CI/CD

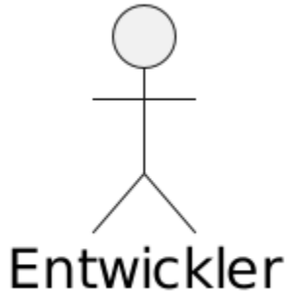
Continuous Integration / Continuous Delivery

Maurice Müller

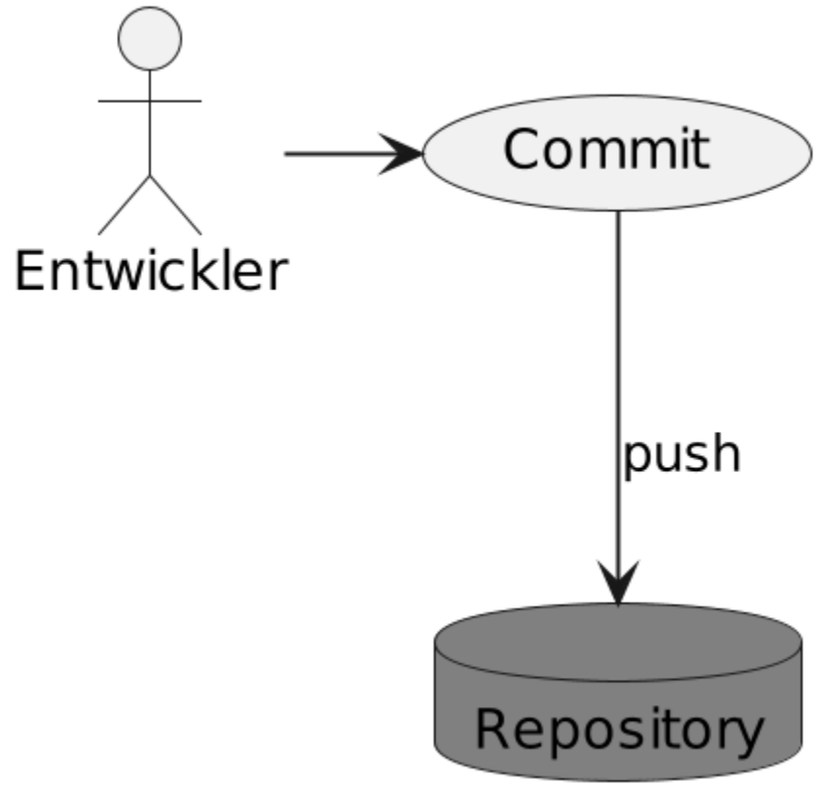
2024-04

Continuous Integration

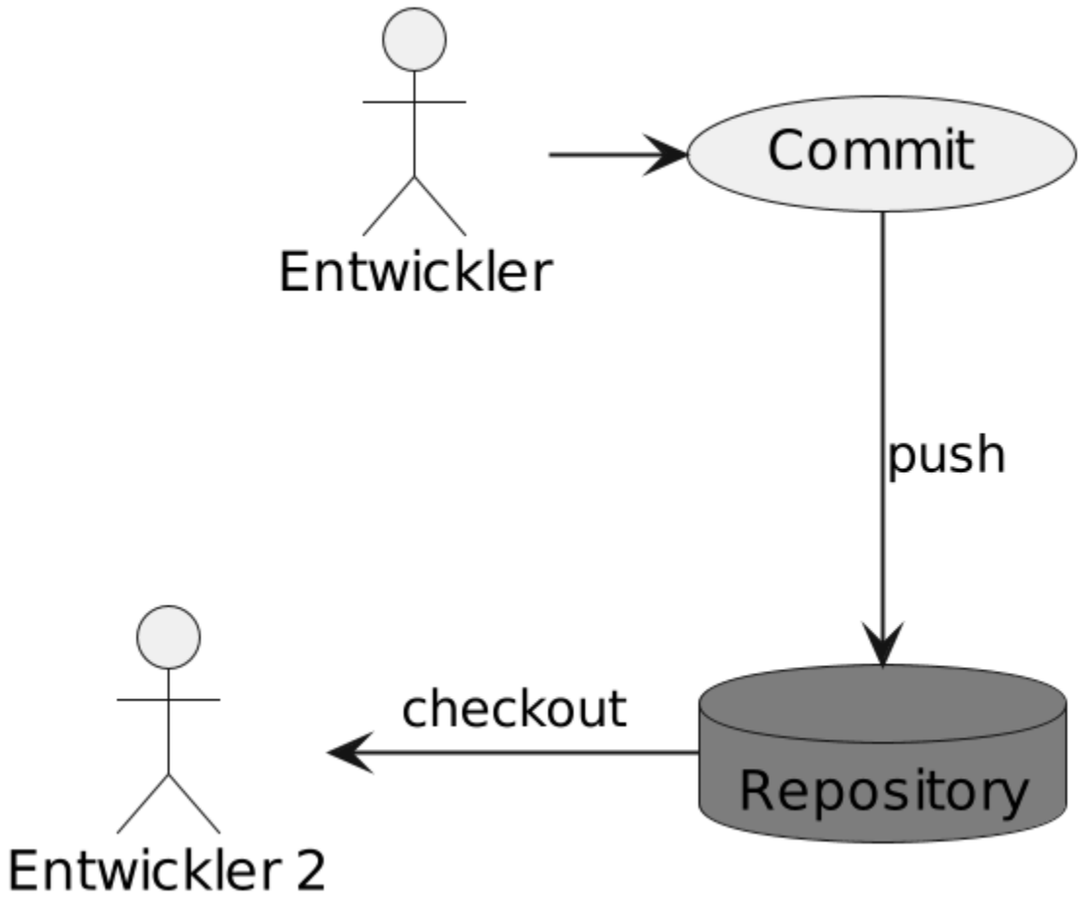
Entwicklungszyklus



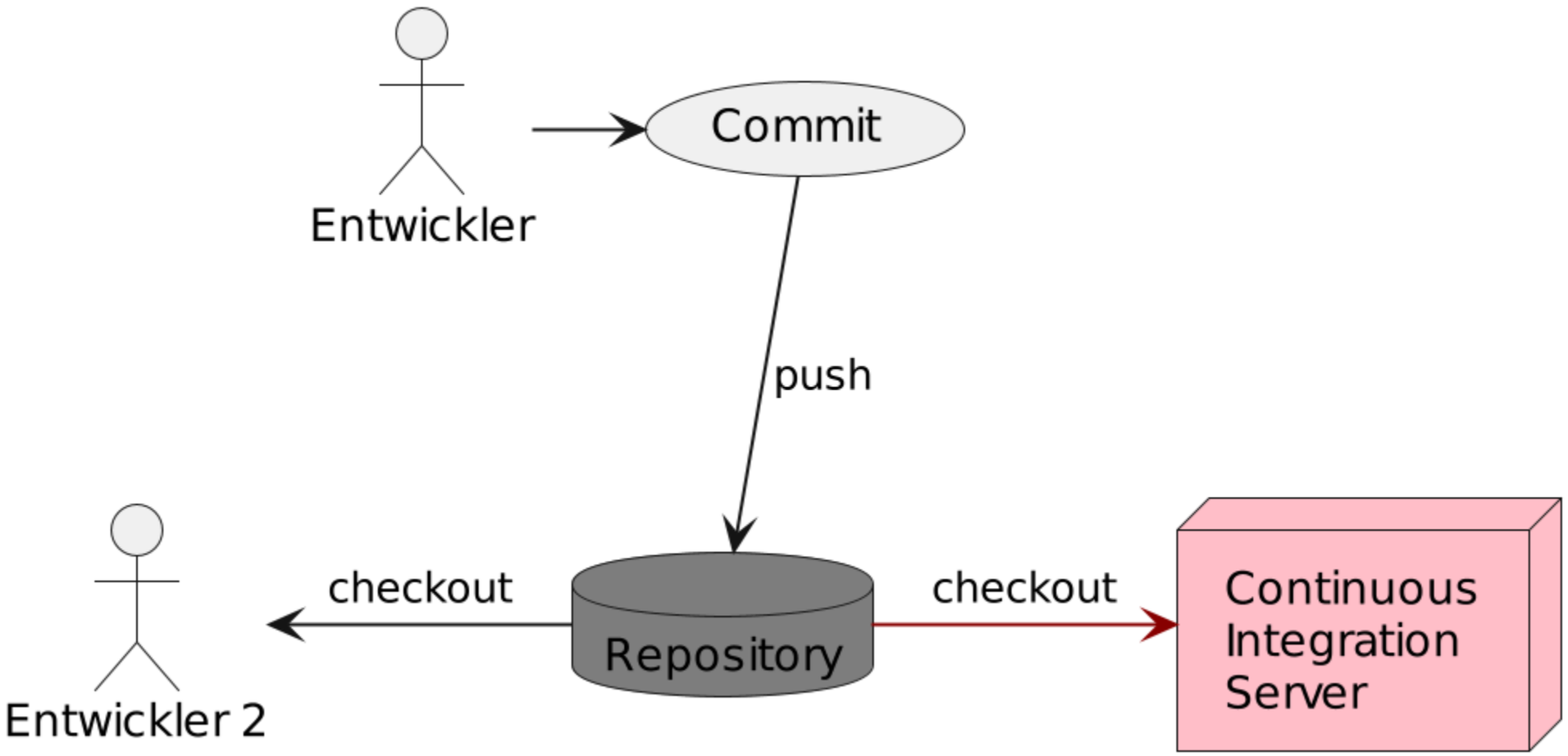
Entwicklungszyklus



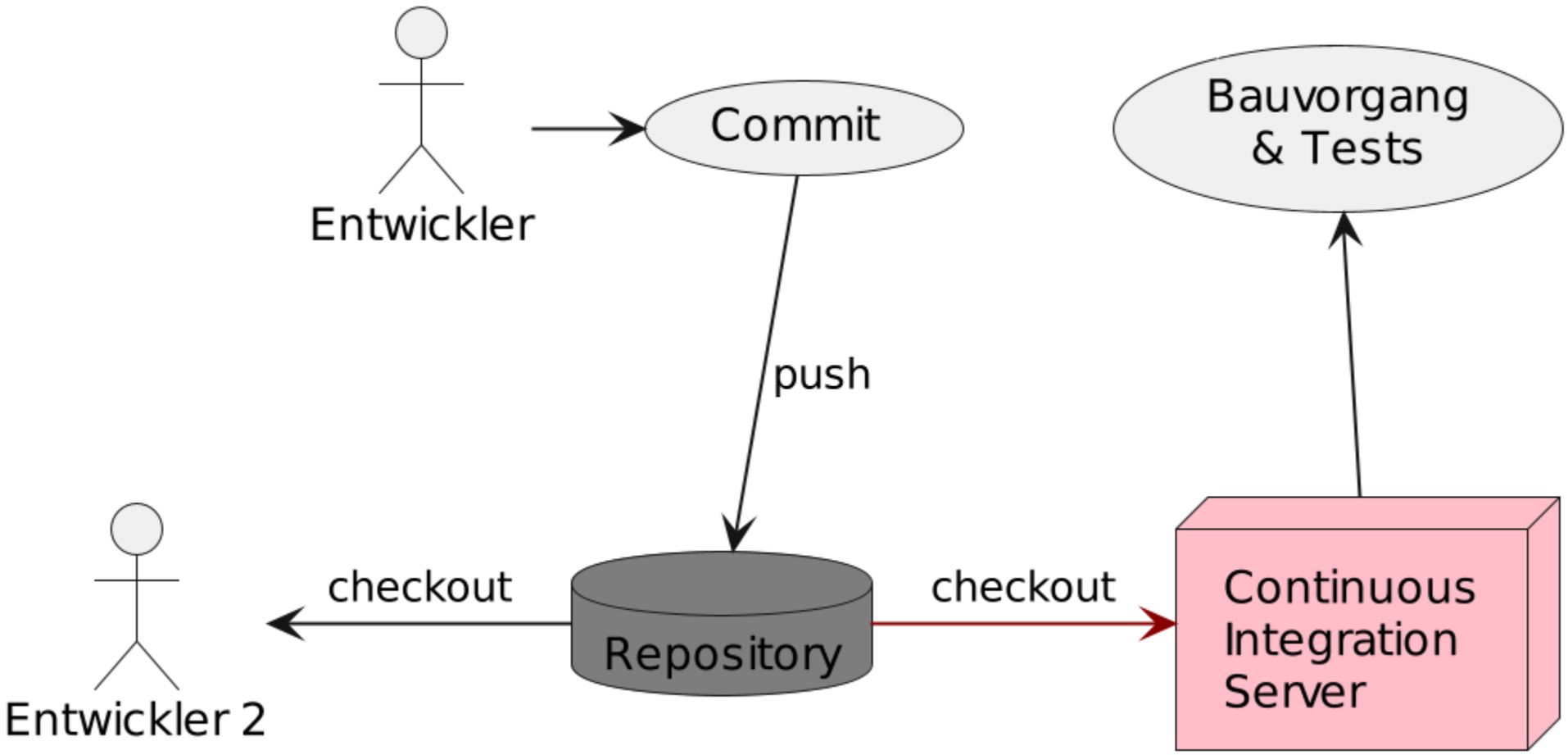
Entwicklungszyklus



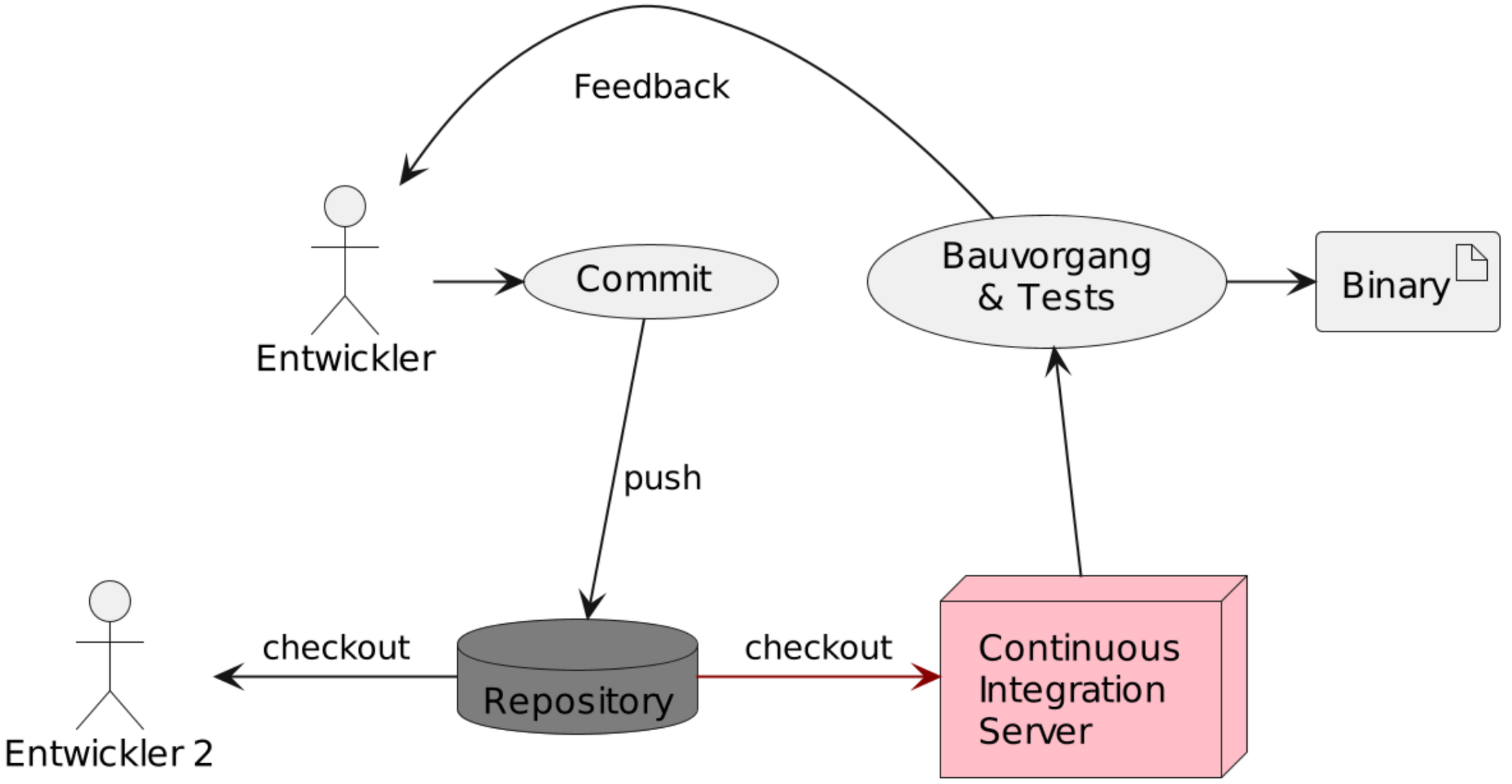
Entwicklungszyklus



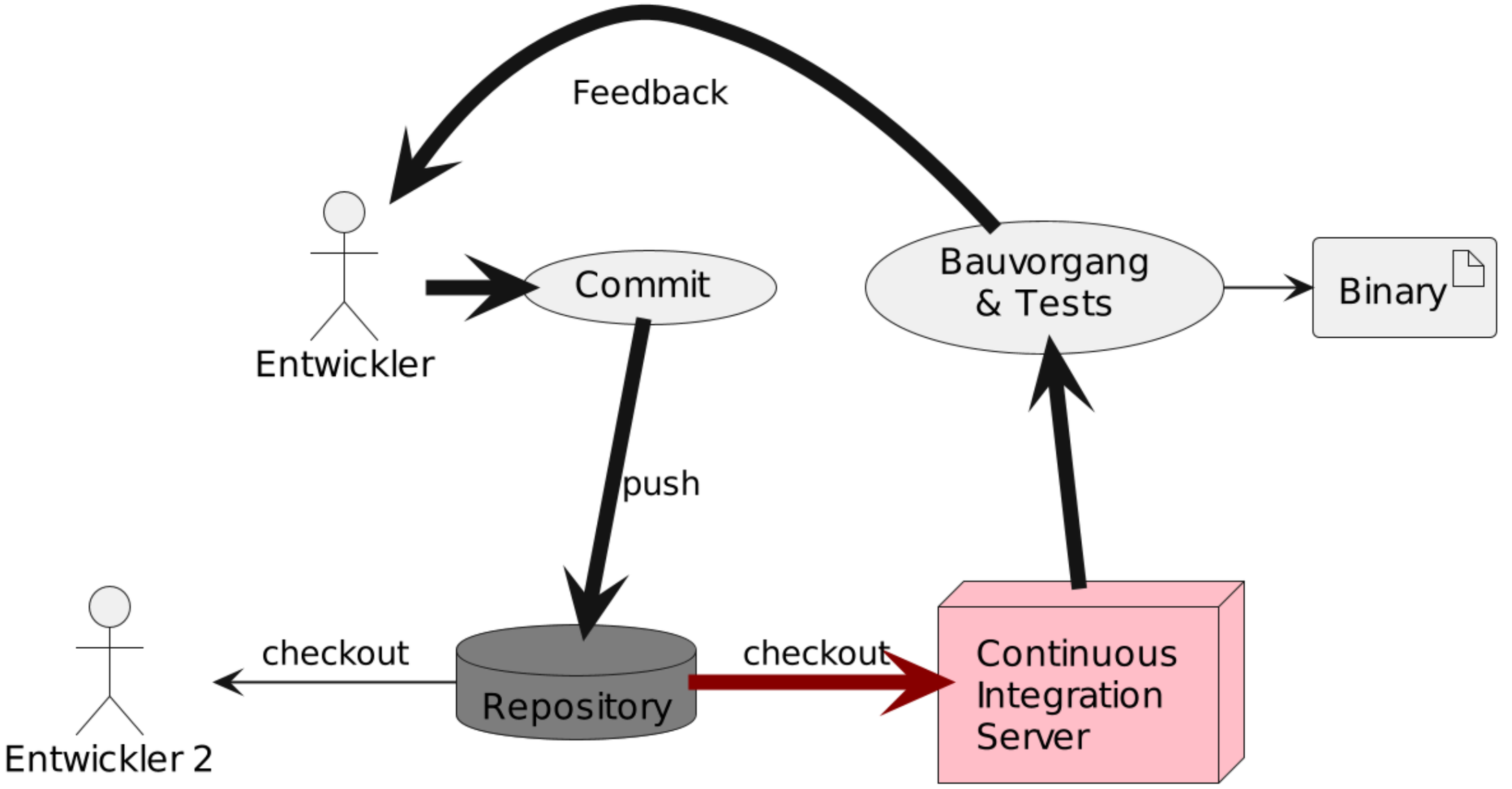
Entwicklungszyklus



Entwicklungszyklus



Feedback-Schleife



- Feedback-Schleife sollte möglichst kurz sein
  - nicht länger als 10 Minuten

CI: Voraussetzungen

- zentrales Repository
- automatisierter Bauprozess
- automatisierte Tests
- automatisierte Fehlerbehandlung
  - Fehler beim Bauen
  - Fehler in den Tests

CI: Prinzipien

- jeder committed täglich ins Repository
- jeder Commit löst einen Bauvorgang aus
- jeder kann das Ergebnis des (letzten) Bauvorgangs einsehen
- Tests möglichst nahe am Live-System
- möglichst einfacher Zugriff auf das Build-Artefakt (Binary)

CI: Nachteile

- (hoher) initialer Aufwand
- CI-Server wird benötigt (Kosten)
- gute Testabdeckung muss vorhanden sein

CI: Vorteile

- letzter funktionierender Stand ist bekannt und im Zugriff
- immer eine stabile Version verfügbar
  - z.B. für Tests, Demos, …
- zeitnahes Feedback an Entwickler bzgl. Qualität und Funktionalität
- Aufdecken von Integrationsproblemen

CI: Vorteile

- unvollständiger / inkompatibler Code wird direkt erkannt
- Erkennung von konfliktreichen Änderungen
- QA-Erweiterungen möglich
  - z.B. statische Code-Analyse

CI: Fazit

- grundlegender Bestandteil moderner Softwareentwicklung
- Erhöhung der Qualität eines Projekts bereits durch bloße Anwesenheit
  - Hawthorne-Effekt
- Projektverlauf wird sichtbar und nachvollziehbar

CD

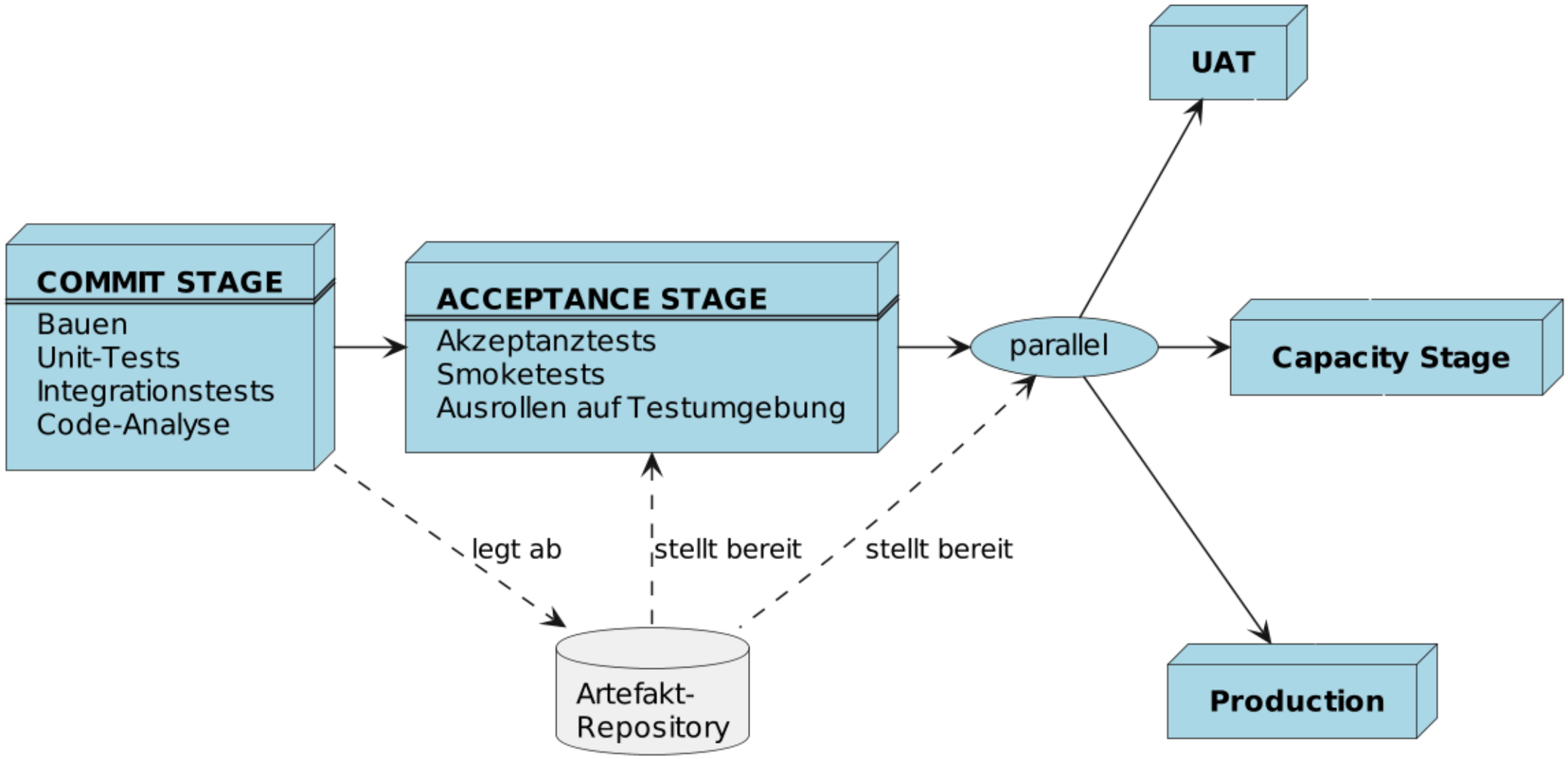
Continuous Delivery

- Software *könnte* jederzeit automatisch ausgerollt werden

Continuous Deployment

- jede Änderung, die erfolgreich alle Phasen durchläuft, *wird* automatisch ausgerollt

Deployment Pipeline



Deployment Pipeline

- Code durchläuft mehrere Phasen (Stages)
- jede Phase bildet ein *Quality Gate*
- mit jeder erfolgreichen Phasen steigt die Wahrscheinlichkeit, dass die Anwendung für die Produktion bereit ist

Commit Stage

- subsummiert CI
- Erzeugung einer eindeutigen Release-Nummer
- Erzeugung aller benötigten Artefakte für nachfolgende Phasen
  - paketierte Anwendung (jar, exe, …)
  - Dokumentation
  - Testdaten

Artefakt-Repository

- enthält alle in der Commit-Stage erzeugten Artefakte
- andere Phasen können über eindeutige ID auf Artefakte zugreifen
- sollte mit ins Backup aufgenommen werden
- sollte regelmäßige aufgeräumt werden

Acceptance-Stage

- Aufsetzen und Konfigurieren einer oder mehrere Zielumgebungen
- Ausrollen der Artefakte in den Zielumgebungen
- Ausführen von automatisierten Tests gegen Zielumgebungen
  - Smoketests
  - Akzeptanztests

Acceptance-Stage

- automatisierte Akzeptanztests können aufwendig in Erstellung und Wartung sein
- es kann sinnvoll sein auf automatisierte Akzeptanztests zu verzichten
  - manuelle Tests sollten dann zumindest dokumentiert und protokolliert werden
- ansonsten mindestens den *Happy Path* testen

Weitere Test-Stages

- bei Bedarf können beliebig viele weitere Test-Phasen folgen
- **UAT**: User Acceptance Test → Freigabe durch Auftraggeber
- **Capacity Testing**: Lasttests
- …

Deployment in Produktion

- wird manuell ausgelöst (per Klick auf gewünschtes Release)
- der Rest läuft automatisch ab
- Best Practices
  - automatisierter Rollback-Plan inkl. Backup
  - Blue-Green-Deployments
  - Canary-Releases
  - Feature Toggles
  - Parallel Code Paths

Blue-Green-Deployment

- zwei identische, getrennte Produktiv-Umgebungen (*blau* und *grün*)
- eine Umgebung ist produktiv, die andere das Staging-System
- Vorgehen
  - Deployment in *blaue* Umgebung
  - Umleitung des Verkehrs nach *blau*
  - *blau* ist die neue Live-Umgebung
  - *grün* wird die neue Staging-Umgebung
  - nächstes Mal: Deployment nach *grün*
  - …
- erlaubt schnelles Rollback, falls ein Deployment schief läuft

Canary-Releases

- ähnlich wie Blue-Green-Deployment
- beide Versionen sind gleichzeitig in Betrieb
- Schritt für Schritt werden mehr Nutzer auf die neue Version geleitet
- Last kann langsam erhöht werden
- schnelles Rollback möglich

Feature-Toggles

- Anwendung enthält Schalter (*Toggles*), um gezielt bestimmte Funktionalität zu aktivieren
- mögliche Lösung, wenn es schwierig ist, auf Feature-Branches zu arbeiten
  - z.B. aufgrund ständiger Merge-Konflikte oder organisatorischer Vorgaben
- mögliche Probleme
  - kann schnell unübersichtlich werden
  - alte Feature Toggles bleiben im Code

Parallel Code Paths

- alter und neuer Code werden parallel ausgeführt
- Ergebnisse werden verglichen
- bei Abweichung: Meldung / Logging
- wenn keine Abweichungen auftreten → Entfernung des alten Codes
- nützlich zum Prüfen neuer Funktionen / Refactorings

CD: Nachteile

- sehr hoher initialer Aufwand
- erhöhte Kosten im Betrieb (z.B. durch Bereitstellungen mehrere Testumgebungen)
- kann große Umstellung für Entwicklungsteams bedeuten

CD: Vorteile

- weniger Risiko durch häufige Deployments
- Vermeidung eines *Big Bang*-Releases
- schnelleres Feedback durch Nutzer
- mehr Vertrauen in die Anwendung
- schnelle, zuverlässige, wiederholbare Prozesse anstatt veralteter (oder keiner) Dokumentation
- forcierte Zusammenarbeit

CD: Fazit

- CD ist der Goldstandard für das Ausrollen
- Muster sind bestenfalls Vorlagen und müssen angepasst werden
- Phasen (Stages) können sich von Projekt zu Projekt unterscheiden