Advanced Software-Engineering

DHBW Stuttgart
11.11.2025

Janko Dietzsch

„Aktuelles“ – and they did it once more … again

KIMI

Introducing Kimi K2 Thinking

Today, we are introducing **Kimi K2 Thinking**, our best open-source thinking model.

Built as a **thinking agent**, it reasons step by step *while* using tools, achieving state-of-the-art performance on Humanity's Last Exam (HLE), BrowseComp, and other benchmarks, with major gains in reasoning, agentic search, coding, writing, and general capabilities.

Kimi K2 Thinking can execute up to **200 – 300 sequential tool calls** without human interference, reasoning coherently across hundreds of steps to solve complex problems.

It marks our latest efforts in **test-time scaling**, by scaling both thinking tokens and tool calling steps.

K2 Thinking is now live on kimi.com under the chat mode [1], with its full agentic mode available soon. It is also accessible through the Kimi K2 Thinking API.

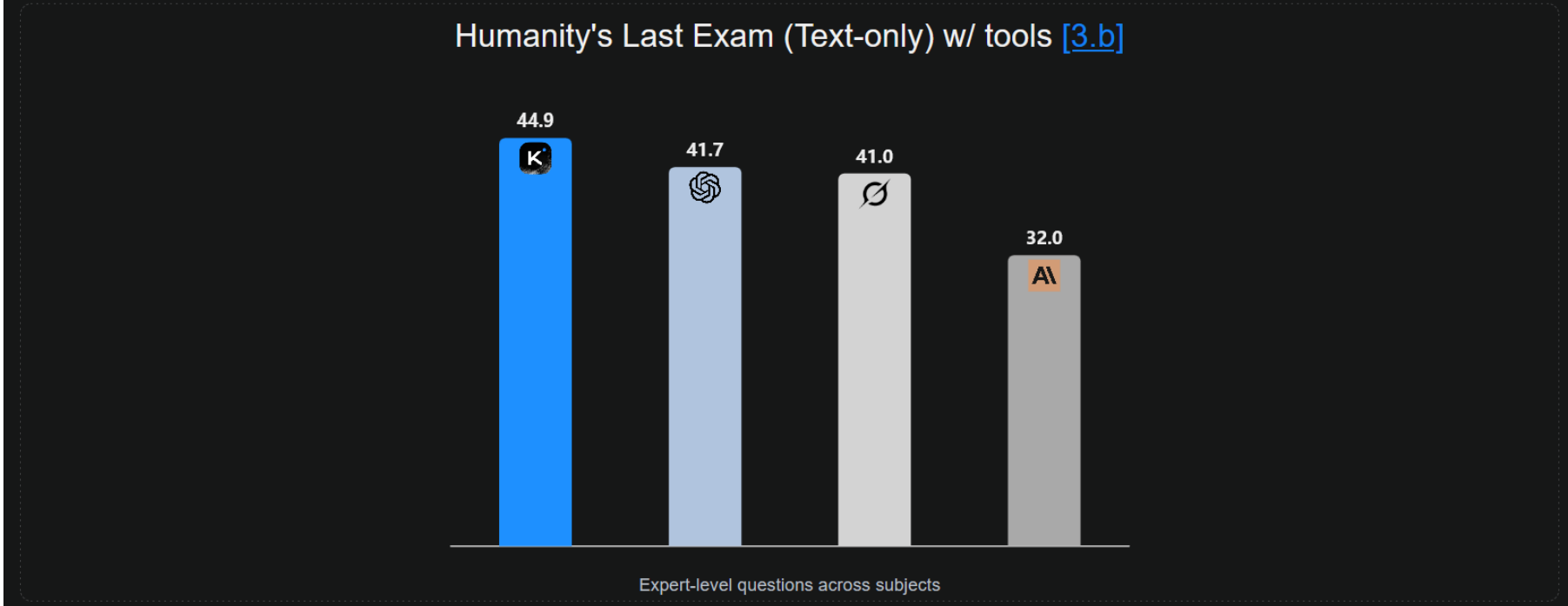
„Aktuelles“ – and they did it once more again ...

- Billionen Parameter-Modell mindestens auf Augenhöhe mit GPT-5 und Co.
- Angeblich nur 4.6 Mill. $ für das Training
- Hinter *Moonshot AI* stehen *Alibaba* und *Tencent*

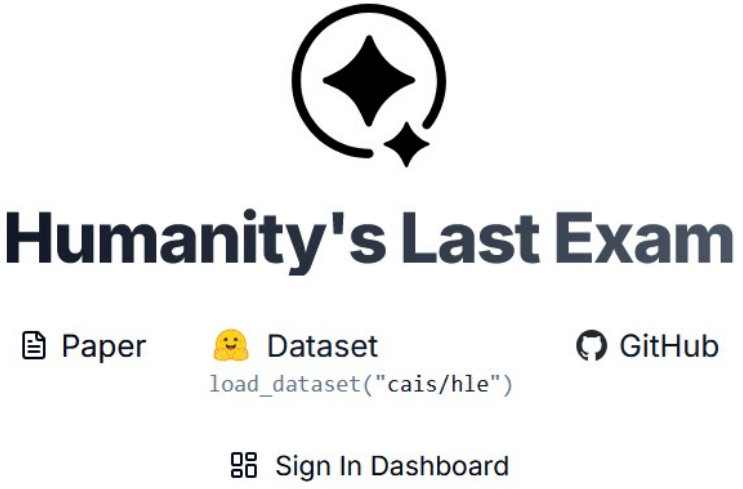
„Aktuelles“ – and they did it once more … again

Agentic Reasoning

K2 Thinking demonstrates outstanding reasoning and problem-solving abilities. On Humanity's Last Exam (HLE)—a rigorously crafted, closed-ended benchmark—spanning thousands of expert-level questions across more than 100 subjects, K2 Thinking achieved **a state-of-the-art score of 44.9%**, with search, python, and web-browsing tools, establishing new records in multi-domain expert-level reasoning performance.

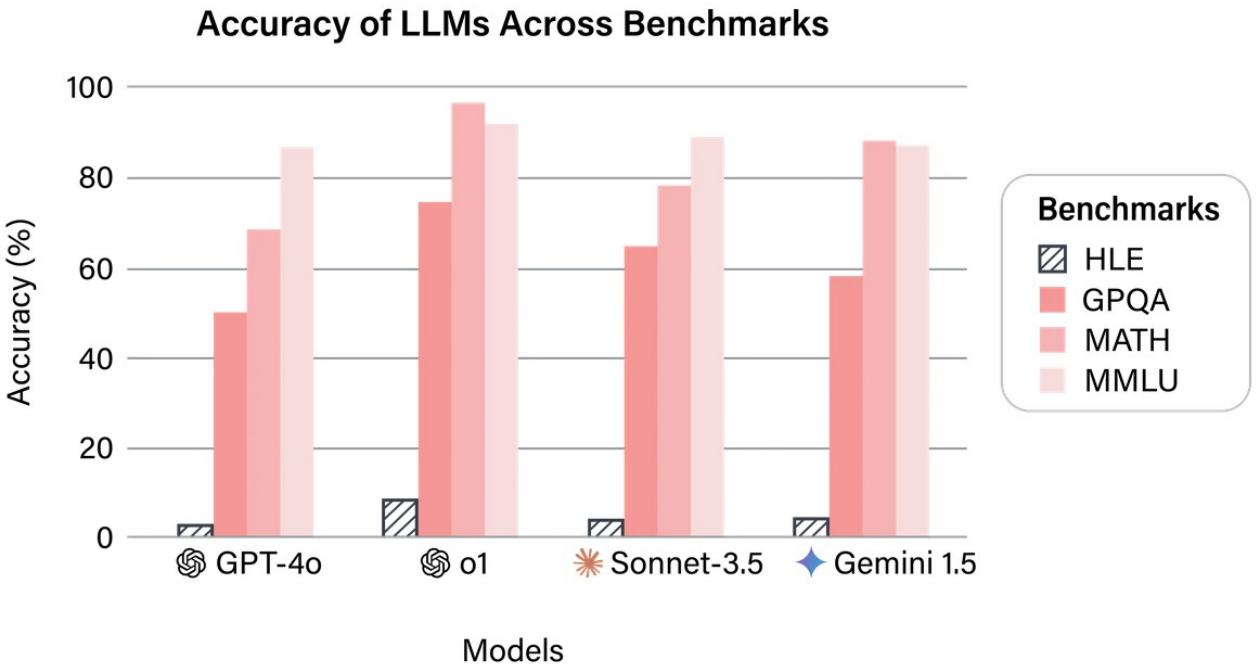


Humanity's Last Exam



Humanity's Last Exam

Benchmarks are important tools for tracking the rapid advancements in large language model (LLM) capabilities. However, benchmarks are not keeping pace in difficulty: LLMs now achieve over 90% accuracy on popular benchmarks like MMLU, limiting informed measurement of state-of-the-art LLM capabilities. In response, we introduce Humanity's Last Exam, a multi-modal benchmark at the frontier of human knowledge, designed to be the final closed-ended academic benchmark of its kind with broad subject coverage. The dataset consists of 2,500 challenging questions across over a hundred subjects. We publicly release these questions, while maintaining a private test set of held out questions to assess model overfitting.



Compared against the saturation of some existing benchmarks, Humanity's Last Exam accuracy remains low across several frontier models, demonstrating its effectiveness for measuring advanced, closed-ended, academic capabilities.

Humanity's Last Exam

Quantitative Results

Accuracy. All frontier models achieve low accuracy on Humanity's Last Exam, highlighting significant room for improvement in narrowing the gap between current LLMs and expert-level academic capabilities on closed-ended questions.

Calibration Error. Given low performance on Humanity's Last Exam, models should be calibrated, recognizing their uncertainty rather than confidently provide incorrect answers, indicative of confabulation/hallucination. To measure calibration, we prompt models to provide both an answer and their confidence from 0% to 100%.

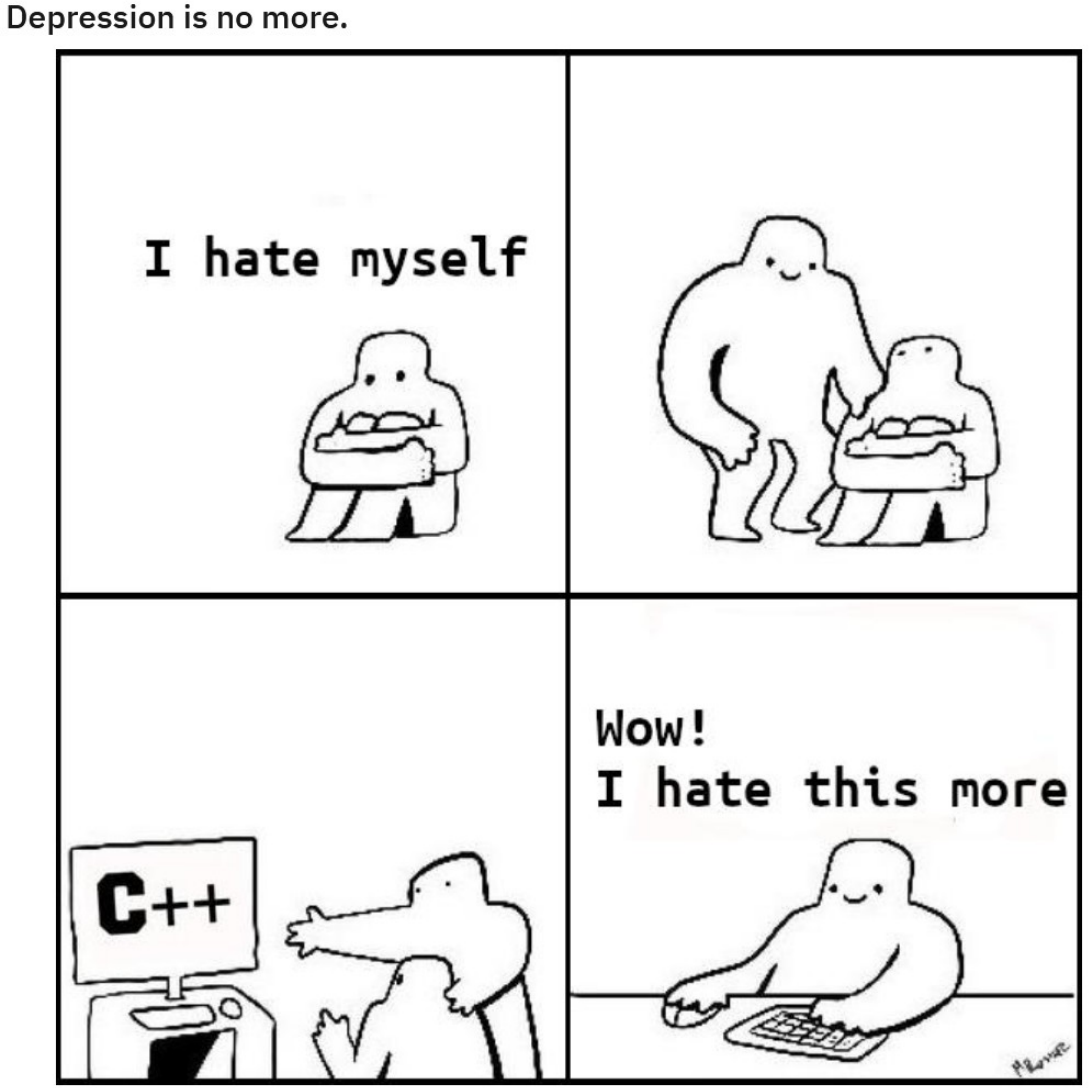
Judge Model: o3-mini | Dataset Updated: April 3rd, 2025

Model	Accuracy (%) ↑	Calibration Error (%) ↓
Grok 4	25.4	
GPT-5	25.3	50.0
Gemini 2.5 Pro	21.6	72.0
GPT-5-mini	19.4	65.0
DeepSeek-R1-0528*	14.0	78.0
Claude 4.5 Sonnet	13.7	65.0
Gemini 2.5 Flash	12.1	80.0
Claude 4.1 Opus	11.5	71.0
DeepSeek-R1*	8.5	73.0
o1	8.0	83.0
GPT-4o	2.7	89.0

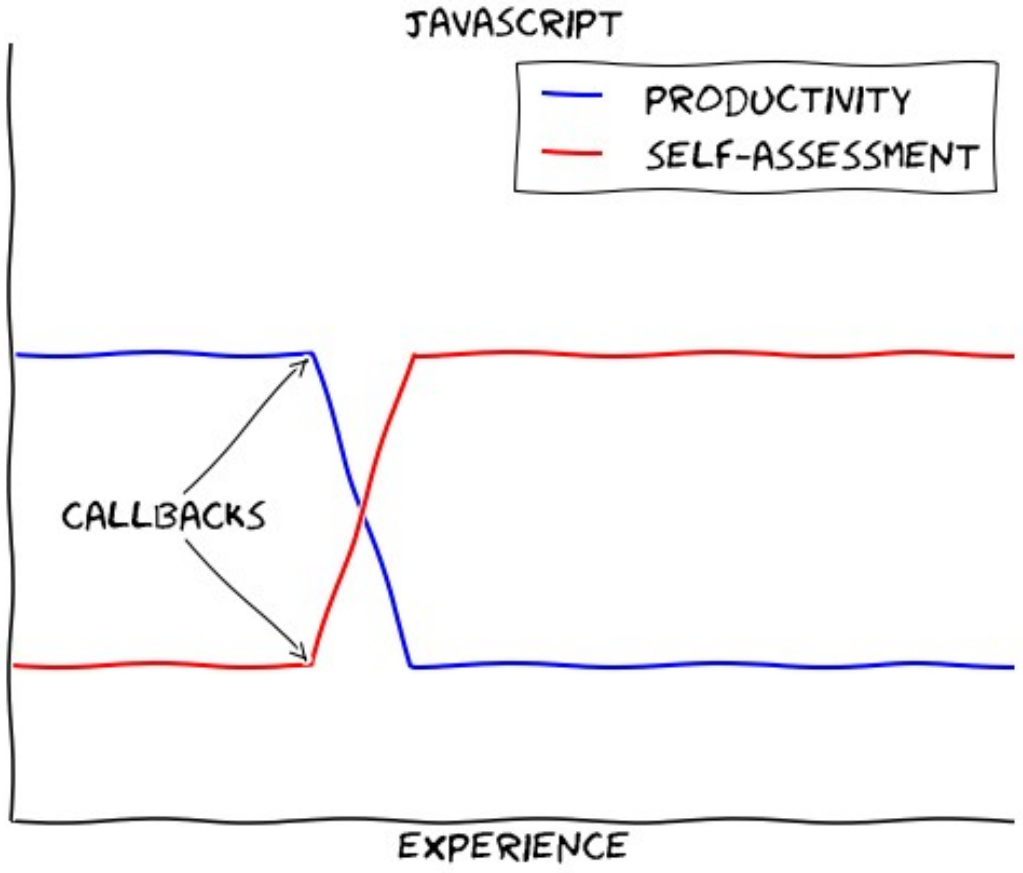
*Model is not multi-modal, evaluated on text-only subset.
Also available at SEAL LLM Leaderboards

„Aktuelles“ – … und das Verhältnis von Technologie und Sozialem … meistens

„Aktuelles“ – Aufbauen mit C++ mal anders

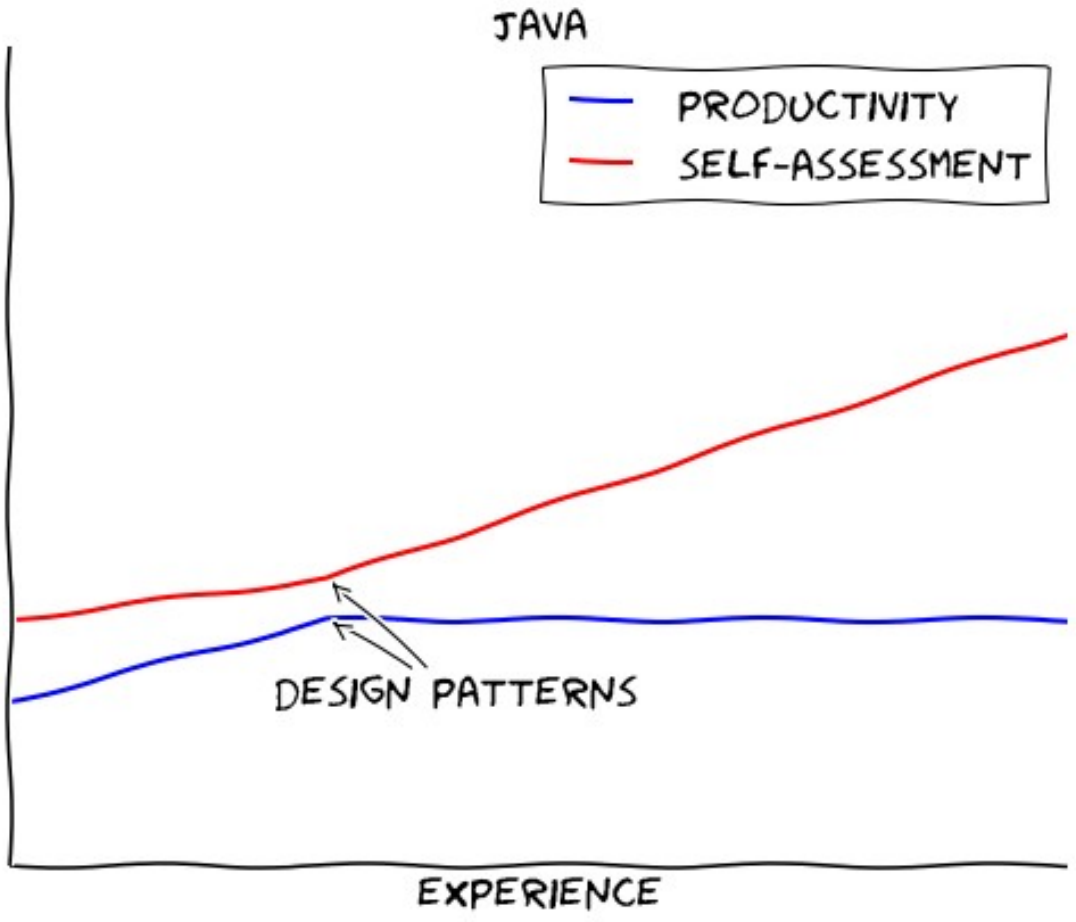


Learning Curves (for different programming languages)



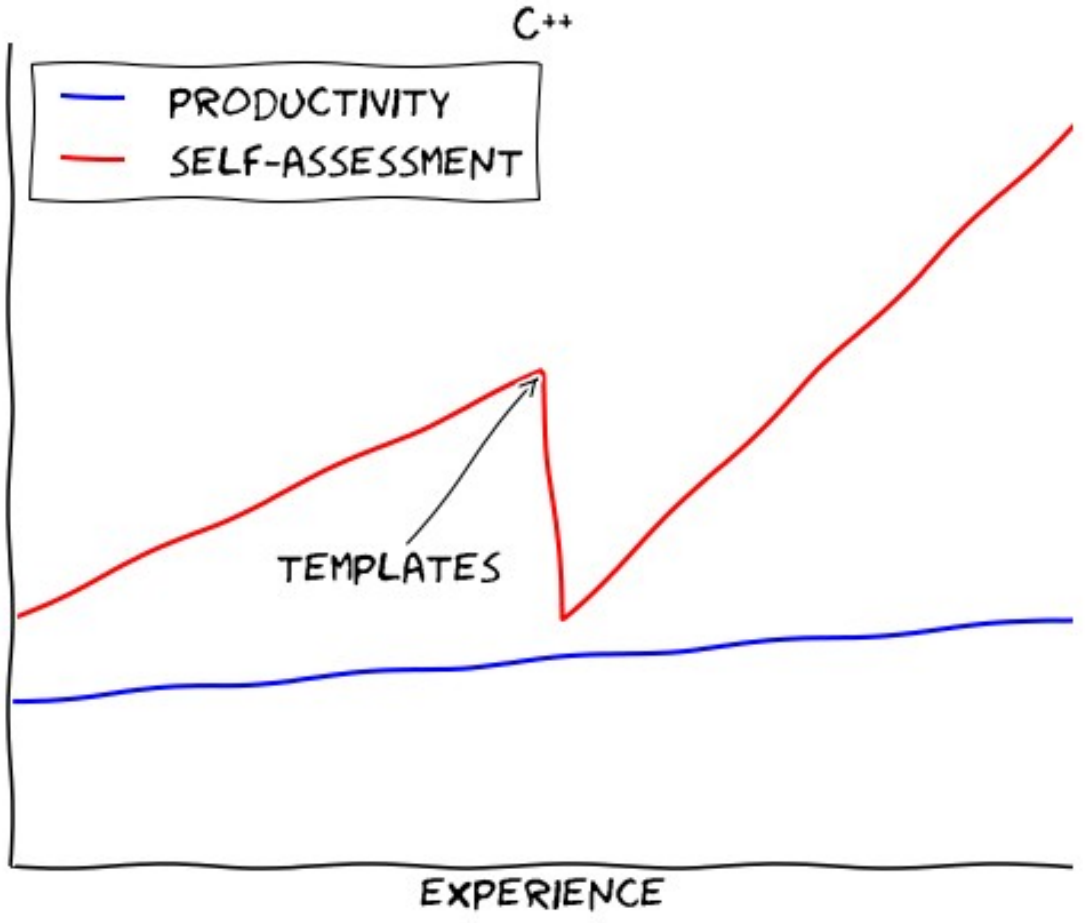
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



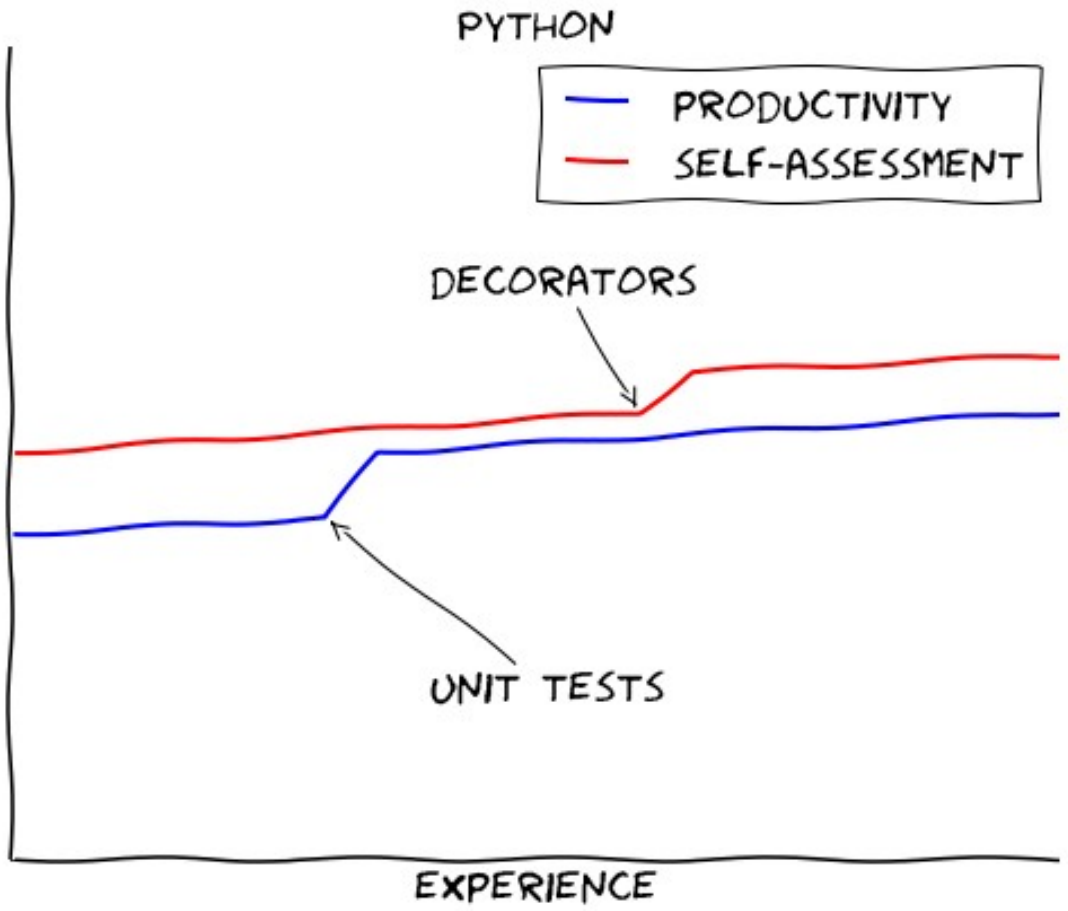
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



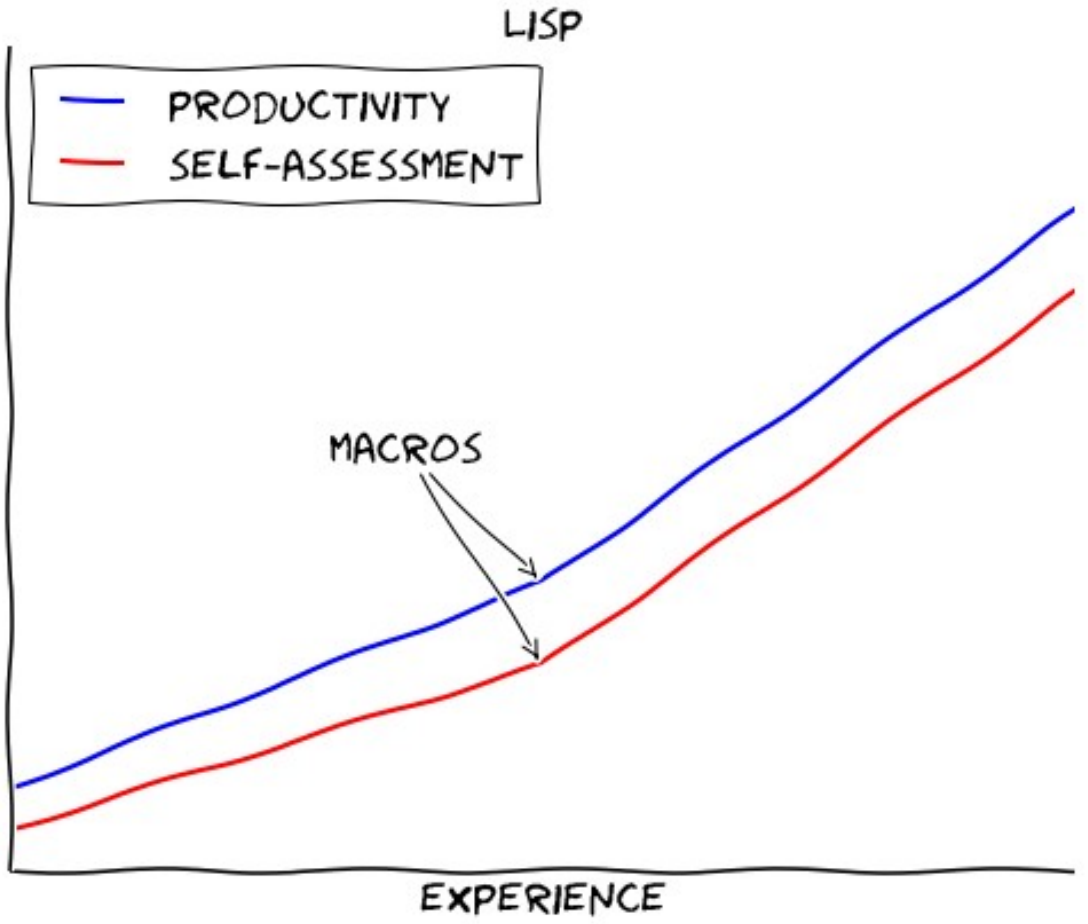
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



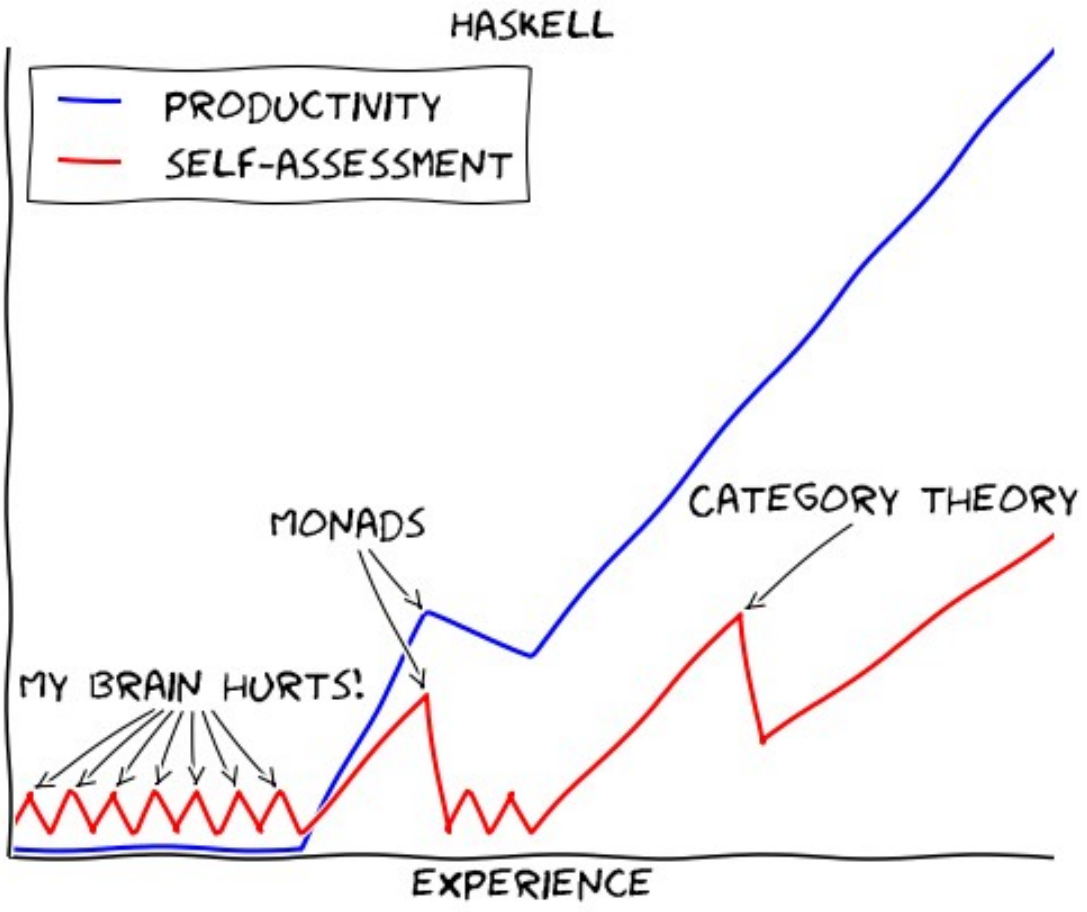
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



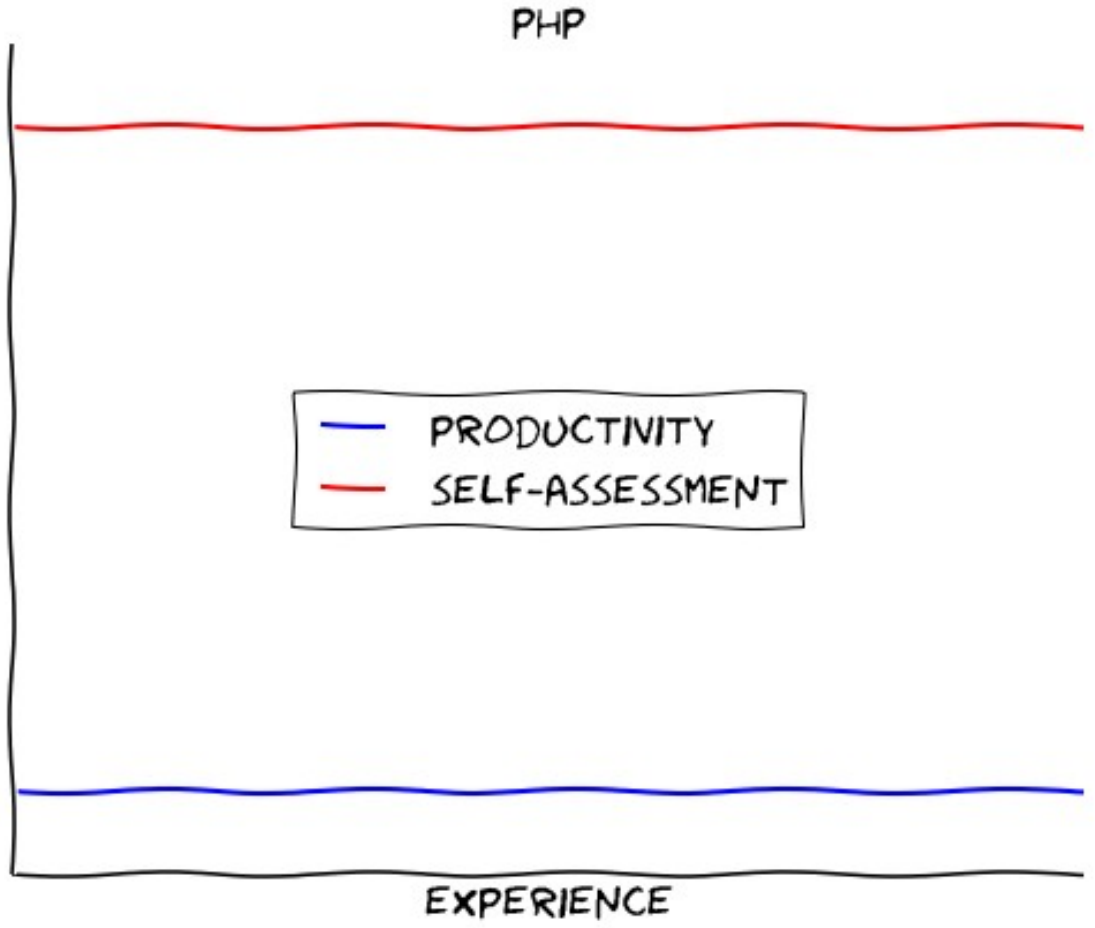
https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

Learning Curves (for different programming languages)



https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md

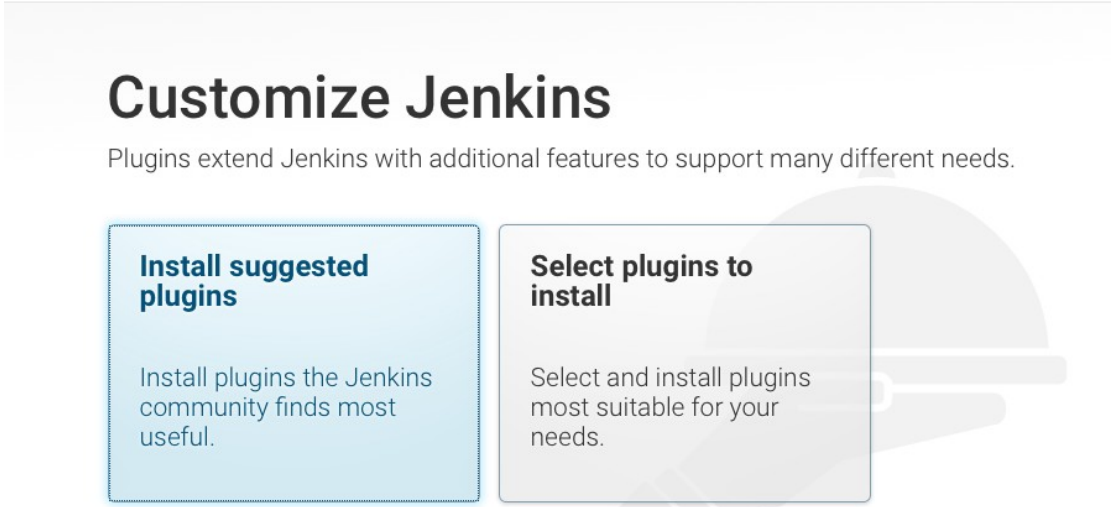
4.6.3 CI-Systeme/CI-Server

- Orchestrieren die ablaufenden Prozesse
- Beispiele:
  - Jenkins / Hudson
  - CruiseControl / CruiseControl.NET
  - TeamCity (JetBrains)
  - GitLab CI
  - Travis CI (gehostet)

Beispiel Jenkins

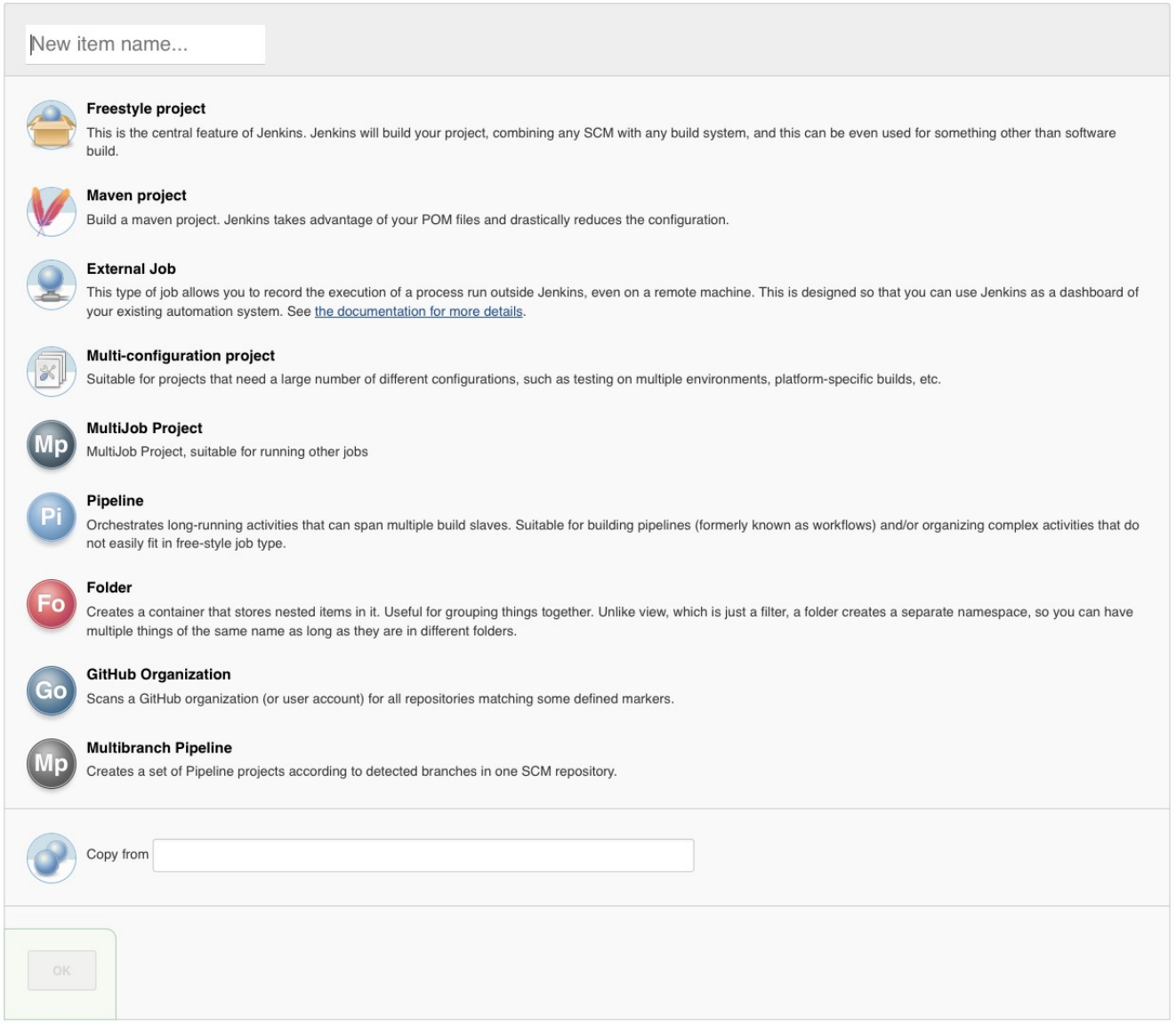
- Jenkins ist ein Java-basierter CI Server
- Erst am 27.04.2016 Version 2.0 nach 10 Jahren herausgegeben (Pipeline-Plugin zur Definition von Pipelines in Groovy)
- Wurde von Kohsuke Kawaguchi (jetzt CTO bei CloudBees) bei Sun unter dem Namen Hudson als Alternative zu CruiseControl entwickelt und fand schnell Verbreitung
- Nach der Übernahme von Sun durch Oracle kam es zu einem Bruch in der Hudson-Community und Jenkins entstand als Fork
- Jenkins entwickelte sich dynamischer und hängte Hudson ab, der mittlerweile zur Eclipse Foundation gewandert ist

Beispiel Jenkins



- Jenkins ist sehr weit verbreitet und kann durch eine sehr große Palette an Plugins äußerst flexibel angepaßt und erweitert werden:
  - März 2017 – 150110 Installationen, 1363 Plugins
  - Febr. 2019 – 228825 Installationen, 1728 Plugins
  - Okt. 2023 – 292425 Installationen, 2146 Plugins
  - Okt. 2024 – 282292 Installationen, 2191 Plugins
  - April 2025 – 280168 Installationen, 2245 Plugins
- Statistische Zahlen auf https://stats.jenkins.io/

Beispiel Jenkins



- Vielzahl von Projektvorlagen um die unterschiedlichsten Anwendungsfälle zu unterstützen

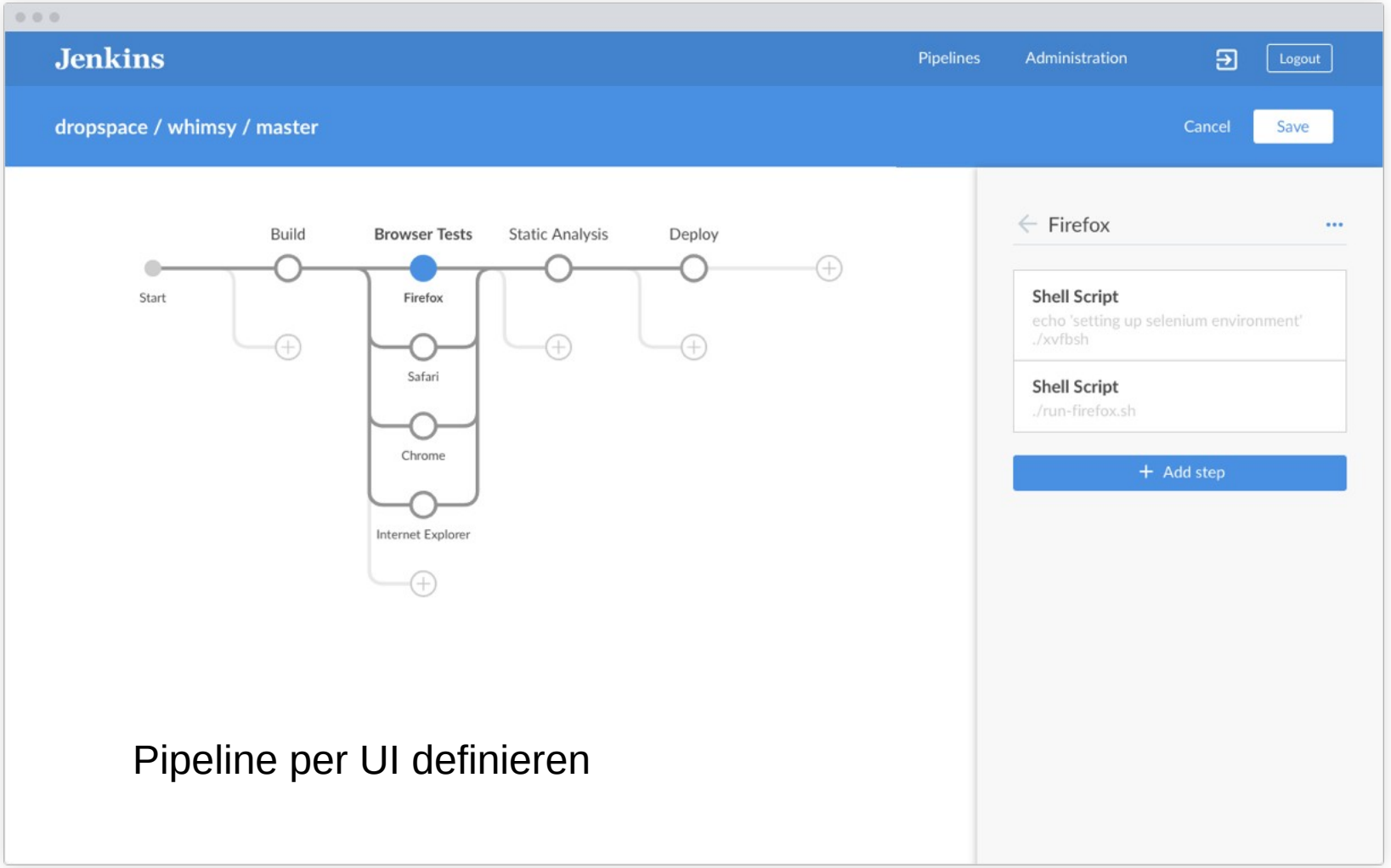
Beispiel Jenkins

Stage View

	Build the sudo images for installation	Basic Test: running from repo with preinstalled libs	Basic Test: running from setup.py install	Basic Test: pip develop mode install
Average stage times: (Average full run time: ~27y 220d)	19s	1min 50s	1min 40s	1min 29s
#34 Mar 03 18:56 — 81 commits	1min 34s master failed			
#33 Dec 22 13:00 — 3 commits	17s master	1min 54s master failed		
#32 Dec 22 12:33 — 20 commits	15s master	1min 35s master	1min 38s master	1min 29s master
#31 Dec 22 12:16 — No Changes	16s master	1min 40s master	1min 42s master	1min 29s master
#30 Dec 22 12:10 — No Changes	15s master	2min 11s master failed		

Alternatives UI/UX – *Blue Ocean*

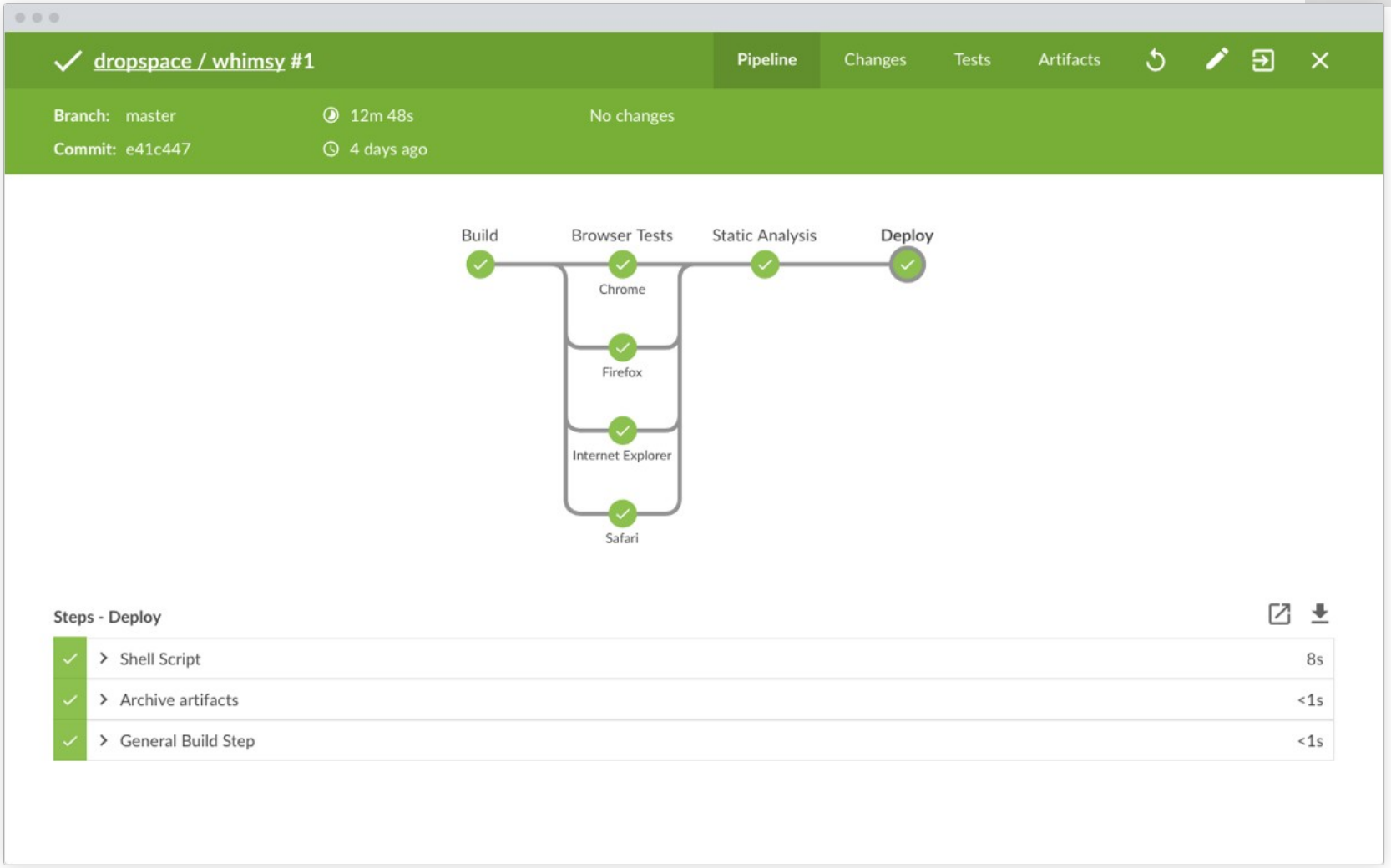
- Seit 2017 …



Pipeline per UI definieren

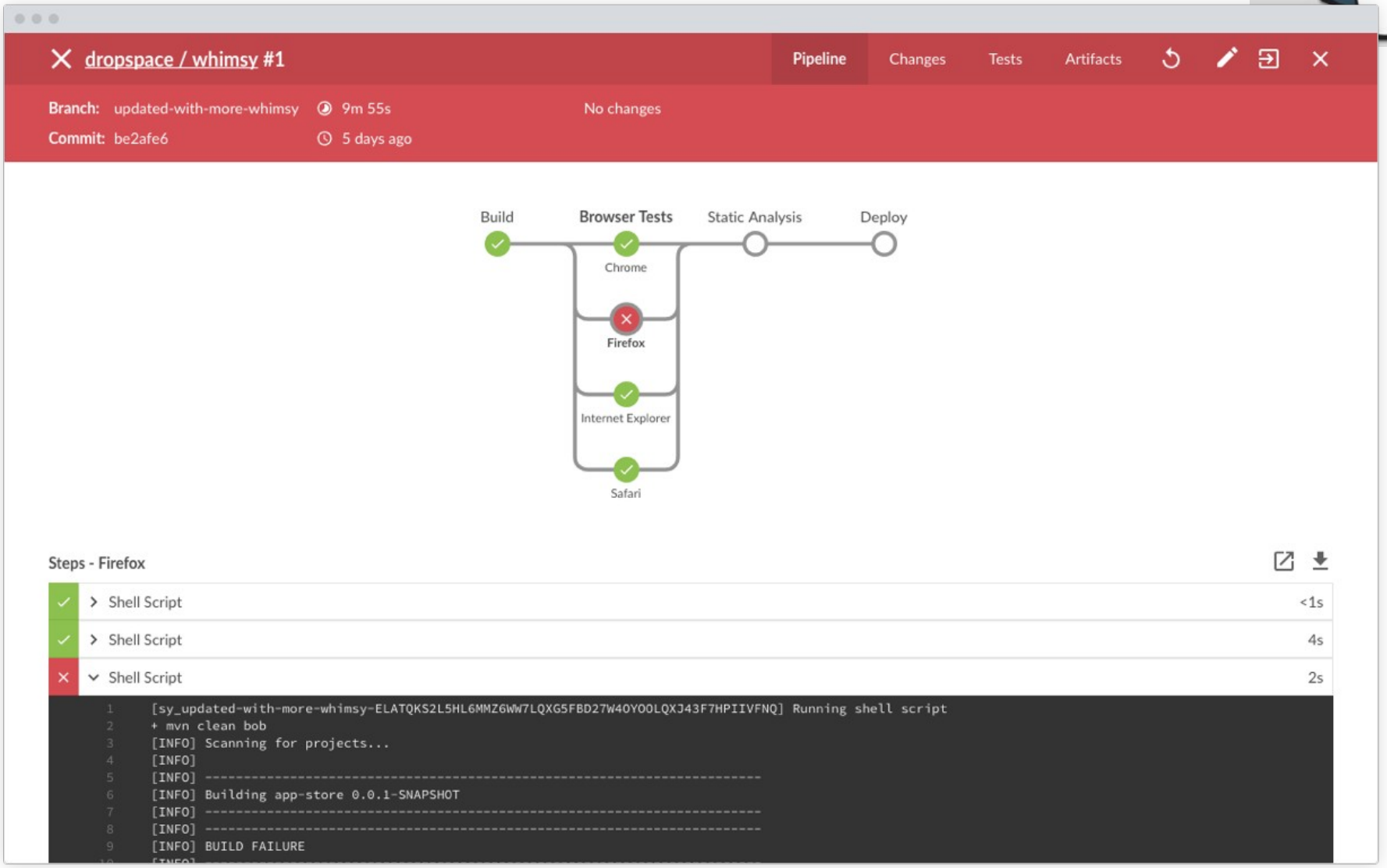
Alternatives UI/UX – *Blue Ocean*

Erfolgreicher Pipeline-Durchlauf:



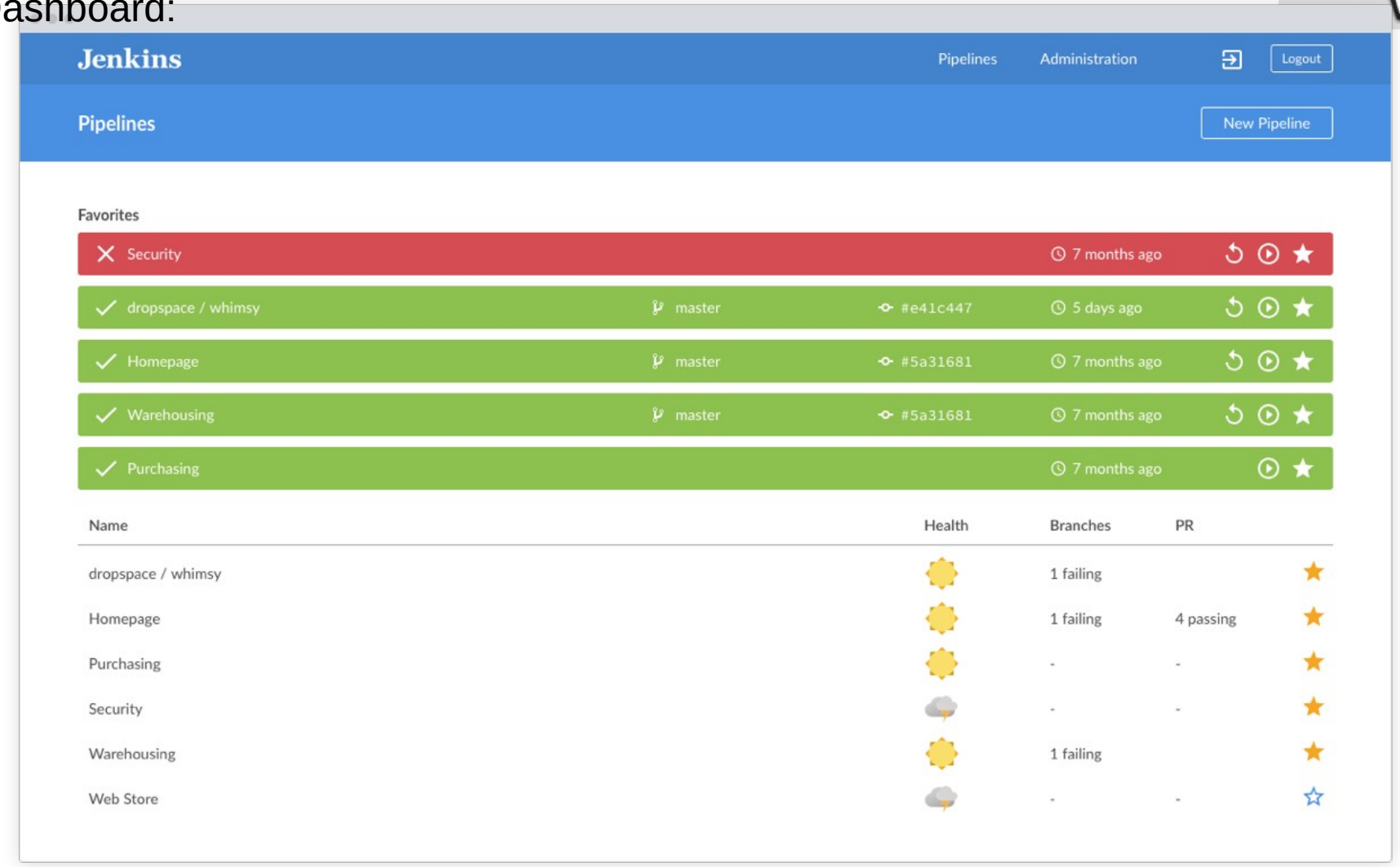
Alternatives UI/UX – *Blue Ocean*

Fehlerhafter Pipeline-Durchlauf:

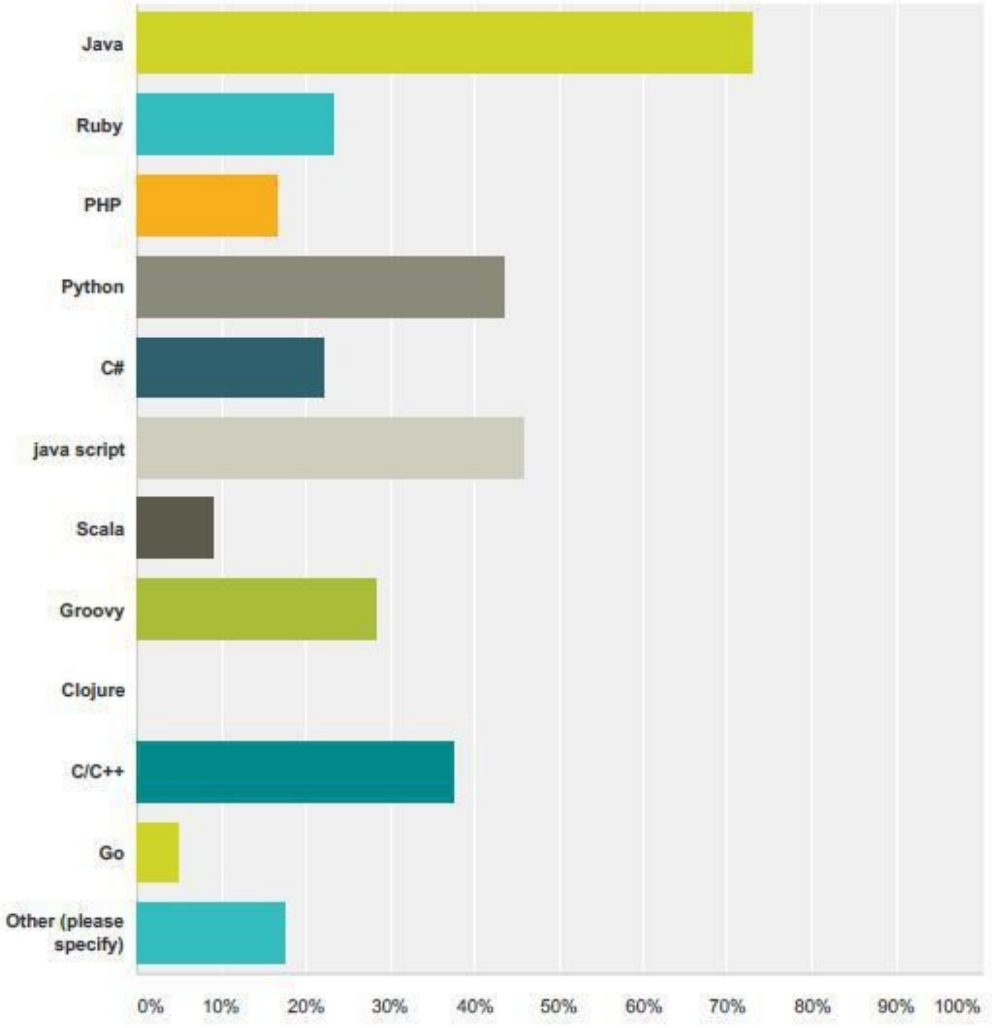


Alternatives UI/UX – *Blue Ocean*

Anpaßbares, personalisiertes Dashboard:

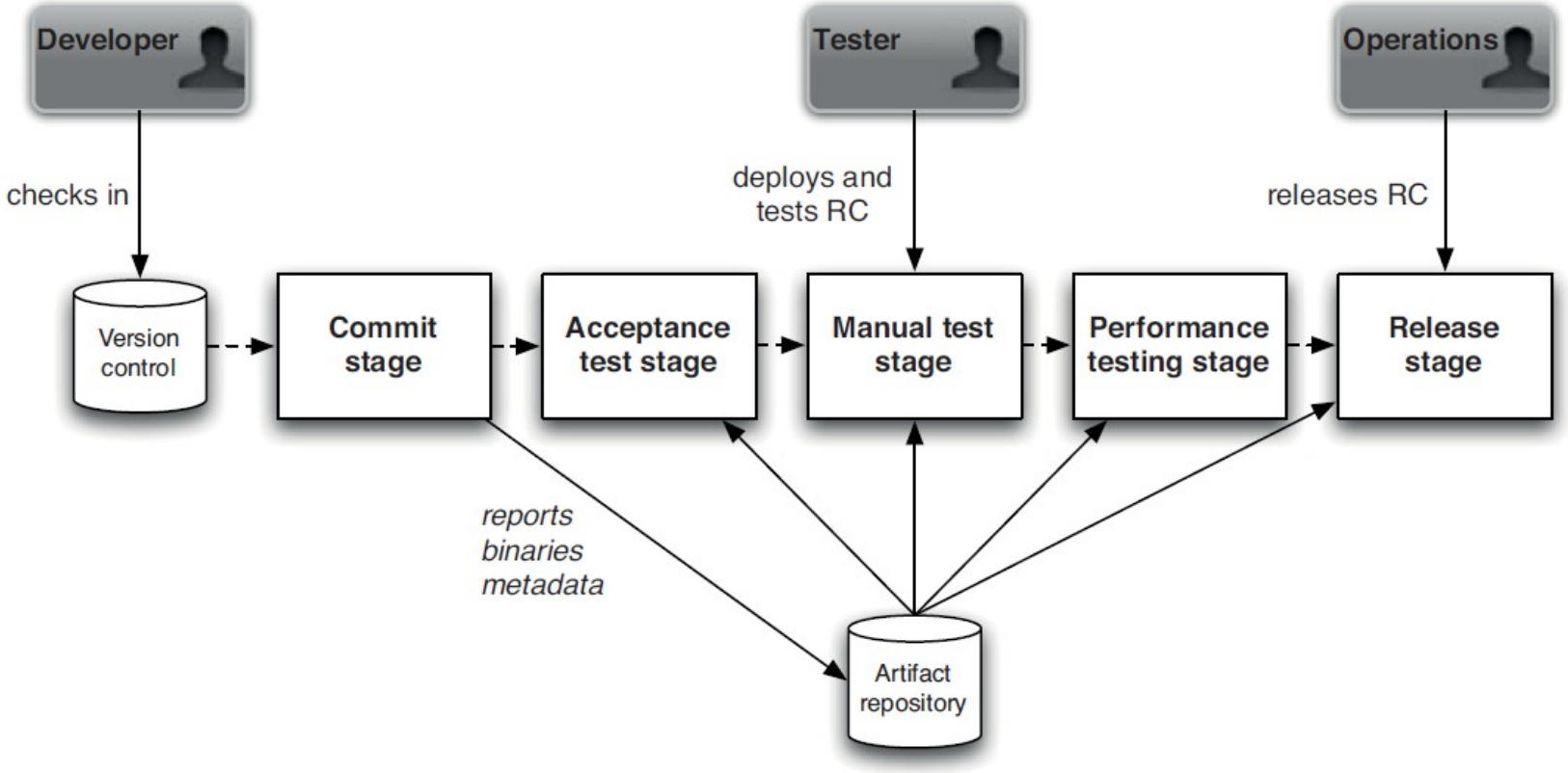


Beispiel Jenkins



- Hudson bzw. Jenkins kommt zwar aus dem Java-Umfeld, hat aber in allen technologischen Bereichen einen „guten Rückhalt“

4.6.4 Artifact Repository



- Speichert die Ergebnisse der Commit-Stage, also Binaries, Reports, Meta-Daten, eventuell auch Scripte, … etc.
- Kernkomponente moderner CI-Systeme

4.6.4 Artifact Repository

- Speichert nur einige Versionen (im Unterschied zum normalen VCS, das die gesamte Historie enthält)
  - z.B. können fehlerhafte Binaries und ihre Reports und Meta-Daten wieder gelöscht werden
- Stellt klaren Bezug zwischen Artefakten und Revision der Software im VCS her
- Reproduzierbarkeit → wenn die Artefakte einer VCS Revision gelöscht werden, sollten durch das Anstoßen der Pipeline mit exakt der gleichen Revision auch die gleichen Binaries entstehen

4.6.5 Verwalten der Komponenten und ihrer Abhängigkeiten

- Damit eine Anwendung lauffähig ist, müssen alle (binären) Komponenten, z.B. Dll's oder SO's in der richtigen, passenden Version (passendes API) vorhanden und im Zugriff sein → das ist das zentrale Probleme das CI, also kontinuierliche Integration, mithelfen soll zu lösen
- Was für Abhängigkeiten gibt es?
  - Zum Beispiel zum Betriebssystem, Java → JVM, .NET → CLR, Rails → Ruby, C → C-Standard-Lib, …
  - Abhängigkeiten zur Build-Zeit (Komponenten im engeren Sinne → intern kontrolliert) und zur Laufzeit (Libraries → extern kontrolliert)

Abhängigkeitsarten

- Build-Zeit-Abhängigkeiten:
  - Müssen während der Kompilations- und eventuell Linkphase vorhanden sein
- Laufzeit-Abhängigkeiten:
  - Müssen während der Ausführung vorhanden sein, damit die Anwendung korrekt funktioniert
- Mitunter sind bzw. können die Versionen, die zur Build-Zeit genutzt werden, verschiedenen von jenen sein, die zur Laufzeit benutzt werden:
  - z.B. in C/C++ zur Build-Zeit sind nur die Header-Dateien nötig, zur Laufzeit die binären Dll's oder SO's
- Die Verwaltung von Abhängigkeiten kann sehr schwierig werden → ***„Dependency Hell“***

Dependency Hell

- Auch manchmal als „Dll Hell“ bezeichnet
  - Übliches Problem auf alten Windows-Systemen, bei denen alle Dll's in windows\system32 ohne Versionierung abgelegt wurden → Überschreiben älterer Versionen
  - In .NET etwas entschärft durch Assemblies mit Versionierung (kryptographisch signiert), Global Assembly Cache (GAC), der verschieden Assemblies mit dem gleichen Filenamen enthalten kann
  - Linux hängt an jedes SO-File eine ganzzahlige Nummer

Verwalten der binären Abhängigkeiten

- Eine Variante ist das Einchecken in das VCS des Source-Code's:
  - Auch separates VCS „mit Verweis“ möglich, z.B. Submodul-Mechanismus von Git
  - Einfachste Variante
  - Für kleine Projekte möglich
  - Eventuell 3 Verzeichnisse für die Build-, Test- und Laufzeitabhängigkeiten
  - Mit Versionsnummer ablegen, z.B. nunit-3.0.1.dll statt nunit.dll
  - Vorteil:
    - Exakte Zuordnung zu Revisionsständen im VCS
  - Nachteil:
    - Binäre Blobs im VCS
    - Sorgt schnell für große Datenmenge im VCS

Verwalten der binären Abhängigkeiten

- Eine Variante ist das Einchecken in das VCS des Source-Code's (cont.):
  - Eventuell ganze Toolchain im VCS → aber dann auf jeden Fall ein anderes VCS als für den Source-Code verwenden
  - Zu beachten, manche Plattformen können mehrere Versionen der gleichen Dll vertragen und manche nicht → wenn unterschiedliche Komponenten von verschiedenen Versionen einer Dll abhängen
  - Problem der transitiven Abhängigkeiten
    - kann lange dauern den Abhängigkeitsgraphen zu traversieren
    - Eventuell widersprüchliche Abhängigkeiten

Komponenten der Applikation

- Abhängigkeiten, die komplett unter der eigenen Kontrolle sind
- Wann sollte eine Code-Basis in Komponenten aufgeteilt werden? … schwierig exakt eine Definition dafür zu geben
- Applikation startet meistens als eine monolithische Einheit → evolviert über die Zeit in eine modularisierte Anwendung
- Kompromiß zwischen „verschiedenen Komplexitäten“:
  - Zu konstruierende Komponenten müssen eine „Mindestkomplexität“ haben, sonst zu trivial und
  - Die Applikation an sich muß so komplex geworden sein, dass sie die weitere Entwicklung massiv behindert
- Definition J. Humble:
  - “A component is reusable, replaceable with something else that implements the same API, independently deployable, and encapsulates some coherent set of behaviors and responsibilities of the system.“

Komponenten der Applikation

- Diese Definition würde zwar prinzipiell auch schon auf Klassen zutreffen, aber Overhead zu groß → Packaging → Abhängigkeiten auflösen … etc. ↔ Microservices !?
- D.h. sie bieten ein API (Interface) und die Implementierung, kann über verschiedenste Mechanismen erfolgen – statisches, dynamisches Binden, Web-Service … etc.
- Was spiegeln Komponenten wieder:
  - Zerlegen ein Problem in mehrere, kleinere ausdrucksstarke Teile
  - Teile haben eine unterschiedliche „Änderungsrate“ bzw. einen unterschiedlichen Lebenszyklus
  - Legen klare Grenzen für Verantwortlichkeiten bzw. den Kontext fest
    - Begrenzt Auswirkungen von Änderungen
    - Erleichtert Verständnis
  - Zusätzliche Freiheitsgrade für Optimierungen (Build, Test, Deploy)

Komponenten der Applikation

- Grad der Komplexität wird durch das Ausmaß der Koppelung sowohl des Interfaces, wie auch des Verhaltens bestimmt
- Wann sind Komponenten unbedingt einzuführen:
  - Wenn Teile unterschiedliche Zyklen haben müssen, z.B. Server und Client
  - Flexible Plattform → Plugin-System, (Micro)Kernel-Systeme
  - Das System besitzt Schnittstellen zu anderen Systemen
  - Es dauert zu lange das Projekt zu bauen oder auch nur zu bearbeiten
  - Die Code-Basis ist so groß, dass sie nicht mehr innerhalb eines Teams bearbeitet werden kann

Wie entwickelt man die Komponenten am besten?

- Generelle Empfehlung ist, nicht „entlang der Komponentengrenzen“, d.h. nicht eine Komponente pro Team
  - Tendenz zur „Silo-Bildung“ - lokales Optimieren und das „Große und Ganze“ wird aus dem Blick verloren
- Besser: tatsächliche End-To-End-Entwicklung, die cross-funktional ist
  - Team entwickelt eine Anforderung aus der Anwendungsdomäne und verändert bzw. bearbeitet dabei jegliche Komponenten, die dafür notwendig ist
  - Entwickler auch zwischen den Teams rotieren lassen
- Nebeneffekt → Kenntnis der ganzen Code-Basis ist verteilt
  - Zusätzliche Redundanz („Single Point of Failure“ vermeiden)
  - Funktionalität wird nicht mehrfach (verschieden) implementiert -> „DRY“ (Don't Repeat Yourself)

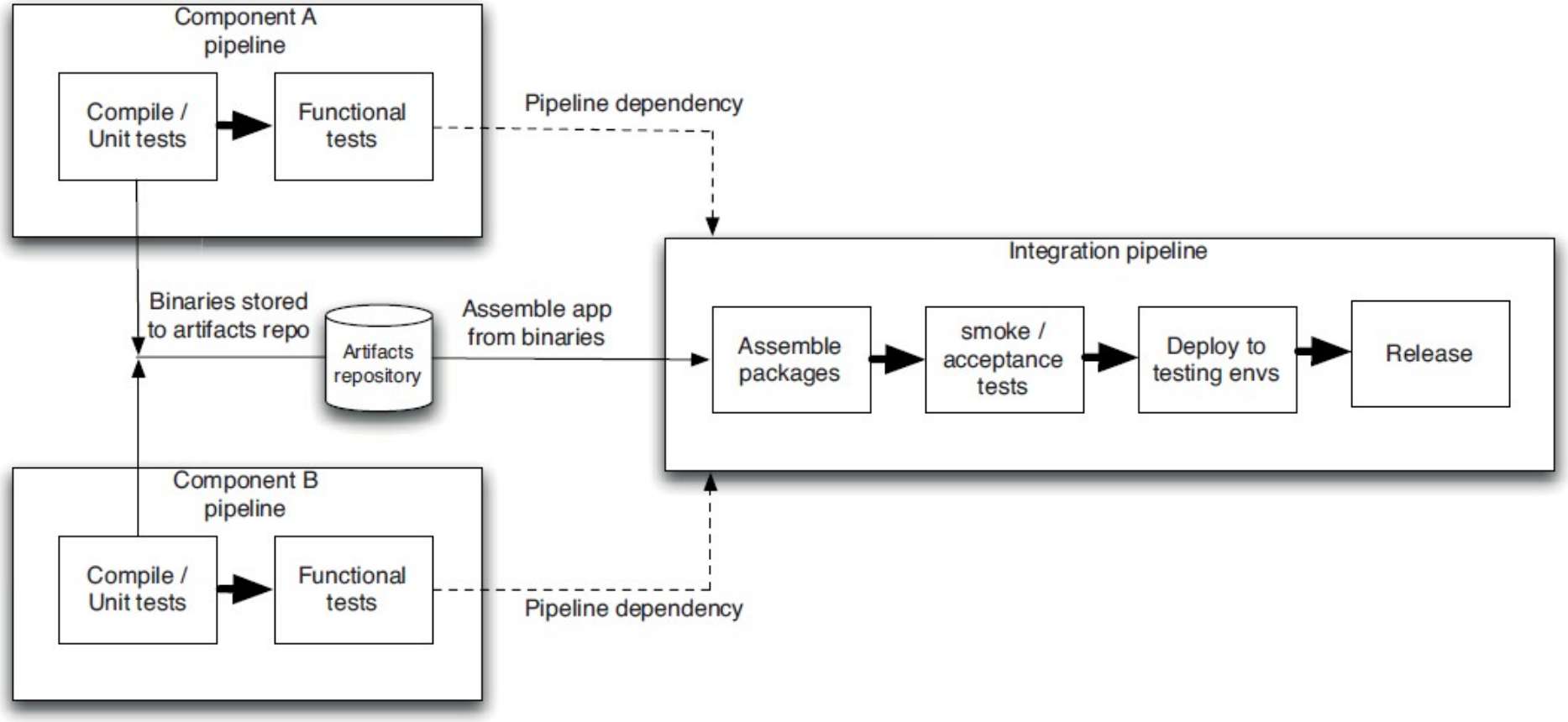
Wie entwickelt man die Komponenten am besten?

- Komponenten arbeiten von Anfang an besser zusammen, da der Blick des Entwicklers über den „Komponenten-Rand“ hinausgeht
- Conway's Law (Melvin Edward Conway):
  - „Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.“
- Am Ende ist die Zerlegung ein zu findender Kompromiß zwischen dem einen großen Monolithen und sehr vielen kleinen Komponenten → Software-Design ist eben auch ein Stück weit Kunst bzw. Handwerk

Komponenten und die Deployment-Pipeline

- Komponenten kann man auch mit einer einzelnen Pipeline behandeln
- Falls aber mehrere Pipelines nötig werden, z.B.:
  - Unterschiedlicher Lebenszyklus
  - Unterschiedliche „Funktionsbereiche“ der Anwendung werden eventuell von verteilten Teams entwickelt
  - Komponenten benutzen unterschiedliche Technologien
  - Komponenten werden von mehreren Teilen der Anwendung genutzt bzw. benutzt (Shared Components)
  - …
- Eine eigene „kleine DP“ für die Komponenten und eine Integration-Pipeline

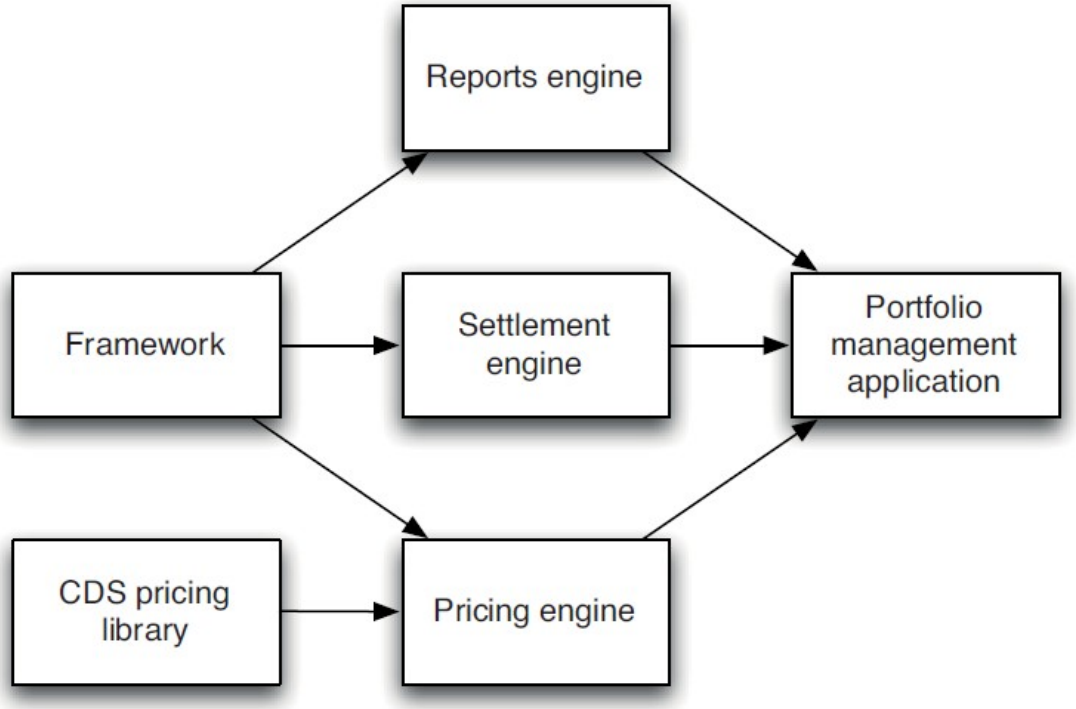
Die Integration-Pipeline



Die Integrationspipeline

- Auch hier wieder die zwei wichtigsten Aspekte – Sichtbarkeit und Feedback
- Möglichst schnell vom Ablauf einer Pipeline die abhängigen triggern
- Rückverfolgbarkeit welche Revision welcher Komponente in einen erfolgreichen oder fehlerhaften Durchlauf mündete
- Zentrale Rolle des Artefakt-Repository‘s

Der Abhängigkeitsgraph

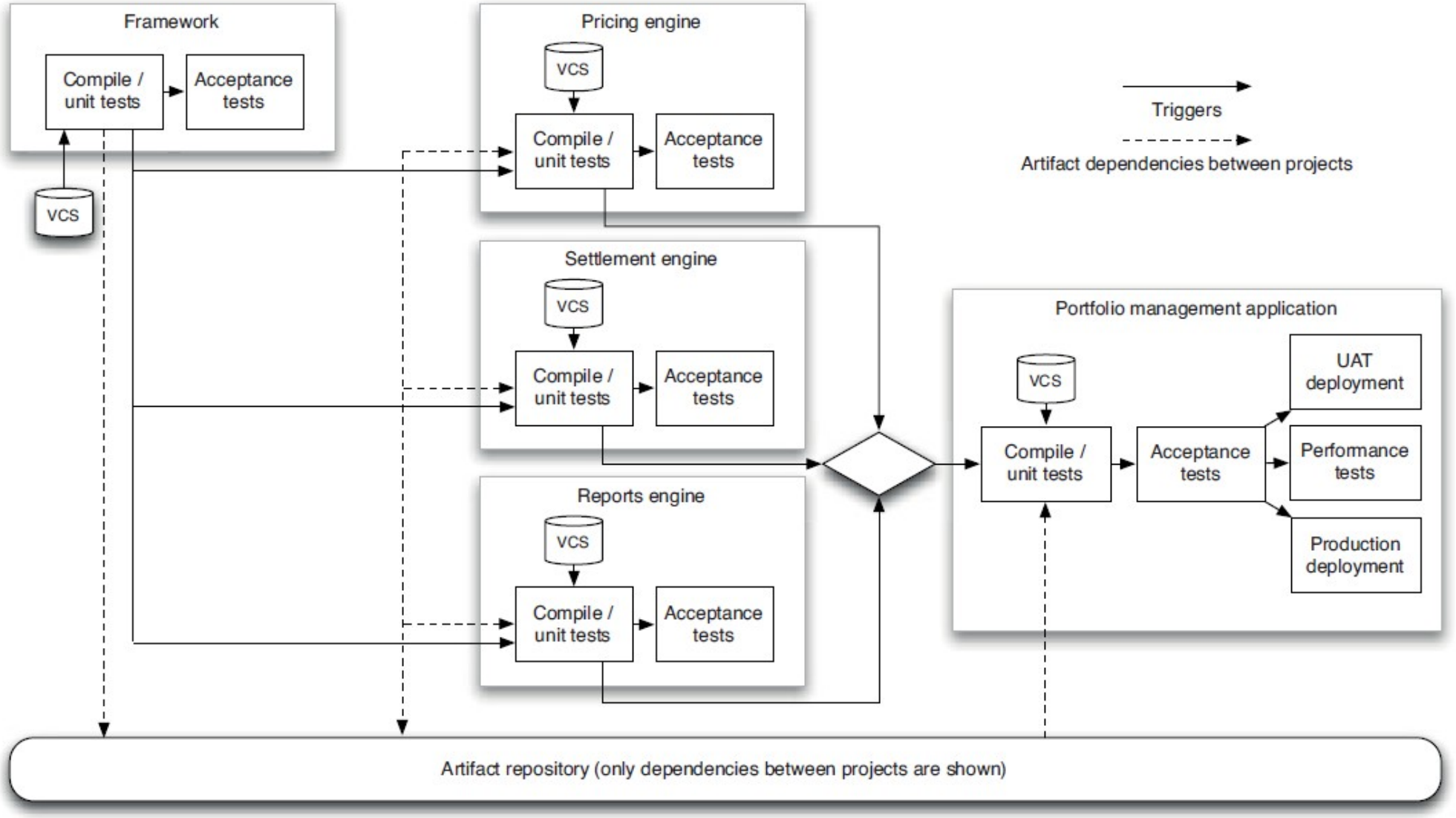


- Gerichteter azyklischer Graph (Directed Acyclic Graph = DAG)
- Sind Zyklen vorhanden besteht ein pathologisches Abhängigkeitsverhältnis
- Upstream-Abhängigkeit – entgegen der Pfeilrichtung
- Downstream-Abhängigkeit – in Pfeilrichtung
  (CDS Pricing Library ist eine fremde, binäre verwendete Komponente)

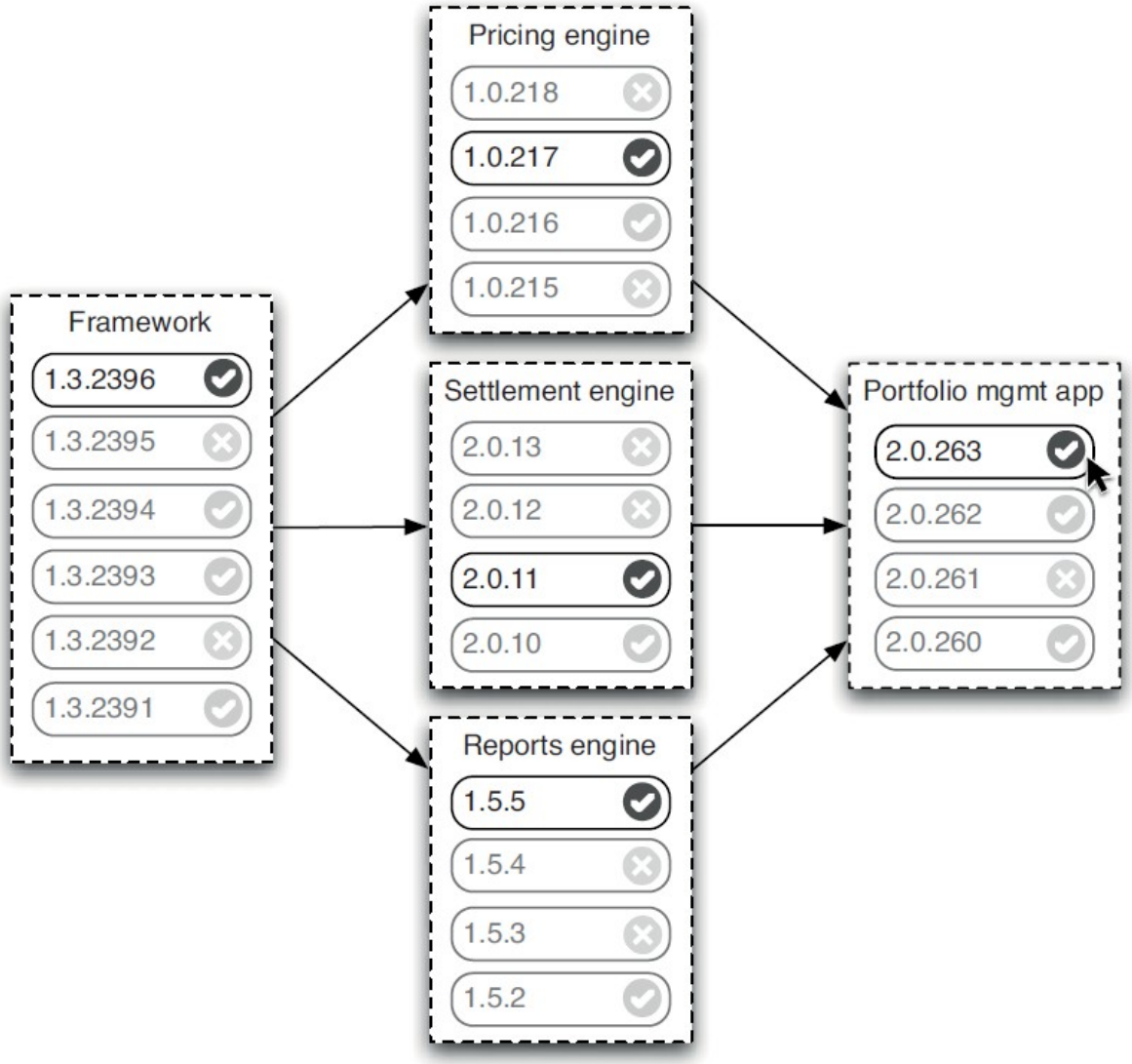
Der Abhängigkeitsgraph

- Was muß neu gebaut werden und welche Builds sind gebrochen, wenn Durchläufe nicht erfolgreich sind?
  - Portfolio-Man.-App. wird geändert → …?
  - Reports-Engine wird geändert → …?
  - CDS Pricing Library wird auf eine neue Version upgedated → …?
  - Framework wird geändert → …?
  - Framework und CDS Pricing Library werden geändert → …? Kann zu schwierigen Szenarien führen.
- Vermieden werden sollte, dass z.B. Pricing Engine und Report Engine mit verschiedenen Versionen des Frameworks gebaut werden („Diamond Dependency“)

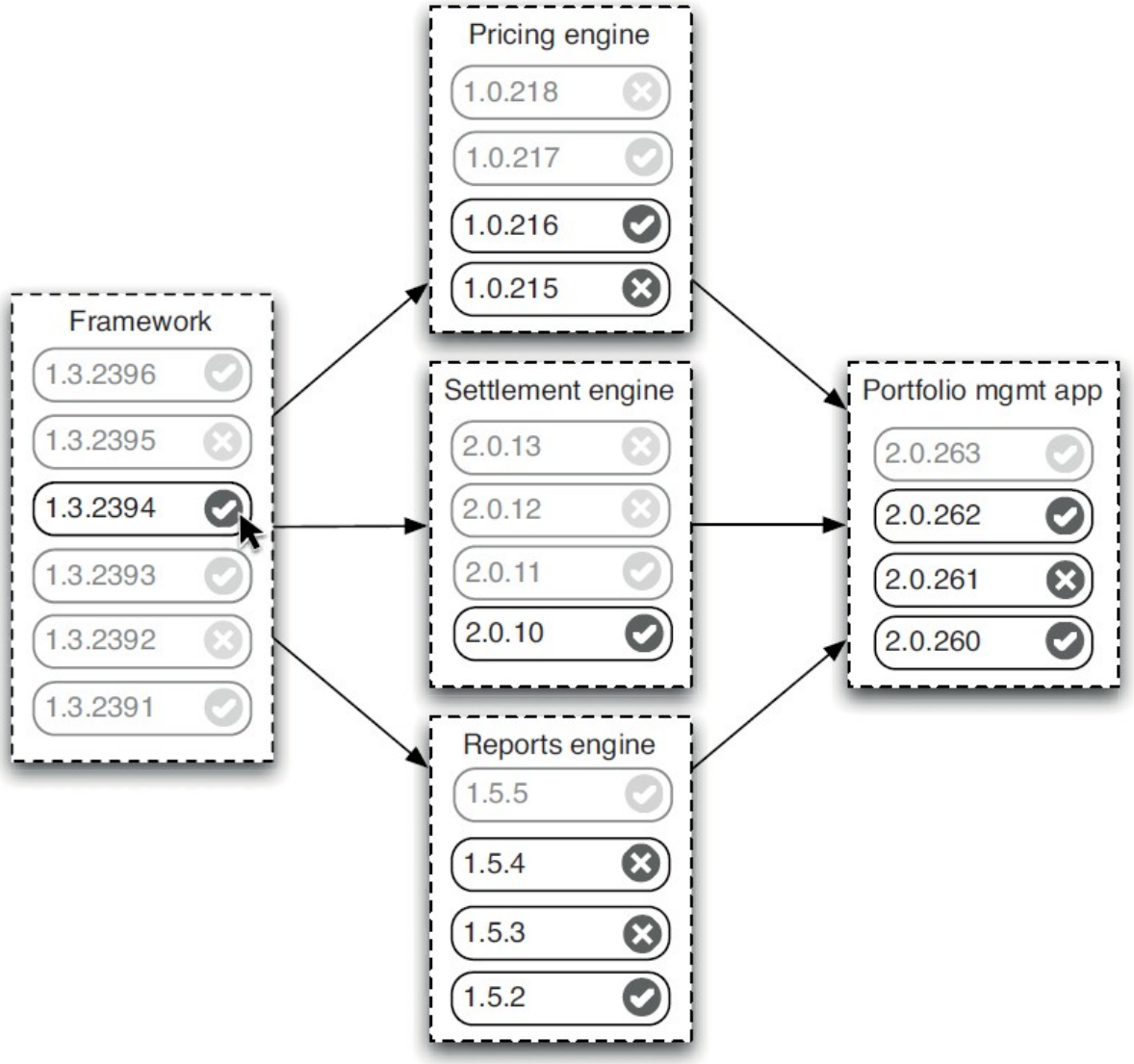
Der Abhängigkeitsgraph abgebildet auf die DP



Upstream-Abhängigkeiten verfolgen

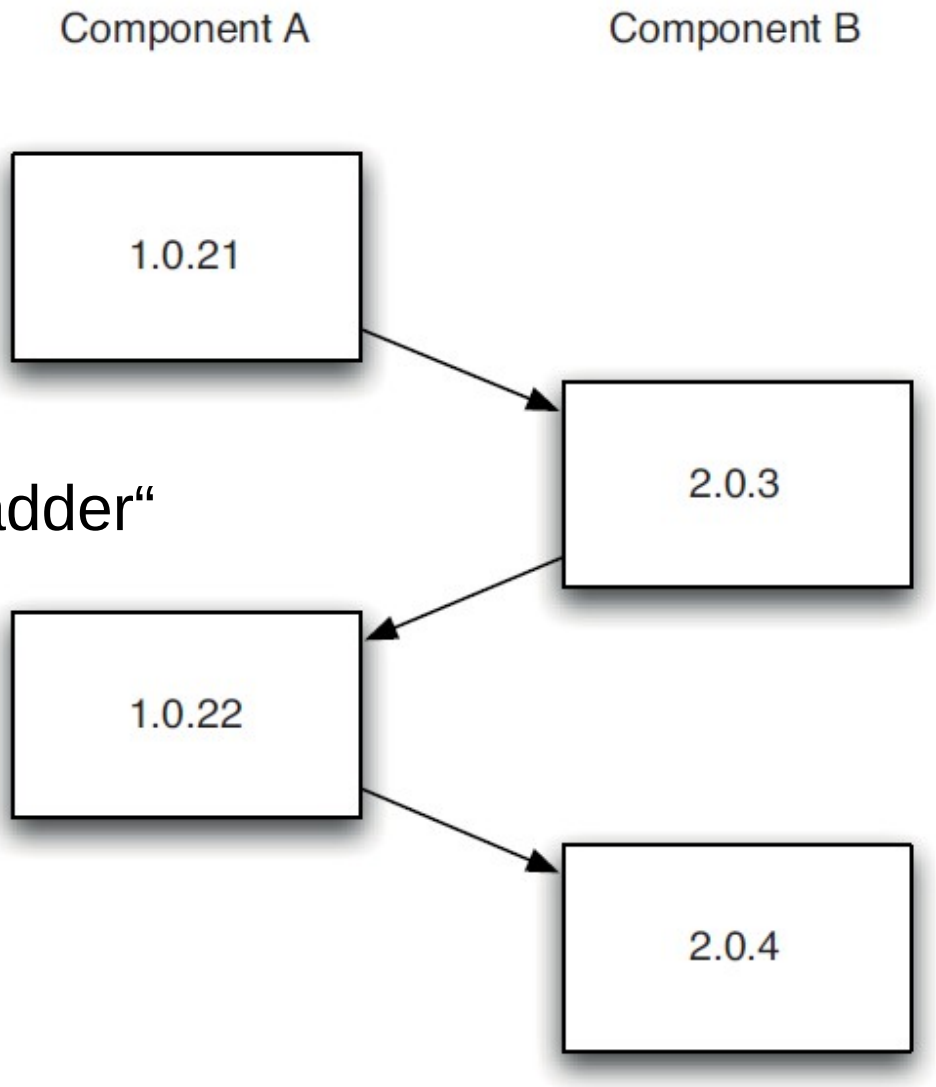


Downstream-Abhängigkeiten verfolgen



Zirkuläre Abhängigkeiten

„Build Ladder“



Wann sollte das Bauen von Komponenten getriggert werden?

- Bisher gezeigter Ansatz:
  - Neubau sobald sich die Upstream-Abhängigkeiten geändert haben
- Häufigster praktizierter Ansatz:
  - Gesamte Code-basis ist einigermaßen stabil (erreichen eines Meilensteins oder nach einem Release … etc.) → im „Rundumschlag“ werden die Abhängigkeiten aktualisiert und integriert, betont Stabilität aber potentiell auf Kosten eines höheren Integrationsaufwandes
- Kompromiß nötig:
  - „schnelle Integration“ sichert neueste Bugfixes & Features, übersichtlichere „Integrationshöhe“ ↔ aber der kleinere Integrationsaufwand muß häufiger erbracht werden
  –

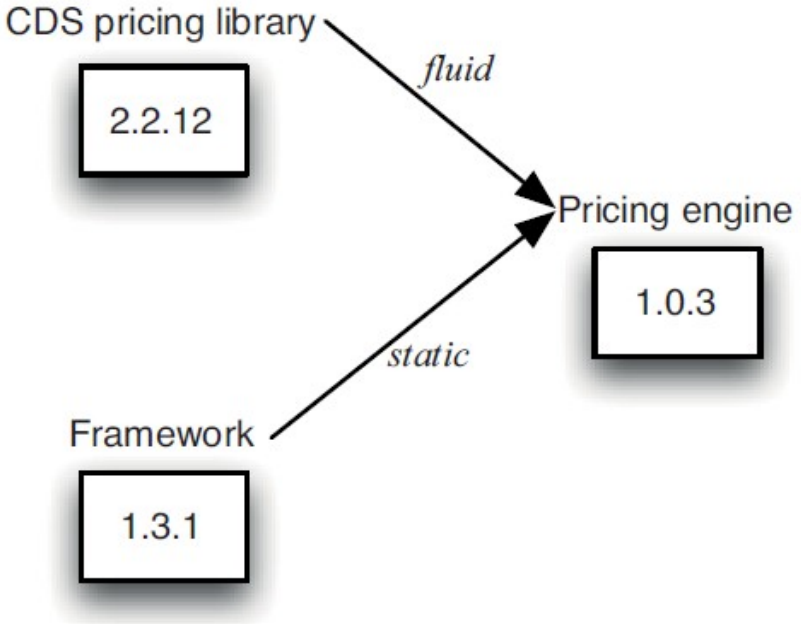
Wann sollte das Bauen von Komponenten getriggert werden?

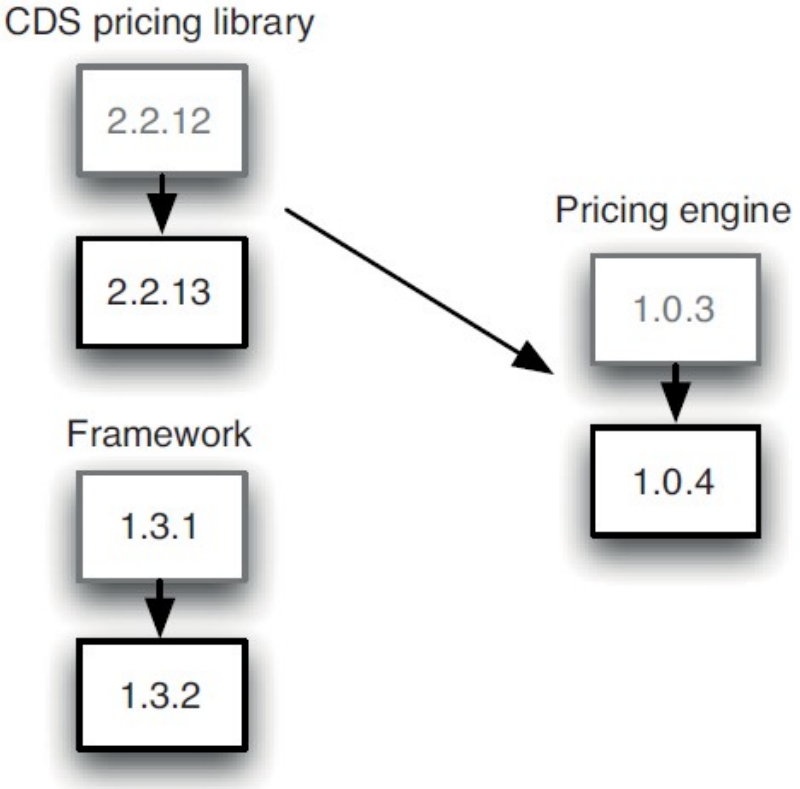
- Kriterien, die in die Kompromissbildung eingehen:
  - Wie groß ist mein Vertrauen in die Upstream-Komponenten (Stabilität und Fehlerfreiheit neuer Release‘s)?
  - Welchen Einfluß habe ich auf die Entwicklung der anderen Komponenten? Kann ich Probleme einfach kommunizieren und werden diese dann schnell behoben?
  - Je weniger Kontrolle, Überblick und Einfluss man auf eine Komponente hat, desto geringer ist das Vertrauen und desto vorsichtiger/zurückhaltender wird man integrieren.
  - Separate Deployment Pipelines sind auch eine gute Möglichkeit zur Entkoppelung der Entwicklungsprozesse
- Interessanter Vorschlag in einem Whitepaper von Alex Chaffee „Vorsichtiger Optimismus“

Cautious Optimism

- Zusätzlicher Zustand im Abhängigkeitsgraphen
- Upstream-Abhängigkeit ist
  - ***Static***
    - Änderungen triggern keinen Neubau der Downstream-Abhängigkeiten
  - ***Guarded***
    - Wie ***static*** dient lediglich als Marker für ein „gebrochenes ***fluid***“
  - ***Fluid***
    - Änderungen triggern immer einen Neubau der Downstream-Abhängigkeiten
    - Wird ein Build gebrochen, wird die Upstream-Abhängigkeit von ***fluid*** auf ***guarded*** und die letzte „gute“ Version festgelegt
- Explizit machen, welche Abhängigkeiten in welcher Frequenz integriert werden

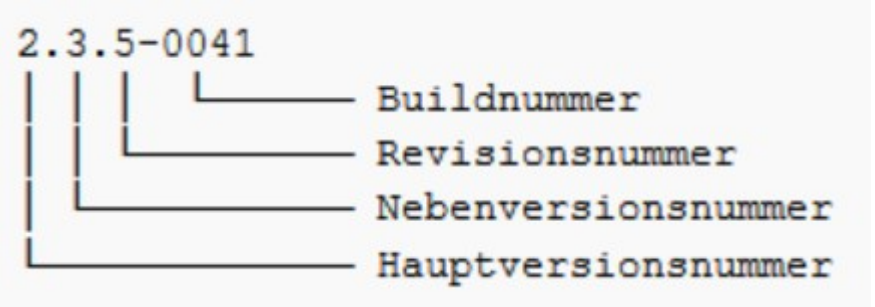
Wann sollte das Bauen von Komponenten getriggert werden?





- Kann ziemlich komplex werden
- Alternativer Strategievorschlage - „Informed Pessimism“
  - Anfangs alle Abhängigkeiten auf ***static***, aber informiert, wenn eine neue Version verfügbar wird

4.6.5 Exkurs: Semantic Versioning



- Aktuelle Version 2.0.0 der Spezifikation ist unter http://semver.org/ zu finden
- Ein Versuch (einer der vielen) gegen die „Dependency Hell“ anzukämpfen und für verschiedene Inkarnationen der „Dependency Hell“ ein Schema zu liefern (Komponenten, Dll-s, Packages, …)
- Weder ganz neu, noch revolutionär, aber es ist gut auf einen (mehr oder weniger) allgemein akzeptierten Standard zurückzugreifen
- Vor allem für die automatische Auswertung durch Package-Manager gedacht, wie z.B. ***NuGet*** oder ***npm***, und nicht so sehr für den Menschen

4.6.5 Exkurs: Semantic Versioning

- Schema ist das übliche MAJOR.MINOR.PATCH mit der Definition (deshalb „Semantic Versioning“):
  - MAJOR: API-brechende Änderungen
  - MINOR: API wird erweitert ohne die vorhergehende Version zu brechen
  - PATCH: abwärtskompatible Bug-Fixes
- Zusätzlich können noch Label über einen Bindestrich angehängt werden
- Bsp:
  - Für Pre-Releases: 1.0.0-alpha, 1.0.0-alpha.1, ...
  - Mit Build-Nummer bzw. String: 1.0.0-0.3.7, 1.0.0-x.7.z.92, …

4.6.5 Exkurs: Semantic Versioning

- Schema + 11 Regeln, inklusive Präzedenz-Regel, z.B:
  - Releases:
    - 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.
  - Pre-Releases:
    - 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0

4.6.5 Exkurs: Semantic Versioning

Dreiseitiger Artikel zu Semantic Versioning auf S. 128 in der Kategorie Wissen der c’t 24/2021

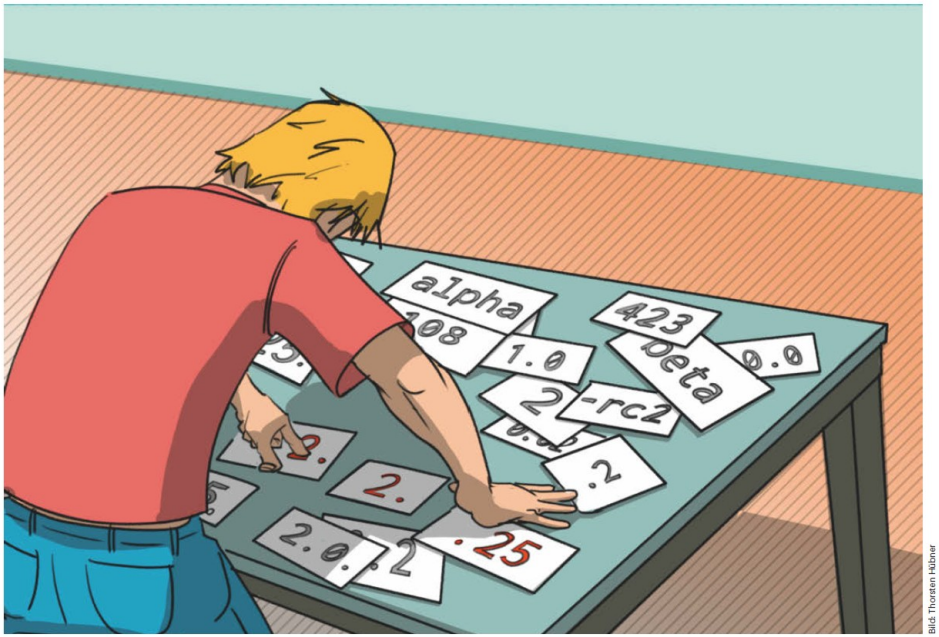


Bild: Thorsten Hübner

Bedeutung 2.0.0

Warum Versionsnummern nicht willkürlich sind

Software ist nie fertig und jedes Entwicklungsprojekt eine ständige Baustelle. Um Veränderungen und Verbesserungen zu dokumentieren, gibt es Versionsnummern. Welchem Schema sollten diese folgen und kommt nach Version 1.9 direkt die 1.10 oder gleich die 2.0? Semantic Versioning liefert klare Antworten.

Von Jan Mahn

Versionsnummern können mehr sein als eine technische Information über den Stand einer Software. Besonders runde Versionsnummern haben eine eigene Ausstrahlung: Als die Apple-Entwickler 2001 bei Mac OS die Version 10 erreichten, müssen sie sich in die Vollkommenheit dieser Zahl verliebt haben. Mac OS hieß von da an Mac OS X (also mit einer römischen 10) und machte fast zwei Dekaden keine großen Sprünge mehr. Von 2001 bis 2019 konnte Apple sich nicht von der 10 trennen und vergab nur die Versionsnummern 10.1 bis 10.15. Erst 2020 war die Zeit reif für einen großen Satz: macOS 11 mit dem Spitznamen Big Sur. Die Karriere der 11 fällt kurz aus, schon ein Jahr später, im Herbst 2021, folgt macOS 12. Auch Microsoft klebte lange an der runden 10 – 2015 brachte das Unternehmen Windows 10 auf den Markt und verkündete damals, dass dies das letzte Windows werden solle. Erst 2021 warf man den Plan über den Haufen und kündigte Windows 11 an.

Schaut man auf die Betriebssystemhersteller, könnte man den Eindruck gewinnen, dass Versionsnummern vollkommen willkürlich vergeben werden und die Marketingabteilung wesentlichen Einfluss auf die Nummerierung hat. Doch die beiden Betriebssysteme sind eher die Ausnahme. Unter Softwareentwicklern hat sich mittlerweile ein Versionsschema durchgesetzt, das wenig Raum für Willkür lässt: Semantic Versioning – also eine Versionierung mit Bedeutung. Dieses Schema

4.6.6 *Artifact-Repository* – Verwalten der Binary's

- Größter Teil der Abhängigkeiten wird binär sein
- Nur abgeleitete Artefakte landen im *AR*
- Gefahrloses Löschen muß möglich sein – jedes Artefakt läßt sich aus dem VCS wieder erzeugen – und kann auch nötig sein, da Artefakte groß werden können
- Hash-Werte des Binary mit Zuordnung zum Source-Code VCS immer aufbewahren (eventuell im Build-System) – auch wichtig für Auditing, Bug-Reports, …
- Einfachster Ansatz wäre ein selbstverwalteter Share auf einem RAID-Server, NAS, …
- Einzige Randbedingung ist die oben genannte Identifikation: f(Binary) → Source Code

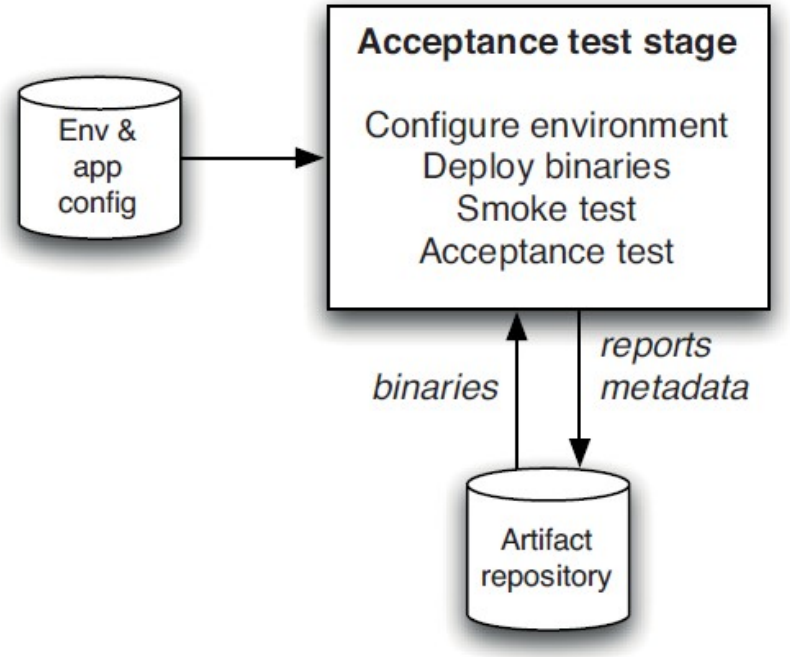
4.6.6 *Artifact-Repository* – Verwalten der Binary's

- Eigen kreierte Lösung
  - Directory pro Pipeline
    - Directory pro Build → Binary's dieses Builds
    - Eventuell ein Index-File das für jeden Build die Ergebnisse/Reports der Stufen speichert
    - …
- Oder gleich eine fertige Lösung wie *Artifactory* oder *Sonatype Nexus*
- Interaktion, z.B. Pipeline mit Compile, Unit-Tests, Acceptance Tests, Manual Acceptance Test, Production
  - Compile → speichert Binary's im AR
  - Unit-Tests → holt Binary's aus AR und speichert Reports
  - Manual Acceptance Test → …
  - ...

4.6.7 Automatisierter Deployment-Prozeß

- (Jeweils) ein Script für alle Stufen der Deployment-Pipeline (VCS-kontrolliert)
  - Anfänglich vielleicht nur eins → evolviert in eine „modularisierte Form“
    - Build-Script: Targets für Kompilieren, Binden, Packaging, Unit-Tests und statische Code-Analyse ausführen
    - Daten vorbereiten, Akzeptanztest-Env. vorbereiten → in Akzeptanzteststufe deployen
    - Script für nichtfunktionale Tests – „Stresstest“, Sicherheitstest
- Das gleiche Script für jede Umgebung, z.B. Entwicklung und Produktion – Konfiguration wird ins Script „injiziert“
- Das zur entsprechenden Umgebung passende Deployment-Tool nutzen, z.B. Ruby Gems, Python Eggs, Chef, Puppet …
- Deployment-Process muß Idempotent sein, d.h. er muß immer zu einem korrekten Zustand führen, egal wie er das Environment vorgefunden hat (vorherige Stufen können abgebrochen worden sein!?)
- Inkrementell Evolvieren

4.7 Stufe der Akzeptanztests



- Testen die „Akzeptanzkriterien“ aus einer geschäflichen Perspektive:

  Liefert die Applikation wertvolle Funktionalität für den Kunden?

4.7 Stufe der Akzeptanztests

4.7 Stufe der Akzeptanztests

- Geschäftsorientiert und nicht Entwicklerorientiert wie Unit-Tests
- Liegen auf höherer „Abstraktionsebene“ als Unit-Tests
- Testen kompletten Use-Case in einer produktionsähnlichen Umgebung
- Oft kontrovers diskutiert, Gegenargumente z.B.:
  - Implementierung und Pflege aufwendig und teuer, insbesondere dann wenn das Design nicht gut ist
  - Umfangreiche Unit-Testsuiten seien ausreichend

4.7 Stufe der Akzeptanztests

- Akzeptanztests detektieren Fehler, die Unit-Tests nicht detektieren können
  - Applikation wird insgesamt in einer produktionsähnlichen Umgebung getestet
  - z.B: komplexe Threading-Probleme, Fehlerzustände in Event-getriebenen Anwendungen, die aus dem Zusammenspiel von Architektur, Konfiguration und Umgebung entstehen
- Manuelle Aktzeptanztests können damit reduziert werden
  - Auf Dauer sind manuelle Tests noch teurer
  - Einfach wiederholbar, man erspart den menschlichen Testern langweilige Wiederholungen
- „Schutzschirm“ bei weitreichenden, übergreifenden (cross-cutting) Änderungen

4.7 Stufe der Akzeptanztests

- Möglichst reine Tester involvieren
- Notwendigkeit der Kommunikation zwischen Entwicklern, Testern und Kunden für die Implementierung der Akzeptanztests führt auch zur besseren Kooperation → stellt zusätzlich Mehrwert der implementierten Funktionalität für Kunden sicher
- Verbessern als Seiteneffekt meistens auch das Design der Applikation, z.B. dünne UI-Layer getrennt von leichter testbaren Modellen oder View-Modellen
- Möglichst wartbare Test-Suites für Akzeptanztests entwickeln
  - Kriterien für Akzeptanztests formulieren → „ausführbare Spezifikationen“ (DSL's wie z.B. SpecFlow und ähnlichem)

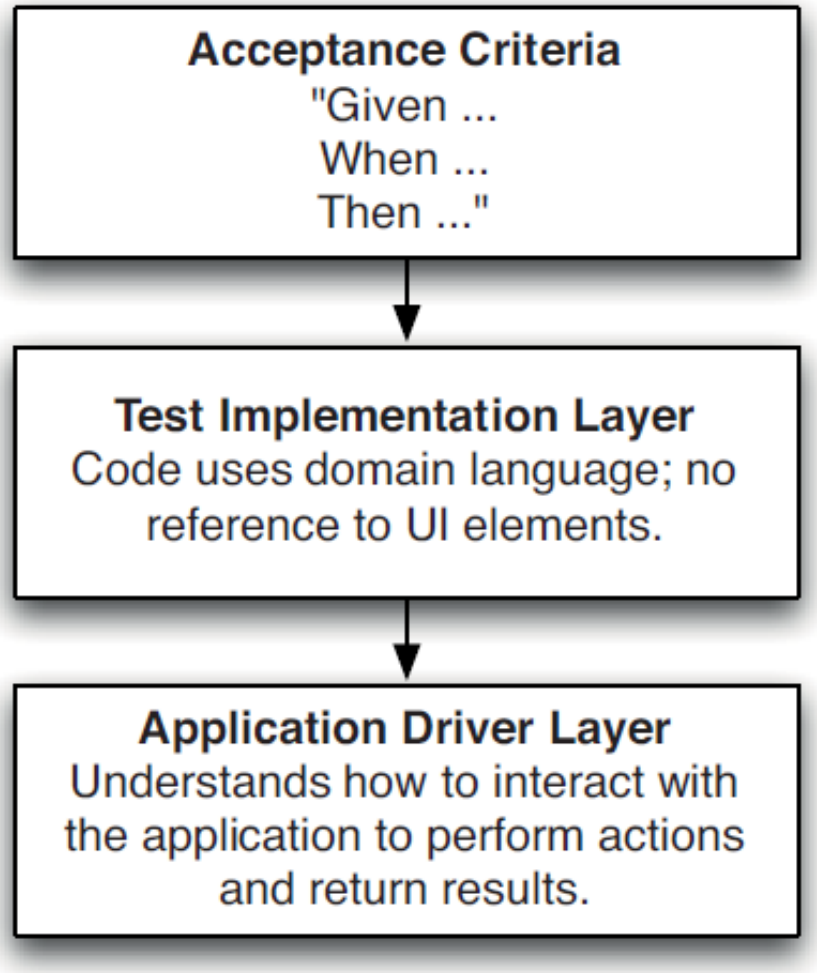
4.7 Stufe der Akzeptanztests

- Möglichst wartbare Test-Suites für Akzeptanztests entwickeln
  - Enthaltenen Tests sollen den ***INVEST***-Prinzipien folgen:
    - ***I***ndependent → unabhängig voneinander
    - ***N***egotiable → man kann sich darauf einigen
    - ***V***aluable → nutzbringende Funktionen abtesten
    - ***E***stimable → lassen sich messen
    - ***S***mall → klein
    - ***T***estable → testbar

Smoke-Test

- Grober erster Test
- Funktioniert die Applikation grundsätzlich?
- Falls er fehlschlägt, brauchen gar keine weiteren Detailtests erfolgen (Testsuites brauchen nicht gestartet werden)
- Durchführung z.B.:
  - Automatisches Deployment in die Zielumgebung
  - Anwendung starten → startet die Anwendung, geht das Hauptfenster auf?
  - Eventuell einfachste Funktionalität testen (File öffnen, … etc.)

4.7.1 Schichten



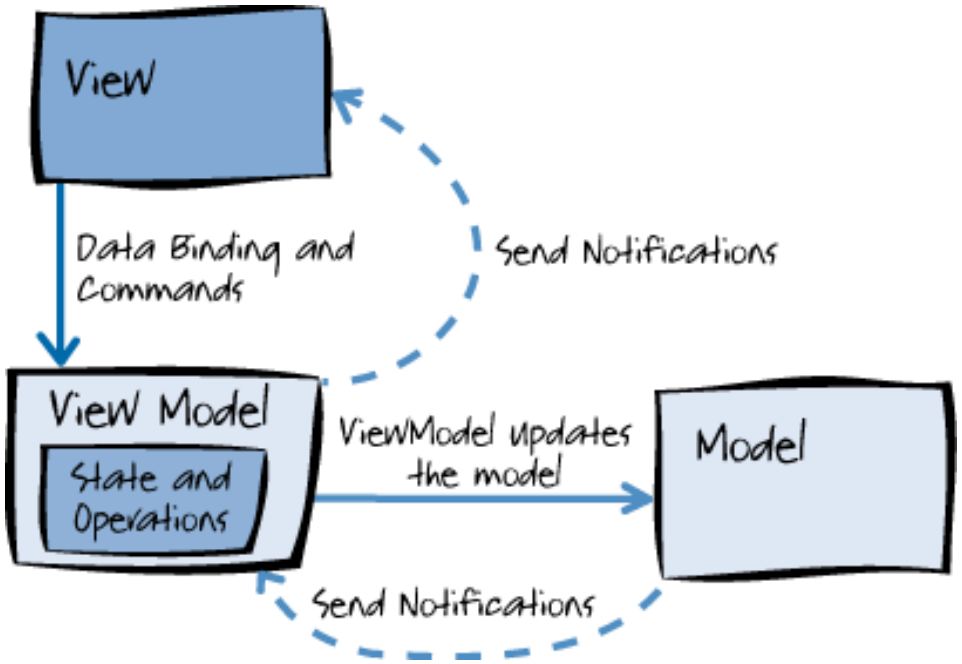
4.7.1 Schichten

- Erste und oberste Schicht definiert die Akzeptanzkriterien
  - Nutzung spezifischer Tools wie Cucumber, JBehave, FitNesse oder SpecFlow
  - Auch möglich mit Hilfe des genutzten Unit-Test-Frameworks zu schreiben
- In der Test-Implementierungsschicht keine direkte Referenz auf das API oder UI der Anwendung
  - Führt sonst zu fragilen Strukturen, die schon bei kleinen Änderungen des API‘s oder UI‘s umfangreich refactored werden müssen
  - Entkopplung der Testimplementierung von API und UI der Anwendung
- Application Driver Layer vermittel die konkrete Interaktion mit der Anwendung (API/UI)

4.7.1 Schichten

- Problem der direkten GUI-Tests
  - GUI sollte direkt getestet werden, da sonst nicht genauso getestet wird, wie der Nutzer mit dem System interagiert
  - GUI-Technologien, die nur extrem aufwendig oder nicht testbar sind
  - Oft schnelle Änderungsrate der GUI‘s
  - Komplexität der GUI‘s
  - Kleine Änderungen in der GUI können große Änderungen in der Testsuite nach sich ziehen
  - Alternativ GUI-Layer so dünn wie möglich → GUI vom Rest entkoppeln und diesen testen, Ansatz z.B. in WPF mit MVVM-Pattern

Model-View-ViewModel



https://msdn.microsoft.com/en-us/library/hh848246.aspx

- View definiert Aussehen und Layout auf der Oberfläche
  - Möglichst nur beschreibend (XAML)
- Model enthält Geschäfts- und Validierungslogik
  - POCO‘s, DTO‘s, ...
- ViewModel als Vermittler zwischen beiden
  - Verantwortlich für die View-Logik
  - Datenbindung zur View

4.7.2 Akzeptanzkriterien als ausführbare Spezifikation

- Auch bekannt als Behavior-Driven Development (BDD)
- Schema:

  Given: Anfänglicher Kontext → Anfangszustand der Applikation

  When: Eintretendes Ereignis → Interaktion der Applikation

  Then: Sich ergebende Zustände und Ausgaben → Endzustand der Applikation
- BDD-Frameworks, z.B. Cucumber

Bsp. Cucumber / Ruby

Spezifikation eines Aktzeptanzkriteriums:

```
Feature: Placing an order

  Scenario: User order should debit account correctly
  Given there is an instrument called bond
  And there is a user called Dave with 50 dollars in his account
  When I log in as Dave
  And I select the instrument bond
  And I place an order to buy 4 at 10 dollars each
  And the order is successful
  Then I have 10 dollars left in my account
```

Cucumber / Ruby – Implementation Layer

```
require 'application_driver/admin_api'
require 'application_driver/trading_ui'

Before do
  @admin_api = AdminApi.new
  @trading_ui = TradingUi.new
end

Given /^there is an instrument called (\w+)$/ do |instrument|
  @admin_api.create_instrument(instrument)
end

Given /^there is a user called (\w+) with (\w+) dollars in his account$/ do
    |user, amount|
  @admin_api.create_user(user, amount)
end

When /^I log in as (\w+)$/ do |user|
  @trading_ui.login(user)
end

When /^I select the instrument (\w+)$/ do |instrument|
  @trading_ui.select_instrument(instrument)
end

When /^I place an order to buy (\d+) at (\d+) dollars each$/ do |quantity, amount|
  @trading_ui.place_order(quantity, amount)
end

When /^the order for (\d+) of (\w+) at (\d+) dollars each is successful$/ do
    |quantity, instrument, amount|
  @trading_ui.confirm_order_success(instrument, quantity, amount)
end

Then /^I have (\d+) dollars left in my account$/ do |balance|
  @trading_ui.confirm_account_balance(balance)
end
```

Bsp. Cucumber / Ruby

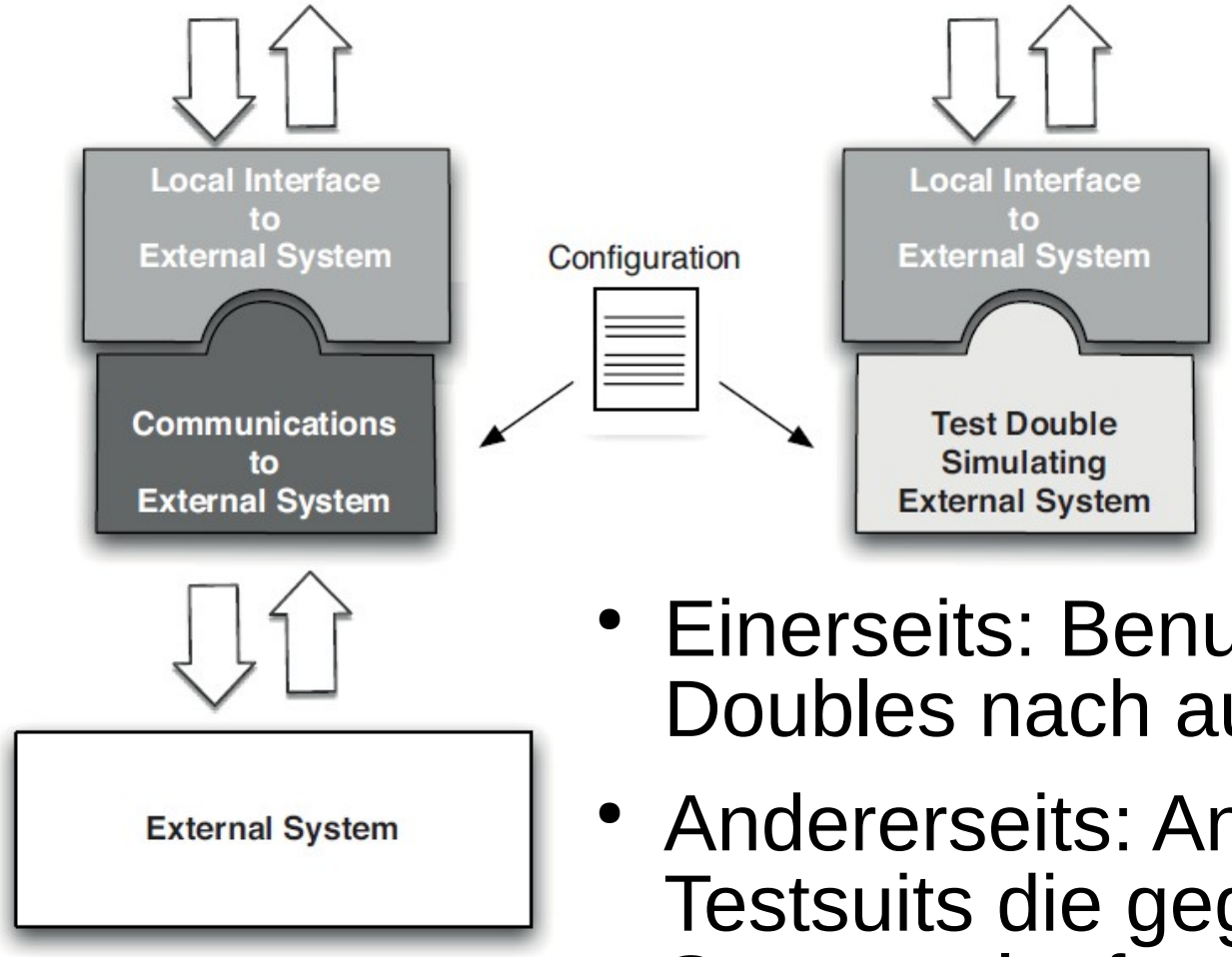
Ausgabe:

```
Feature: Placing an order

  Scenario: User order debits account correctly
                # features/placing_an_order.feature:3
  Given there is an instrument called bond
                # features/step_definitions/placing_an_order_steps.rb:9
  And there is a user called Dave with 50 dollars in his account
                # features/step_definitions/placing_an_order_steps.rb:13
  When I log in as Dave
                # features/step_definitions/placing_an_order_steps.rb:17
  And I select the instrument bond
                # features/step_definitions/placing_an_order_steps.rb:21
  And I place an order to buy 4 at 10 dollars each
                # features/step_definitions/placing_an_order_steps.rb:25
  And the order for 4 of bond at 10 dollars each is successful
                # features/step_definitions/placing_an_order_steps.rb:29
  Then I have 10 dollars left in my account
                # features/step_definitions/placing_an_order_steps.rb:33

1 scenario (1 passed)
7 steps (7 passed)
0m0.016s
```

4.7.3 Testumgebung

- Automatische Akzeptanztests hängen davon ab in einer möglichst produktionsähnlichen Umgebung ablaufen zu können
- Aber muss kontrollierbar sein → Minimieren bzw. Kontrollieren des Einflusses von externen Abhängigkeiten
- Andererseits muss aber auch der Einfluss der externen Komponenten abgetestet werden
- ***Integrationspunkte***
- Kompromiss rund um diese Integrationspunkte nötig → zweifache Strategie

4.7.3 Testumgebung

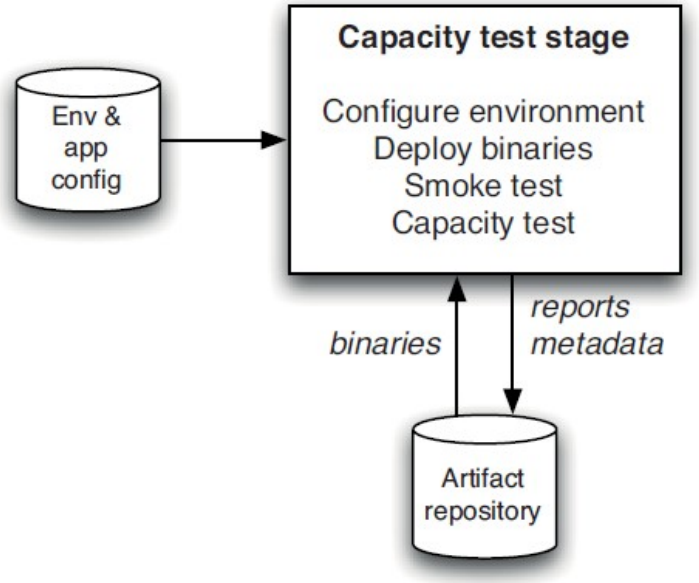


- Einerseits: Benutzen von Test-Doubles nach außen hin
- Andererseits: Anlegen von Testsuits die gegen die externen Systeme laufen

4.7.3 Testumgebung

- Vorteile:
  - Entkoppelung
  - Abhängigkeiten explizit gemacht
  - Möglichst eine Komponenten für die Kommunikation zu einem externen System
  - Möglichkeit verschiedene Szenarien zu definieren und zu simulieren, z.B. Fehlerzustände, hohe Lasten, lange Latenzen … etc.

4.8 Stufe der Nichtfunktionalen Tests



- Weites Spektrum an zu testenden Parametern:
  - Sicherheit
  - Skalierbarkeit
  - Durchsatz
  - Load-Testing
  - Ressourcenverbrauch bei langen Laufzeiten (Memory-Leaks)
  - ...

Nichtfunktionale Requirements

- Häufig einfach mit „Performance“ subsumiert
- ***„Performance is a Feature!“***
  (Matt Warren …. BenchmarkDotN

- Nichtfunktionale Requirements zu verfehlen stellt ein großes Risiko der Softwareentwicklung dar
  - Werden während des Prozesses leicht vergessen
  - Tauchen oft erst am Ende auf → besonders „Teuer“
  - Berühren oft mehrerer Aspekte der kompletten Anwendung gleichzeitig
- Einteilung in Nichtfunktionale und Funktionale Requirements ist teilweise willkürlich, deshalb Vorschläge zu anderer Namensgebung wie ***Cross-Functional Requirements*** oder ***System Characteristics***

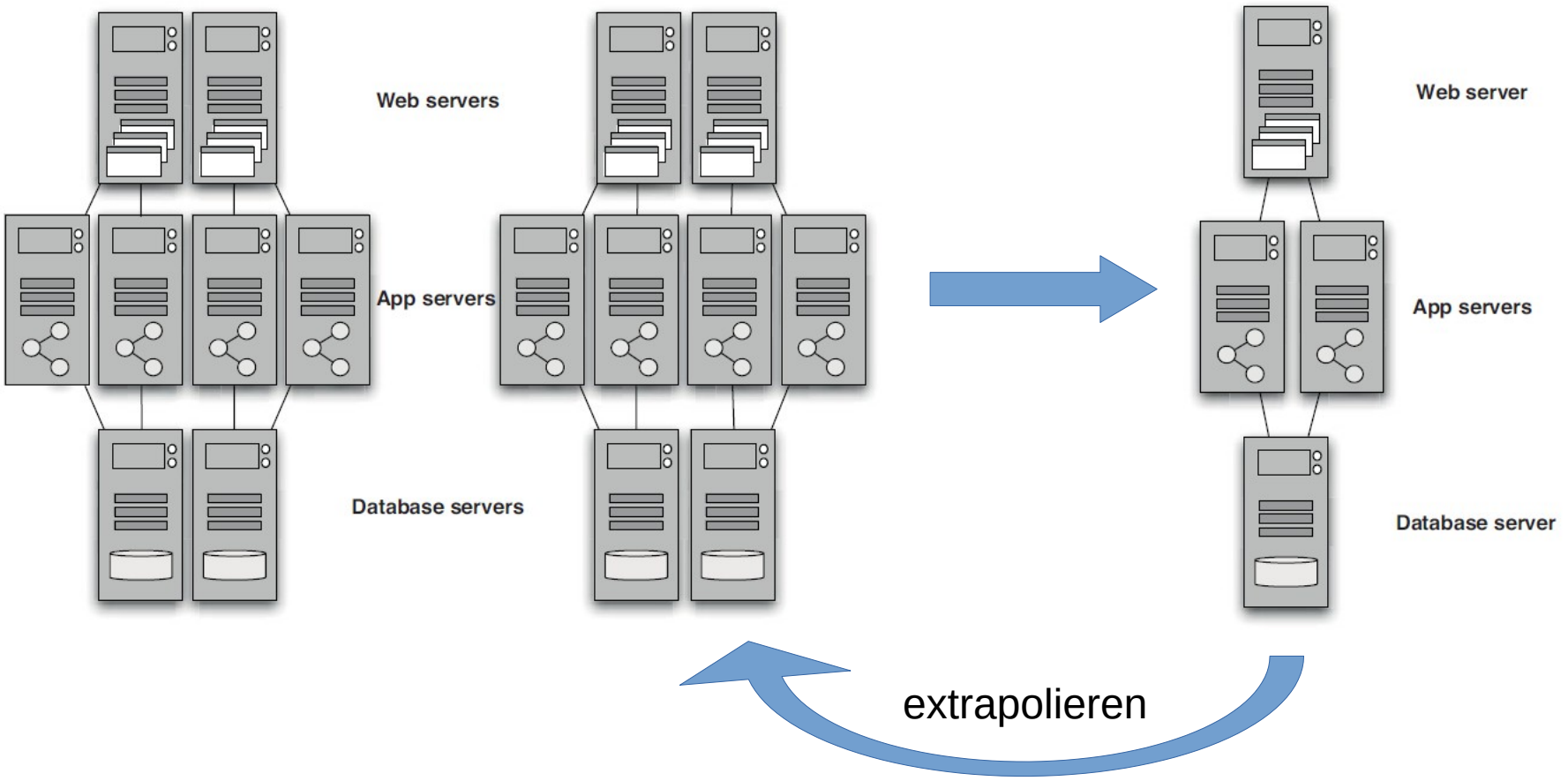
Nichtfunktionale Requirements

- Sollten vor allem auch bei der Priorisierung beachtet werden
- Müssen bis zu einem gewissen Grad schon am Anfang identifiziert, beachtet und vorausgeplant werden (architekturrelevant)
- Entsprechende Metriken müssen definiert und regelmäßig getestet werden
- Beeinflussen sich oft gegenseitig – mitunter ungünstig, z.B. hohe Sicherheit erschwert einfache Nutzung (Windows Vista)
- Aber ***Premature Optimization*** verhindern → Messen, messen! - d.h. regelmäßig testen

Testumgebung

Server Farm (Produktionsumgebung)

Eine Schicht (Testumgebung)



Testumgebung

- Möglichst geeignete, geschickt „herunterskalierte“ Testumgebung, um Kosten zu sparen
- Extrapolieren auf die Produktivumgebung

  Skalierungsfaktoren oft nicht einfach zu bestimmen → immer mit etwas Vorsicht betrachten
- Zusätzliche Vorteile:
  - Komplexe Fehlzustände in der Produktionsumgebung können nachgestellt werden
  - Tuning von Konfigurationsparametern, Garbage Collection, …
  - Simulation pathologischer Zustände, Worst-Case Szenarios, Integrationsfehler, …
  - ...

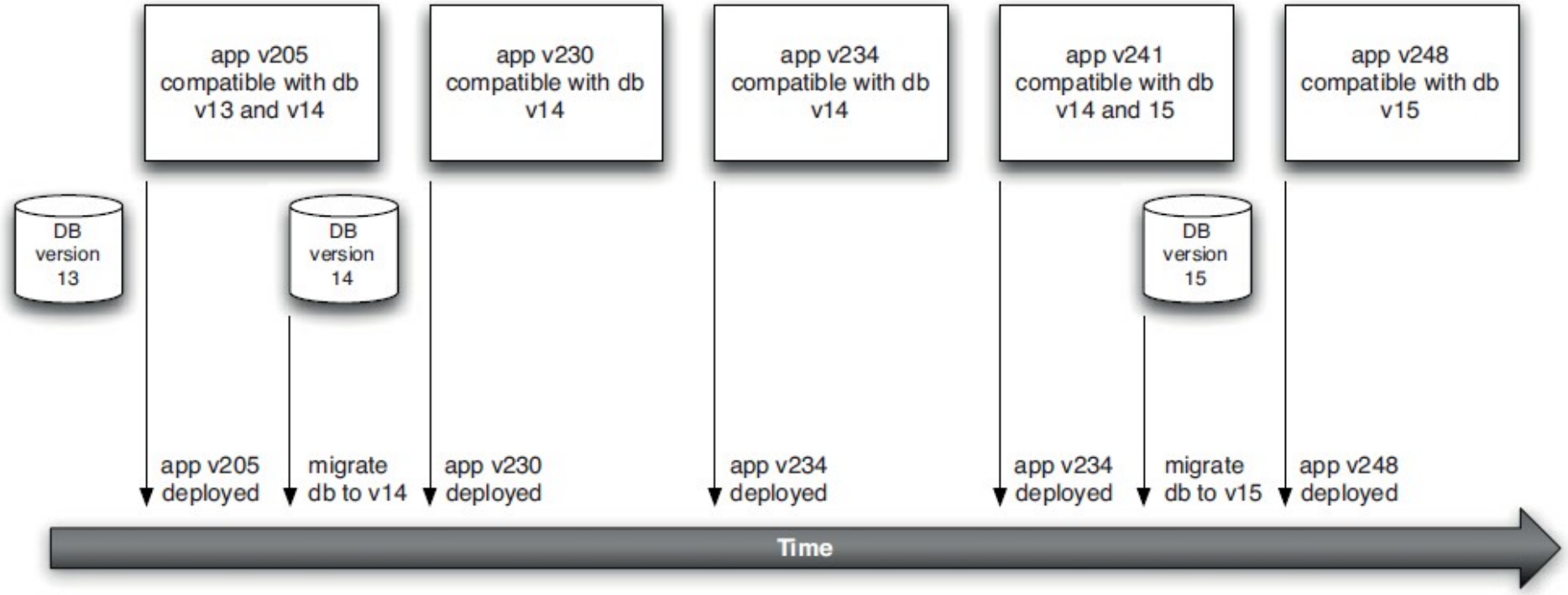
4.9 Daten und Datenbanken

- Stellt spezielle Herausforderung für Test und Deployment dar
  - Schiere Menge der Daten
  - Unterschiedlicher Lebenszyklus von Applikation und Daten der Applikation
- Daten der Anwendung sind oftmals das wertvollste im ganzen System
- Evolution der Anwendung → Datenstrukturen werden sich verändern → Modifikationen ohne Einschränkungen der Funktionsfähigkeit müssen möglich sein
- Nach wie vor stellen RDBMS den Hauptteil der verwendeten DB-Technologie dar

4.9 Daten und Datenbanken

- Unterstützung durch Tools → Datenmigrationsscripts
  - Versionierung der Datenbank
  - Migration von einer DB-Version zur nächsten
  - Entkoppelung von Entwicklungs- und Deployment-Prozess
  - Inkrementelles Vorgehen mit der Entwicklung der Applikation möglich
- Auch hier gilt wieder, so weit wie möglich automatisieren und Scripte nutzen
- Alle Initialisierungsscripte und Migrationen kommen in die Versionskontrolle des Source-Codes
- Anwendung muß beim Deployment die Datenbank mit passendem Schema vorfinden

4.9 Daten und Datenbanken

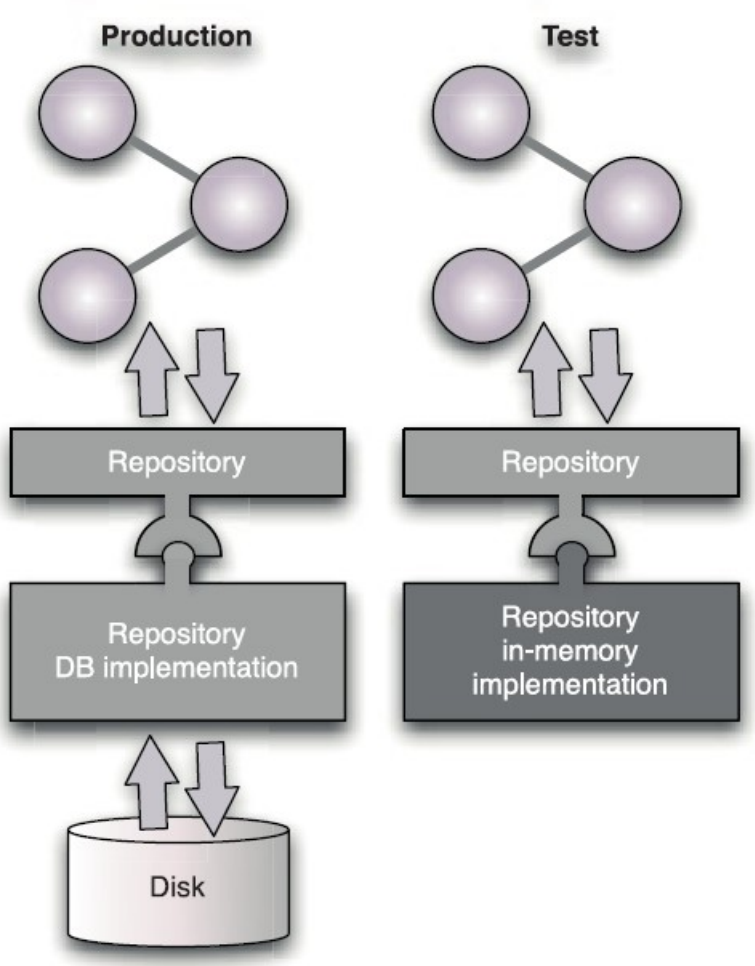


Migrationen

- Initialisierung der Datenbank
  - Alles vorherige löschen
  - Anlegen der Datenbank-Strukturen, -Instanzen, -Schema‘s und -Nutzer
  - Befüllen der Datenbank mit Daten
- Mechanismus um automatisch von einer Datenbank-Version zur anderen zu kommen:
  - Up-Migration → Roll-Forward-Script
  - Down-Migration → Roll-Backward-Script

Umgang mit Datenbanken in Tests

- Tests sollten nicht gegen eine reale Datenbank laufen
- Ersatz durch Test-Doubles
- Oft besteht sowieso eine Abstraktionsschicht zur DB hin (Repository-Pattern) → kann genutzt werden, um eine In-Memory-DB zu benutzen

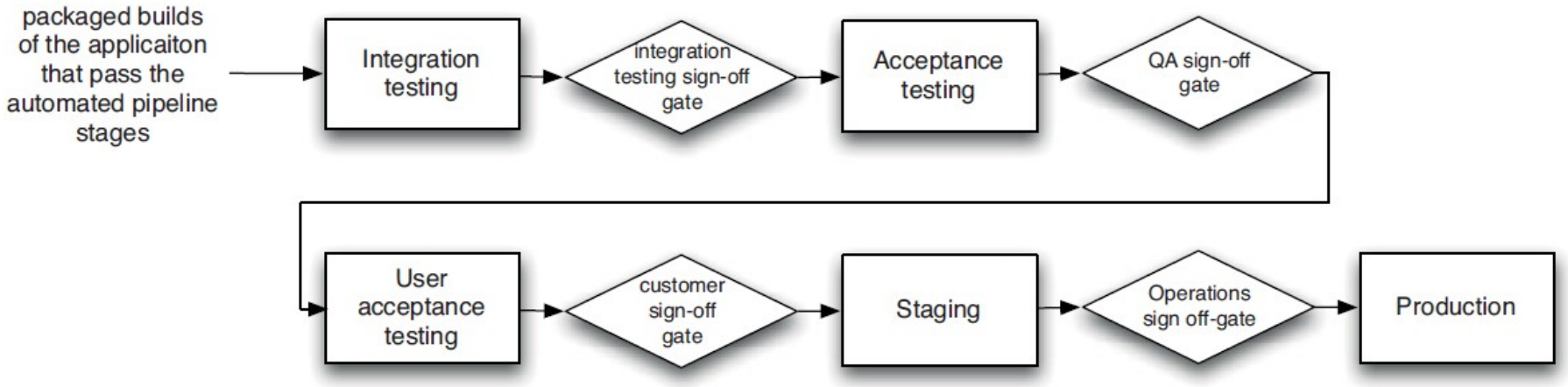


4.10 Deployment und Release

- Ausrollen einer Applikation in den Markt benötigt einen Release-Plan:
  - Schritte für ein erstes Release
  - Was muß getan werden, falls die Applikation nicht funktioniert?
  - Wie kann der Zustand der Applikation gesichert und wieder hergestellt werden?
  - Wie kann die alte Version gestartet und re-deployed werden, falls das neue Release nicht funktioniert?
  - Methoden für das Monitoring
  - Logs und Informationen über das System
  - Datenmigrationen
  - ….

4.10 Deployment und Release

- Quality-Gates auf dem Weg in die Produktion bzw. auf dem Weg zum Release:
  - Anzahl und Art der „Stationen“ auf dem Weg zum Release festlegen
  - Wer hat die Berechtigung das Ergebnis des Gates festzulegen?

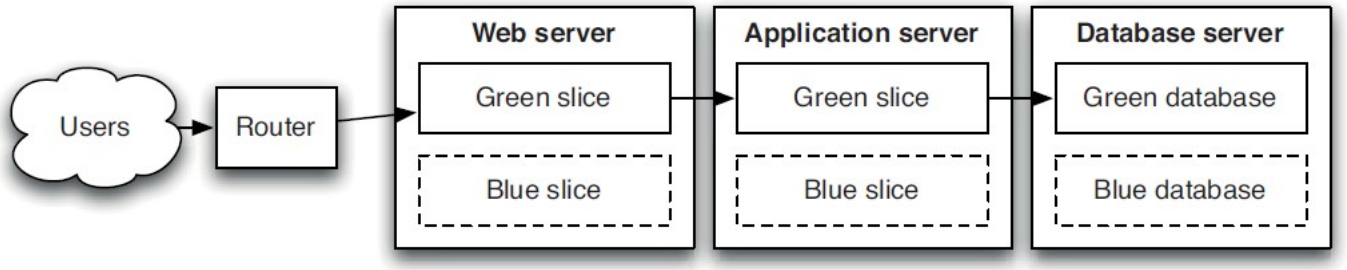


Rolling-Back Deplyoyments & Zero-Downtime Releases

- Beim Ausrollen eines neue Releases sind vor allem zwei Punkte sehr wichtig:
  - Wie kann ich die Funktionalität meines Systems möglichst schnell wieder herstellen, wenn das ausgerollte Release fehlerhaft war?
  - Wie kann ich die Funktionalität meines Systems trotz ausrollen möglichst unterbrechungsfrei sicherstellen?
- Falls man Web-, Server- oder Cloud-basierte Dienste zur Verfügung stellt → Blue-Green Deployment & Canary Release, Feature-Toggle

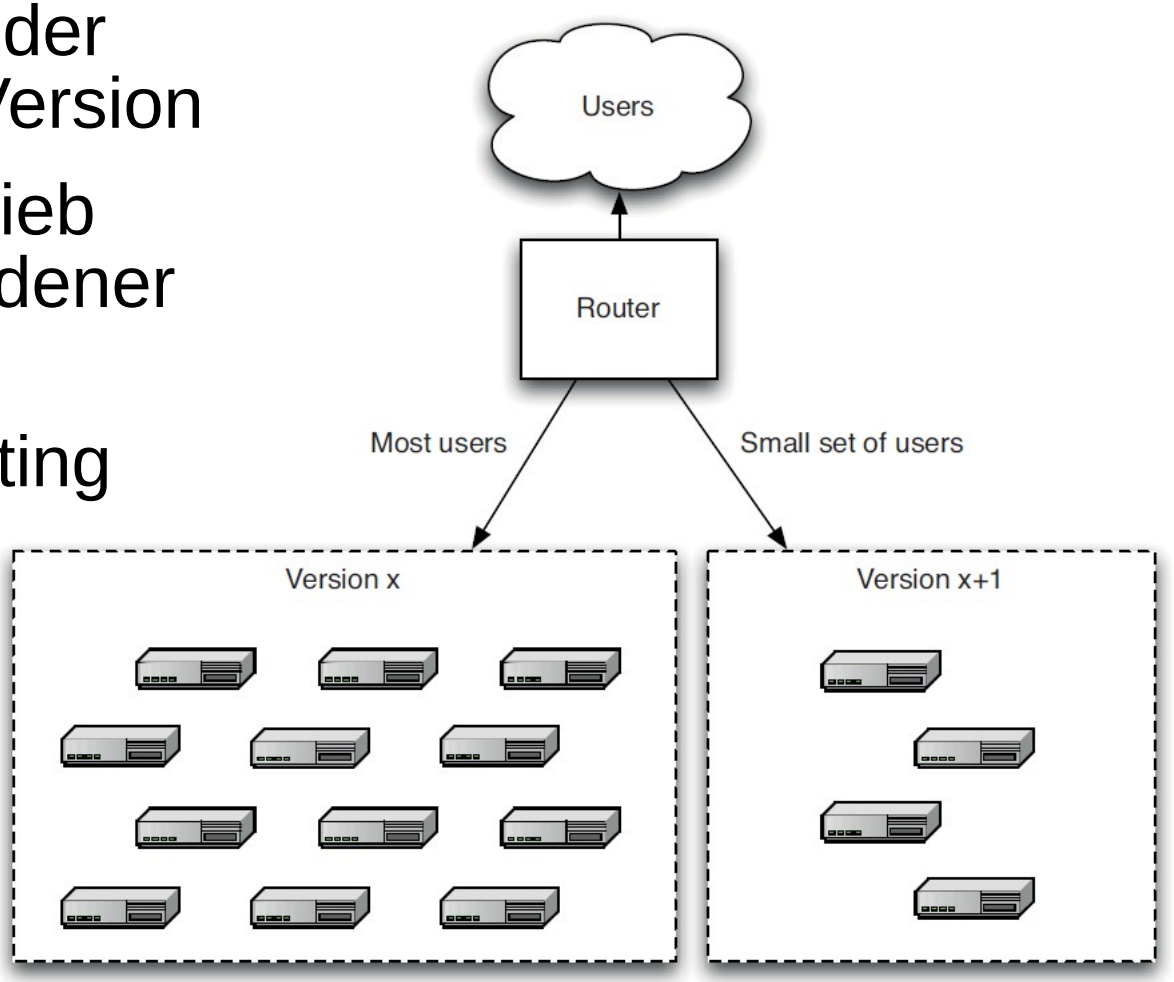
Blue-Green Deployment

- Zwei identische Versionen des Produktionssystems – eines ist produktiv, z.B. Green
- Blue erhält die neue Version → läuft, eventuell getestet → umlenken der Anwender auf Blue
- Blue wird neues Produktivsystem
- Bei Problemen zurückschalten auf Green
- Datenbank bedarf besonderer Beachtung beim Umschalten zw. den Systemen



Canary Release

- Ein Teil der Anwender benutzt die neue Version
- Gleichzeitiger Betrieb mehrerer verschiedener Versionen möglich
- Einfaches A/B-Testing möglich
- Ähnlichkeit zu Feature-Toggles



4.11 Nachteile von Continuous Delivery

- Hoher anfänglicher Mehraufwand für alle Beteiligten
- Mehr Zeit, höhere Kosten → höherer personeller Aufwand, Hardware, Schulungen, …
- Umstellung für alle Beteiligten