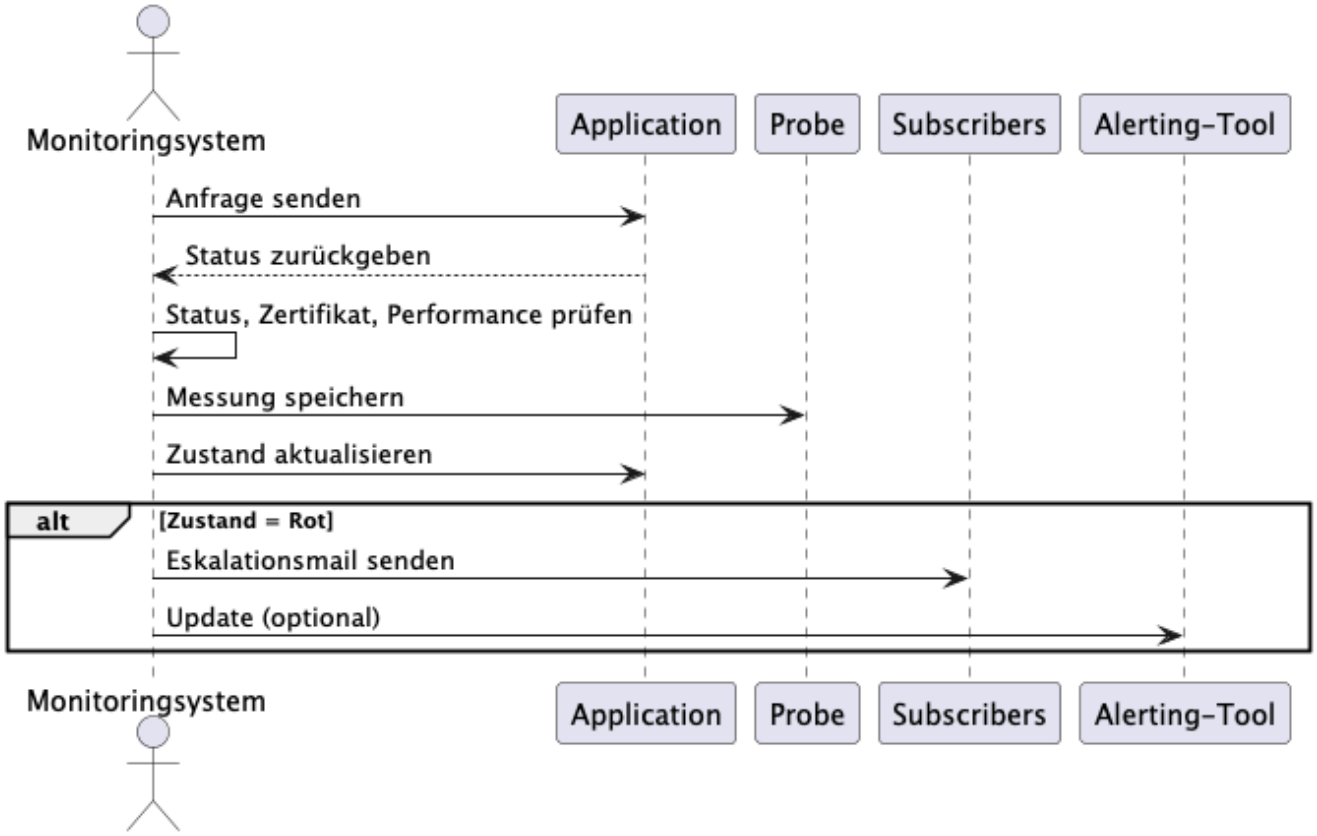
Zustandsdiagramm



Aktivitätsdiagramm



Sequenzdiagramm



Notizen

Falls der Container nicht startet:

Finde den Container mit:

```
kubectl get pods -n monitoring
```

Liste der Container:

```
NAME                                READY   STATUS                       RESTARTS
AGE
grafana-local-7b76f9749d-5j8hp      2/2     Running                      0
8m31s
loki-local-7569b86798-xv625         0/1     CreateContainerConfigError   0
8m31s
management-api-local-7785bb4f8-78tdp   1/1     Running                   0
8m32s
management-gui-local-85fdcf589d-p9lk9  1/1     Running                   0
8m32s
otel-collector-local-576bf59844-pkng7  2/2     Running                   0
8m31s
postgres-local-7c86bffbcc-s6qw6     1/1     Running                      0
8m31s
prometheus-local-fb664bb6f-6z8wv    2/2     Running                      0
```

```
8m31s
```

Wähle den Namen des Containers, der fehlschlägt und Analysiere das log:

```
kubectl describe pod loki-local-7569b86798-xv625 -n monitoring
```

Lösche den Container:

```
kubectl delete pod loki-local-7569b86798-xv625 -n monitoring
terraform apply
```