Seminar: Softwareentwicklung mit Dev(Sec)Ops
Sommersemester 2021
Universität Augsburg

Seminar: Softwareentwicklung mit Dev(Sec)Ops

Seminararbeit - License Checker

Jonas Kell: jonas.kell@student.uni-augsburg.de

Studiengang: B.Sc. Informatik

Abgegeben am: 20. September 2021

Inhaltsverzeichnis

1 Diskussion: Dev(Sec)Ops

Diese Arbeit stellt den schriftlichen Teil des Seminars *Softwareentwicklung mit Dev(Sec)Ops* dar, das im Sommersemester 2021 im Zuge des B.Sc. Informatik an der Universität Augsburg angeboten wurde.
Die Arbeit setzt sich zusammen aus drei Teilen: Einem allgemeinen ersten Teil, indem das Thema *Dev(Sec)Ops* (im folgenden vereinfachend oft nur *DevOps*) ohne besondere Vorgaben in einem generellen Diskurs behandelt werden soll. Einem zweiten Teil, in dem die im Seminar behandelten Werkzeuge, Anwendungen und Methoden in einem möglichst funktionalen Gesamtsystem modelliert werden. Und einem dritten Teil, der an die Präsentation, bzw. den Vortrag *Compliance: License-Checker* des Autors anknüpft, der ebenfalls im Zuge des Seminars gehalten wurde und einige der dort präsentierten Punkte nochmals schriftlich ausführt.

1.1 Warum Dev(Sec)Ops

Um zu verstehen, warum es in der modernen Softwareentwicklung nahezu unerlässlich geworden ist auf *Dev(Sec)Ops* zurückzugreifen, muss zunächst der Begriff an sich verstanden werden. *Dev(Sec)Ops* ist eine Zusammensetzung aus drei Abkürzungen und steht für *Development(Security)Operations*, also *Entwicklung(Sicherheit)Betrieb* [1].
Damit stellt DevOps ein Gerüst von Prinzipien dar, die ein Team bei der Entwicklung einer Anwendung über den ganzen Produktlebenszyklus (von der Entwicklung bis zum Betrieb) hinweg unterstützen. DevOps hat keine festgeschriebene Definition und ist keine Anleitung der man Schritt für Schritt folgen kann. Es ist vielmehr ein Sammelbegriff für verschiedene Denkweisen, Praktiken und Werkzeuge [2].
Dabei haben alle der Methodiken gemein, dass durch sie eine Verkürzung der Reaktionszeit eines Entwicklerteams ermöglicht werden kann. Wie bereits erwähnt, erstreckt sich diese Suche nach Zeitersparnis über den gesamten Lebenszyklus einer Anwendung. So kann beispielsweise durch einen *automatisierten Test*, der bereits auf der Maschine des Entwickler ausgeführt werden kann, verhindert werden, dass ein Programmierfehler in die Produktionsumgebung vordringen kann. Dies wäre ein Beispiel für die *Development* Stufe. Auf der anderen Seite lassen sich auch Verfahren betrachten, die der *Operations* Stufe beigemessen werden können. Ein *aktives Monitoring* einer Anwendung kann das Entwicklerteam sofort auf einen Ausfall eines Systems hinweisen. Im besten Fall können damit Fehler behoben werden, bevor sie beim Endnutzer zu Problemen führen. Ohne das Monitoring als DevOps Baustein, wird der Entwicklerstab im besten Fall erst herangezogen wenn die Menge an Beschwerden und Schadensmeldungen zu

hoch wird, im schlechtesten Fall gar nicht. In keinem der letzten beiden Fälle ist dies für den Ruf eines Produktes zuträglich.
Durch diese hohe Reaktivität ist die DevOps Mentalität eng mit den Methodiken der *agilen Entwicklung* verwandt [3]. In beiden Fällen ist es das Ziel möglichst *agil*, also schnell und flexibel auf (sich verändernde) Anforderungen reagieren zu können. Die agile Entwicklung stellt eine Sammlung von Organisationsformen dar, die ein flexibles und modernes Entwicklungsklima erzeugen. Sie zeichnen sich vor allem dadurch aus, dass das jedem Mitarbeiter individuell mehr Verantwortung und Vertrauen zukommt, die Teamgrößen kleiner werden, die Hierarchien flacher und die Release-Zeiten kürzer [3]. Dadurch wird die Produktivität des Teams gesteigert und es bleibt offen für Veränderungen. Eines der bekanntesten Beispiele für eine agile Methode ist *Scrum*. Dieses fokussiert sich darauf, den Entwicklern Zeit zum ungestörten Arbeiten ohne Unterbrechungen zu geben und gleichzeitig die Produktivität und Abschlussrate hoch zu halten und das Ziel nicht aus den Augen zu verlieren [4].
Bleibt noch der dritte Teil der Definition, die *Security*. Dabei ist die Idee, die IT-Sicherheit nicht als eigenständigen Abschnitt im Lebenszyklus des Projektes zu betrachten, sondern kollaborativ jeden mit in die gemeinsame Verantwortung zu ziehen, die Sicherheit *während* jeder Phase mit in den Ablauf zu integrieren [5].
Letzter Begriff, der im Zusammenhang mit Dev(Sec)Ops einhergeht, ist der des *CI/CD/CT*. Diese Reihe von Abkürzungen steht für *Continuous Integration/Continuous Delivery/Continuous Testing* [6]. Damit bilden diese drei Komponenten die Teile der sogenannten *Pipeline*, eine automatisiert ausführbare Sammlung von Werkzeugen und Skripten um den Code des Projektes bei Änderungen zu *integrieren* (z.B. durch automatisierte Versionskontrolle, Buildprozesse und Coding-Style Überprüfungen), in die Staging- oder Produktionsumgebung *auszurollen* (z.B. durch automatisches/inkrementelles Patchen auf dem Produktionsserver, kompilieren für verschiedene Betriebssysteme und Monitoring aktiver Applikationen) und in jedem Schritt des Prozesses möglichst kontinuierlich zu *testen* (z.B. durch Unittests, Komponententests oder Sicherheitstests (Pentests)).
Damit stellt das Dev(Sec)Ops einen Sammelbegriff für eine *Entwicklungs-Kultur* [7] dar, unter deren Schirm Prozesse, Technologien und Personen zusammenkommen. Dabei liegt diesen allen ein *feedbackgesteuertes, reaktives* Modell zugrunde und damit kann dies als die zentrale Idee beziehungsweise Eigenschaft gesehen werden.

1.2 Für wen ist Dev(Sec)Ops?

Im folgenden Abschnitt soll eine grundlegende Frage beantwortet werden: „Für wen ist die DevOps Kultur?“. Ist es nur eine neuartige Erscheinung, oder etwas, dessen Prinzipien sich nur die marktführenden Riesenkonzerne zu Nutze machen können?
Die kurze Antwort wird der Leser bereits erahnen können: „Für JEDEN!“. Die ausführlichere, dafür aber auch zufriedenstellendere Antwort ist komplizierter.
Warum die DevOps Kultur für jedes Entwicklerteam geeignet ist, hat zunächst eine ganz grundlegenden Ursache, die im ersten Abschnitt dargelegt wurde: Es gibt keine festen Regeln oder

Vorgaben. Möchte man DevOps Kultur in sein Unternehmen oder Projekt einfließen lassen, so muss man keine Standards erfüllen, keine Mindestmaße erreichen. Man kann soviel oder wenig wie man möchte inkrementell einführen. Zugegeben, manche Prozesse und Technologien werden in Kombination mächtiger als alleine, aber was nicht ist kann ja noch werden. Und eine Umgebung, die bereits Vorteile durch einzelne Elemente eines DevOps Prozess genießt, ist in den meisten Fällen offener für weitere Elemente.
Gerade die agilen Methoden *Kanban* oder *Scrum* [4] gehören in den heutigen Zeiten zum Quasi-Standard, was das Projektmanagement angeht. Kaum ein neues Projekt kann heutzutage noch strikt nach dem Wasserfallmodell durchgeführt werden [8]. Zu unvorhersehbar sind die Entwicklungen des Software-Marktes und zu gering die verfügbare Zeit für Planung. Ideen verwerfen und iterativ testen zu können triumphiert in den meisten Anwendungsfällen, da es keine „perfekten“ Lösungen gibt. Durch die hohe Zahl an Frameworks und Drittanbieter-Paketen gibt es oft viele verschiedene Lösungsstrategien für ein Problem. Oft ist es also gar nicht möglich, im Voraus die „ideale“ Lösung zu kennen oder gar zu planen. An einem iterativen Entwicklungsansatz führt also kein Weg vorbei.
Neben den Punkten *inkrementeller Adaption* und *agiler Methoden* stellt allerdings der dritte hier aufgeführte den wichtigsten dar: die Werkzeuge. CI/CD Werkzeuge können bereits mit kleinem Aufwand große Kosten- und Zeitersparnis verursachen. Betrachte man das einfache Beispiel: Automatisches Ausrollen in die Produktionsumgebung [9]. Diese Aufgabe ist oft simpel und mit wenigen Kommandozeilen-Befehlen erledigt. Allerdings ist die Aufgabe repetitiv (Ausrollen bei jedem Release, auf mehreren Servern), mitunter Zeitaufwändig (Lange Build-Zeiten / Downloads) und unsicher (wenn jeder Zugriff auf die Produktionsumgebung hat, ist dies ein Sicherheitsrisiko). All dies macht den Prozess fehleranfällig bei manueller Durchführung, jedoch ideal für das Ausführen durch eine Maschine. Der initiale Mehraufwand durch die Konfiguration der Werkzeuge wird in der Regel bereits nach wenigen Zyklen ausgeglichen sein.
Es lässt sich also zusammenfassend sagen, dass jedes Element aus dem Bereich der DevOps-Kultur ein Team bereichern kann, ohne es dabei einzuschränken. Da es also praktisch nur Vorteile mit sich bringt, eignet sich DevOps im Prinzip für jeden.

2 Die „ideale“ Dev(Sec)Ops Pipeline

Bereits im letzten Abschnitt wurde klar: Jeder kann sich Elemente der DevOps-Kultur aneignen und nach belieben kombinieren. Eine „Musterlösung“ für den idealen Prozess kann es also nicht geben.
Trotzdem gibt es Anordnungen der CI/CD/CT-Pipeline, die Vorteile gegenüber anderen Anordnungen mit sich bringen.
In diesem Kapitel soll eine Pipeline beispielhaft modelliert und nach einer persönlichen Einschätzung hin optimiert werden.

2.1 Welche Werkzeuge kommen zum Einsatz?

Die Zahl der CI/CD-Werkzeuge, die bei der Konstruktion einer Pipeline zur Verfügung stehen ist groß [10] [11]. Würde man versuchen jedes Werkzeug zu integrieren / zu thematisieren, so könnte man keinem gerecht werden. Aus diesem Grund soll an dieser Stelle eine Auswahl an Werkzeugen behandelt werden. Diese setzt sich zusammen aus den Werkzeugen die im Seminar behandelt wurden, Teil von Vorträgen waren oder vom Autor als wichtig erachtet werden.
Die folgenden Abschnitte bieten eine Übersicht über die betrachteten Werkzeuge. Dabei werden diese in die Kategorien CI/CT/CD eingeteilt. Es wird eine kurze Erklärung gegeben, was das Werkzeug bewirkt und mit welchem realen Produkt die Stufe besetzt werden könnte.

2.1.1 Abschnitt: Continuous-Integration

1. **Coding Style Guidelines**
   Überprüfung, ob die *Coding-Conventions* eingehalten werden. Sollte direkt in der IDE des Entwicklers überprüft werden. Hierfür kann ein *Linter* [12] zum Einsatz kommen. Welcher ist dabei von der Programmiersprache abhängig.
2. **Code Architecture**
   Überprüfung, ob *architekturelle Konventionen* und *Namenskonventionen* eingehalten werden. Sollte direkt in der IDE des Entwicklers überprüft werden. In den Seminarvorträgen wurde hierfür das Werkzeug *Arch Unit* [13] ausgewählt.
3. **Commit-Conventions**
   Um die Einhaltung der *Commit-Merge*-Konventionen garantieren zu können, sollten Regeln im Dependency-Manager gesetzt werden. GitHub und GitLab beispielsweise können so konfiguriert werden, dass Merges nur von bestimmten Branches den Produktions-Branch vorgenommen werden dürfen. Durch einfache Skripte kann auch das Format der Commit-Nachrichten überprüft werden [14].

4. **(Pre-) Versioning**
   In der Softwareentwicklung ist die Vergabe inkrementeller Versionsnummern für die bessere Übersichtlichkeit entscheidend [15]. In der Pipeline können für bestimmte Merges Versionsnummern entweder geprüft oder sogar automatisch vergeben werden. Mit kurzen Skripten kann auch ein automatisiertes *Tagging* von Releases im Dependency-Manager vorgenommen werden.
5. **License-Checker**
   Um den korrekten Umgang mit (Open-Source)-Lizenzen frühzeitig zu unterstützen, können automatisierte Lizenz-Überprüfungen vorgenommen werden. Hierauf wird in Kapitel 3 genauer eingegangen.

2.1.2 Abschnitt: Continuous-Testing

6. **Unit-Tests**
   Die Grundlage eines Test-Frameworks wird von den sehr schnell ausführbaren *Unit-Tests* gebildet. wie die Tests ausgeführt werden, ist von der Programmiersprache abhängig. Zum Einsatz kommen können beispielsweise *jUnit* oder *phpUnit*.
7. **Vulnerability-Checker**
   Mithilfe dieses Werkzeuges können Schwachstellen oder Sicherheitslücken in Drittanbieter-Bibliotheken erkannt werden und die Kompatibilität von Abhängigkeiten kann gewährleistet werden. In den Seminarvorträgen wurde hierfür das Werkzeug der Plattform *Snyk* eingesetzt [16].
8. **E2E-Tests**
   Um die Funktionsweise einer vollständigen Anwendung in der Produktionsumgebung zu testen, kann ein sogenannter *End-to-End Test* eingesetzt werden. Im Bereich der Webentwicklung können hier beispielsweise *Browsertests* mit dem Werkzeug *Selenium* durchgeführt werden [17].
9. **Dynamic Application Security Testing (DAST)**
   Eine Vielzahl von Sicherheitslücken wird durch wiederkehrende Programmierfehler verursacht. Um dem vorzubeugen, kann nach häufigen Sicherheitslücken in einer Applikation automatisiert gesucht werden. In den Seminarvorträgen wurde hierfür das Werkzeug von GitLab eingesetzt [18].
10. **Application Security Management; Test/Code Coverage Statistiken**
    Um die Informationen aus Tests und anderen Quellen möglichst gezielt verwenden zu können, sollten diese zentral aggregiert und einheitlich zusammengefasst werden. Dieser Schritt dient mehr der Gesamtübersicht, anstatt neue Informationen zu gewinnen.

2.1.3 Abschnitt: Continuous-Delivery

11. **Application Building**
    Automatisiertes Kompilieren von Packen von Code zu ausführbaren Dateien oder Anwendungen kann durch einfache Skripte in grundlegenden Pipelines vorgenommen werden.

Besonders ist dabei, dass leicht für verschiedene Prozessorarchitekturen und Betriebssysteme kompiliert werden kann.

12. **Application Deployment**
    Automatisches ausrollen das Code in die Produktionsumgebung wird von praktisch jedem Pipelinesystem nativ unterstützt. Sollte ein spezielles Vorgehen erwünscht sein, kann zusätzlich mit SSH und einfachen Skripten unterstützt werden. Fokus sollte auf der permanenten Erreichbarkeit der Produktionsumgebung liegen, was beispielsweise mit *Zero Downtime Deployment* erreicht werden kann [19].
13. **Application Monitoring**
    Um die kontinuierliche Verfügbarkeit der Produktionsumgebung zu garantieren und auf Anomalien reagieren zu können, sollte aktives *Monitoring* verwendet werden. In den Seminarvorträgen wurde hierfür das Werkzeug *Sentry* eingesetzt [20].

2.2 Wie ordnet man seine Pipeline am besten an?

Nicht nur die Wahl der Werkzeuge ist entscheidend, sondern auch deren Reihenfolge und Anordnung [21]. Sinnvolle Priorisierung und Parallelisierung können die Feedback-Zeit verringern und die Server entlasten.
Die Pipeline ist mit *Integration*, *Testing* und *Delivery* in drei Hauptbestandteile zerlegt. Grafisch ist die Pipeline in Abbildung 2.1 dargestellt. Die Nummerierung entspricht den entsprechenden Werkzeugen, die in *2.1 Welche Werkzeuge kommen zum Einsatz?* aufgelistet wurden.
Hauptziel dieser Pipeline ist, das Feedback möglichst zeitnah liefern zu können. Daher wurde versucht, die Werkzeuge möglichst so anzuordnen, dass die schnellen Tests und Überprüfungen zuerst durchgeführt werden. *Werkzeug 1: Coding Style Guidelines* kann praktisch in Echtzeit ausgeführt werden und liegt deswegen am Anfang des Prozesses. Der statische Check von *Werkzeug 2: Code Architecture* kann ebenfalls in der IDE des Entwicklers ausgeführt werden. Bei großen Projekten sollte für diese Stufe auch ein Job in der Pipeline eingesetzt werden. *Werkzeug 3: Commit-Conventions* und *Werkzeug 4: (Pre-) Versioning* gehen mit jedem Commit in die Versionskontrolle einher. Damit die Qualität der Commits im weiteren Verlauf garantiert ist, werden diese zwei Schritte als erstes Element in der tatsächlichen Pipeline aufgeführt. *Werkzeug 5: License-Checker* wird früh ausgeführt, da es sich um eine statische Analyse handelt, die schnell durchgeführt werden kann. Zudem können fehlerhafte Lizenzen den Gebrauch eines Paketes gänzlich unmöglich machen. Aus diesem Grund sollte diese rechtliche Überprüfung agbeschlossen werden, bevor die „teuren“ Testverfahren gestartet werden.
Damit kann auch auf die makroskopische Funktionsweise der Pipeline nochmals eingegangen werden. Wie in Abbildung 2.1 gut erkannt werden kann, wird die Pipeline in die drei Hauptäste *Integration*, *Testing* und *Delivery* unterteilt. Durch den Verlauf der Pfeile wird klar, dass diese nacheinander durchlaufen werden müssen. Somit ist sichergestellt, dass der *Testing*-Abschnitt nur nach erfolgreicher *Integration* durchgeführt werden kann. Ebenso der *Delivery*-Abschnitt nur nach erfolgreichen *Tests*. Bei Fehlschlag eines Werkzeuges, ist jedoch immer der schwarze Weg zurück zum Anfang zu nehmen, um and den nicht ausgeführten Ästen vorbei zurück zum

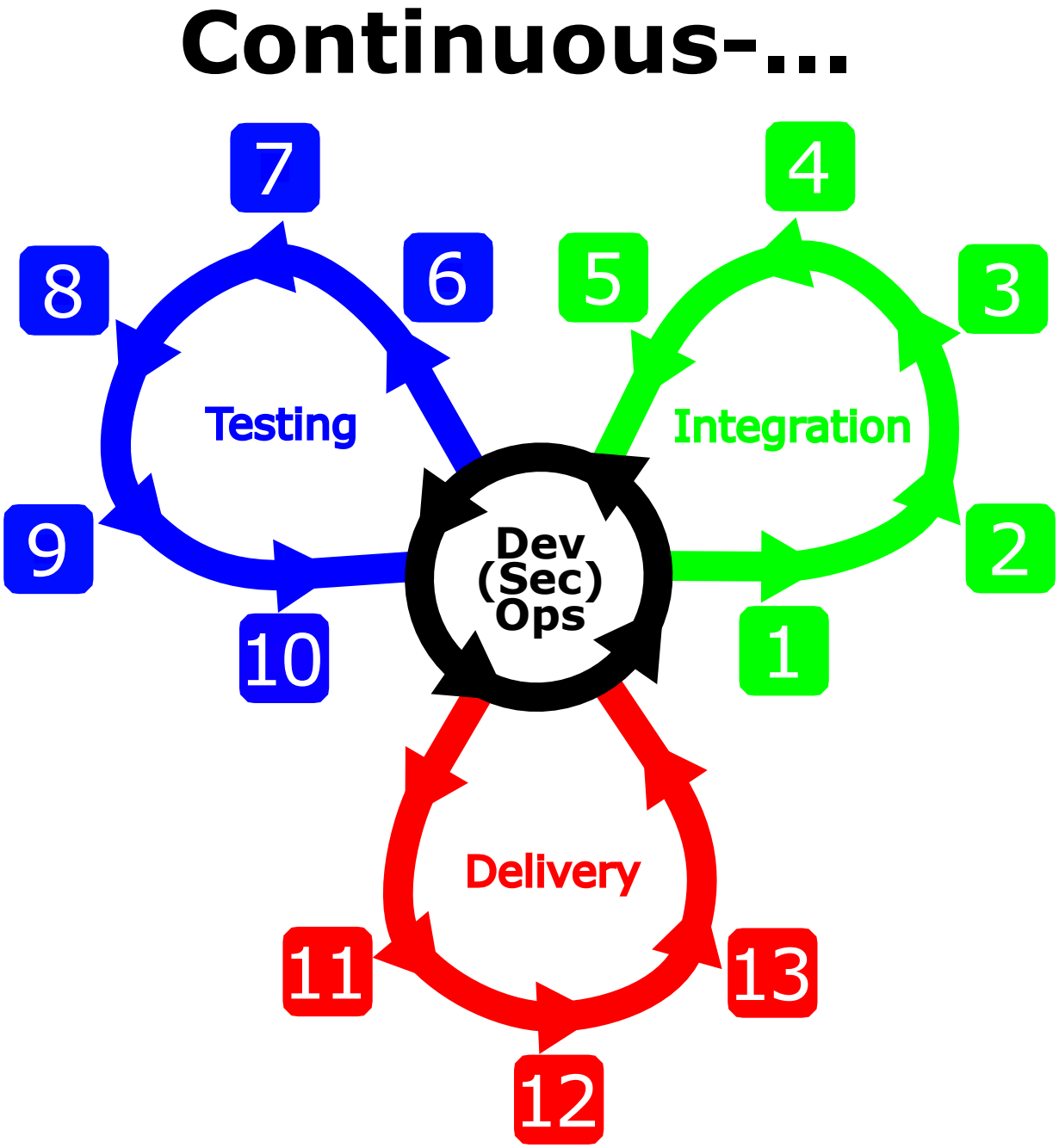

Abb. 2.1: Grafische Darstellung des „idealen“ Dev(Sec)Ops Prozesses in Form einer CI/CT/CD-Pipeline. Die drei Arme stellen die drei Phasen (Integration, Testing und Delivery) dar. Die Pfeile geben dabei die Richtung und den Weg an, in dem eine Änderung in die Produktionsumgebung über gehen kann. Die Zahlen entsprechen dabei mit ihrer Nummerierung den Werkzeugen aus 2.1, wie sie in der Pipeline angeordnet werden. Betont werden soll, dass über den inneren Kreis Stufen wiederholt werden können. Scheitert beispielsweise eine Änderung an einem Werkzeug in der *Testing*-Phase, so wird die *Delivery*-Phase innen übersprungen und mit der *Integration*-Phase erneut begonnen. Damit soll betont werden, dass Feedback von jedem Punkt schnell wieder zum Ausgangspunkt zurückfließen kann.

Start einer neuen Iteration zu gelangen. Dadurch ist die Nacheinander-Ausführung aufeinander aufbauender Test gewährleistet, und es müssen trotzdem (vor allem bei einem Fehlschlag) nur so wenige Schritte ausgeführt werden wie nötig.
Auch im *Testing*-Abschnitt bauen die Tests von niedriger zu hoher Laufzeit und Komplexität aufeinander auf. Der Abschnitt beginnt mit *Werkzeug 6: Unit-Tests*. Diese sind in den meisten Fällen noch schnell ausführbar ohne viel Rechenleistung zu benötigen. Trotzdem decken sie bereits einen großen Teil der Funktionen eines Projektes ab. In der mittleren Phase des *Testing*-Abschnittes kommt das erste Mal Parallelisierung zum Einsatz. *Werkzeug 7: Vulnerability-Checker*, *Werkzeug 8: E2E-Tests* und *Werkzeug 9: DAST* arbeiten alle mit externen Pipelines oder vollständigen Versionen der Applikation. Tests auf diesen Umgebungen sind zumeist mit Wartezeiten oder der Nutzung höherer Rechenleistung verbunden. Dadurch können diese Abschnitte mehr Zeit in Anspruch nehmen als alle vorhergegangenen Werkzeuge. Aus diesem Grund bietet sich hier sehr gut an, diese „teuren“ Tests erst hier und möglichst parallel auszuführen, um weniger häufig lange auf das Ergebnis warten zu müssen. *Werkzeug 10: Test/Code Coverage Statistiken* schließt diesen Abschnitt ab. Da hier die Informationen aus den vorherigen Tests aggregiert werden, kann dieses Werkzeug nur hier platziert werden.
Nur eine vollständig getestete Applikation kann in den *Delivery*-Abschnitt vordringen. Die Reihenfolge innerhalb dieser Stufe ist logisch vorgegeben. Zuerst muss der Code mit *Werkzeug 11: Application Building* kompiliert und gepackt werden, damit die daraus resultierende Anwendung mit *Werkzeug 12: Application Deployment* in die Produktionsumgebung übergeben werden kann. Diese Umgebung kann im letzten Schritt mit *Werkzeug 13: Application Monitoring* überwacht werden. Von dieser Stufe ist auch der Weg zur Integration am kürzesten, um im Fehlerfall mit direkt mit einer neuen Entwicklungs-Iteration beginnen zu können.
Zusammenfassend lässt sich diese Pipeline als „ideal“ bezeichnen, wenn das Ziel ist, einen Fehlschlag immer möglichst früh abzufangen. Damit wirken sich die Wartezeiten in den späteren Stufen seltener auf den Programmierer aus und es können möglichst viele der Probleme über die billigen Teststufen abgefangen werden. Dies stellt nach der Meinung des Autors die effizienteste Nutzung der Ressourcen Zeit und Rechenleistung dar und ist daher anderen Formen der Ausführung überlegen.

3 Dev(Sec)Ops Werkzeug: License Checker

Im letzten Kapitel wird auf eine Anforderung des begleitenden Vortrages *Compliance: License-Checker* eingegangen. Im Zuge des Seminars sollten Nachforschungen zu einem ausgewählten Bestandteil des DevOps Prozesses angestellt werden. In diesem Fall wurde als Thema die *License-Compliance* gewählt, also das Aufbauen unternehmensinterner Richtlinien und Methoden um bei der Arbeit mit Fremdsoftware die Lizenz Bestimmungen einzuhalten [22] [23].
Neben der Behandlung des Themas Lizenzen, war es integraler Bestandteil der Aufgabe, einen Vergleich zwischen Software-Werkzeugen vorzunehmen. Diese Werkzeuge sollen das (Entwickler-)Team bei der Arbeit mit Lizenzen unterstützen. Und somit als Bestandteil der DevOps-Pipeline dazu beitragen, die rechtlichen Anforderungen gewährleisten zu können.

3.1 Auswahl des geeignetsten Werkzeugs

Im Vortrag wurde ein Vergleich zwischen vier Werkzeugen vorgenommen: Die „Open Source Security Platform“ *Snyk* [24], das „Open Source License Compliance Management Tool“ *Fossa* [25], das in der Versionskontrolle *GitLab* integrierte Tool *License Compliance* [26] und der *Open-Source Ansatz*, einem Adapter für die Integration bereits existierender Lizenz-Überprüfungs-Werkzeuge in eine GitLab Pipeline [27].
Verglichen wurde nach mehreren Parametern. Eine tabellarische Übersicht ist auf Seite 19 gegeben. Die Informationen auf dieser Tabelle stammen allesamt aus den Quellen [24] [25] und [26]. Der Open-Source Adapter wurde speziell für dieses Seminar geschrieben. Er nutzt dabei zwei Open-Source Pakete [28] und [29].
Die Entscheidung, welches der Werkzeuge das geeignetste ist, häng stark von den Gegebenheiten in der betrachteten Organisation ab. Für große Firmen sind die eigenständigen Plattformen Fossa und Snyk vermutlich die beste Wahl. Ab einer bestimmten Unternehmensgröße fallen einige tausend Dollar Kosten pro Monat wenig ins Gewicht. Diese werden schnell durch die Vorteile, wie hohe Regelkomplexität, Vorkonfiguration und Support aufgewogen. Snyk bietet die Lizenz-Analyse nur in den höheren Bezahlstufen an, verlangt also für alle für diesen Zweck geeigneten Anwendungsformen einen von der Entwicklerzahl abhängigen Preis. Da die Lizenz-Analyse nur Nebenfeature zu sein scheint, rentiert sich Snyk vermutlich nicht für den Zweck der Lizenzverwaltung allein, bietet aber ein starkes Ökosystem mit mehreren Werkzeugen, sollte das Angebot möglicherweise schon zu anderen Zwecken wahrgenommen werden.
Fossas Angebot bringt den Vorteil einer fähigen kostenfreien Version. Durch den Verzicht auf u.a. Verwaltungsoperationen bekommt man Zugriff auf einen Lizenz-Scanner mit hoher Konfigurierbarkeit und hochwertigen Voreinstellungen. Auch die Darstellung in der eigenen GUI

ist übersichtlich. Das Angebot die Werkzeuge lokal auf den Entwicklermaschinen oder eigenen Servern laufen zu lassen und nur die Ergebnisse in das Dashboard hochzuladen ermöglicht es, Analysen vorzunehmen ohne Fossa Zugriff auf den Code zu gegeben. Sollte man sich für die kostenpflichtige Version entscheiden ist diese allerdings teurer, als die *Team*-Stufe von Snyk.
Beide Plattformen lassen sich schnell mit Projekten aus einem GitLab-Repository betreiben, jedoch erfordert die Rückintegration in die GitLab Oberfläche auf den ersten Blick einen höheren Aufwand.
Mit genau diesem Argument besticht das integrierte Werkzeug von GitLab. Für Organisationen, die bereits stark auf GitLab-Pipelines setzen, ist das Aufsetzten dieses Analysetools mit wenigen Schritten getan und es ist vollständig in GitLabs Oberfläche integriert. Die Analyse ist solide, jedoch weniger stark konfigurierbar. Da in den meisten Fällen die Lizenz-Analyse allerdings ohnehin von - im Thema Recht geschulten - Mitarbeitern überprüft werden muss, ist es mitunter schon ausreichend, wenn das Werkzeug die Lizenzen auflisten kann und Alarm schlägt, wenn ein einfacher Regelfilter durch einen Änderung verletzt wird. Beides kann GitLabs internes Werkzeug problemlos liefern. Für sehr kleine Firmen oder Projekte ist schade, dass die vielseitigen Analysefunktionen nur in der *Ultimate*-Version von GitLab zur Verfügung stehen. Diese ist für sich gesehen teurer, als sowohl die Angebote von Snyk als auch die von Fossa. Jedoch erlangt man durch die *Ultimate*-Version Zugriff auf das komplette Angebot von Premium GitLab Funktionen. Dies ist für Firmen ab mittlerer Größe ohnehin beinahe unerlässlich, wenn GitLab als Hauptversionskontrollsystem genutzt wird.
Zuletzt soll mit der Open-Source Variante ein direkter Ersatz für das GitLab-Werkzeug vorgestellt werden. Beinahe jeder Paketmanager bietet Lizenz-Analyse direkt oder über ein Open-Source Paket an. Diese sind allerdings nicht einer Pipeline Struktur automatisiert. Die wenigsten dieser Pakete formatieren leider ihre Ausgabe so, dass sie von GitLab analysiert werden kann. Mit Hilfe kurzer Shell-Skripte, lässt sich die Ausgabe der meisten Werkzeuge allerdings schnell in das gängige JUnit-Format [30] übersetzen. Dies integriert die Fähigkeiten der Open-Source Werkzeuge direkt in die Oberfläche von GitLab und bietet dazu das Überprüfen von einfachen *Erlaubt-* oder *Verboten-Filtern*.
Zusammengefasst lässt sich sagen: Kleine Unternehmen können mit etwas Aufwand für ihre Projekte die kostenlose Variante zum Einsatz bringen. Die anderen Werkzeuge rentieren sich für einzelne Projekte selten. Besteht bereits ein GitLab-Ultimate Zugang, so bietet dieser die schnellste und einfachste Lizenz-Analyse mit der besten Integration. Ist man Teil einer großen Organisation hohen Entwicklerzahlen und vielen Projekten, so werden die Angebote von Fossa oder Snyk irgendwann unerlässlich, um die Qualität der produzierten Software garantieren zu können.

3.2 Inbetriebnahme

Der finale Abschnitt soll eine Kurzanleitung für die Integration gängiger Open-Source Lizenz-Analyse-Werkzeuge in eine bestehende GitLab Pipeline sein.
Die Skripte zusammen mit einer Kurzdokumentation auf Englisch findet sich auch unter: [27].

Für die Verwendung der Methode bedarf es einer funktionierenden GitLab Pipeline, mit entsprechenden *Runnern*. Dies soll allerdings hier nicht näher erläutert werden und wird vorausgesetzt. Für jeden verwendeten Paketmanager wird ein eigenes Skript benötigt. Die Skripte werden in einem Ordner mit dem Namen *license-checker* im untersten Projektordner platziert und richtig benannt. Vorkonfigurierte Skripte findet man im Anhang. Für den Paketmanager *npm* auf Seite 16 und für den Paketmanager *composer* auf Seite 17.
Damit die Skripte in einer Pipeline richtig ausgeführt werden, muss ein entsprechender *i*n der Pipeline definiert werden. Vorkonfigurierte Beispiele für Jobs, die die Skripte ausführen und die Ergebnisse an GitLab zurückgeben, findet man auf Seite Seite 15 im Anhang.
In der Zeile *script* müssen dann lediglich alle Lizenzen, die erlaubt werden sollen, mit Semikolon getrennt zwischen die Hochkommata geschrieben werden.

4 Literaturverzeichnis

[1] Forcepoint, *What Is DevSecOps? Defined, Explained, and Explored | Forcepoint*, https://www.forcepoint.com/de/cyber-edu/devsecops (besucht am 13. 09. 2021).

[2] AWS, *Was ist DevOps? - Amazon Web Services (AWS)*, https://aws.amazon.com/de/devops/what-is-devops/ (besucht am 13. 09. 2021).

[3] Haufe-Lexware GmbH & Co KG, *Agile Methoden: Definition und Überblick*, Agile Methoden und Techniken im Überblick, https://www.haufe.de/personal/hr-management/agile-methoden-definition-und-ueberblick_80_428832.html (besucht am 13. 09. 2021).

[4] S. Frömling, *Agile Methoden: Was Scrum von Kanban unterscheidet*, (1. Juli 2021) https://www.computerwoche.de/a/was-scrum-von-kanban-unterscheidet,3548315 (besucht am 13. 09. 2021).

[5] Redhead, *Was ist DevSecOps?*, https://www.redhat.com/de/topics/devops/what-is-devsecops (besucht am 13. 09. 2021).

[6] E. Kinsbruner, *How to Make CI, CT and CD Work Together and Avoid the Drama of a DevOps Love Triangle*, (17. Apr. 2018) https://www.itproportal.com/features/how-to-make-ci-ct-and-cd-work-together-and-avoid-the-drama-of-a-devops-love-triangle/ (besucht am 13. 09. 2021).

[7] C. Kölbl, *Softwareentwicklung mit Dev(Sec)Ops*, 2021.

[8] Tutorialspoint, *SDLC - Waterfall Model*, https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (besucht am 13. 09. 2021).

[9] B. Son, *A Beginner's Guide to Building DevOps Pipelines with Open Source Tools*, (8. Apr. 2019) https://opensource.com/article/19/4/devops-pipeline (besucht am 16. 09. 2021).

[10] XebiaLabs, *XebiaLabs Präsentiert Das „Periodensystem Der DevOps-Tools V.3"*, (26. Juni 2018) https://www.businesswire.com/news/home/20180625006332/de/ (besucht am 16. 09. 2021).

[11] digital.ai, *Periodic Table of DevOps Tools*, https://digital.ai/periodic-table-of-devops-tools (besucht am 16. 09. 2021).

[12] G. Guimarães, *What Is a Linter and Why Your Team Should Use It?*, (3. Juli 2020) https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it (besucht am 16. 09. 2021).

[13] P. Gafert, *ArchUnit*, Unit test your Java architecture, https://www.archunit.org/ (besucht am 16. 09. 2021).

[14] *Conventional Commits*, https://www.conventionalcommits.org/en/v1.0.0/ (besucht am 16. 09. 2021).

[15] T. Preston-Werner, *Semantic Versioning 2.0.0*, https://semver.org/ (besucht am 16. 09. 2021).

[16] Snyk, *Open Source Security Management | Snyk*, https://snyk.io/product/open-source-security-management/ (besucht am 16. 09. 2021).

[17] Selenium, *Selenium Automates Browsers*. https://www.selenium.dev/ (besucht am 16. 09. 2021).

[18] GitLab, *Dynamic Application Security Testing (DAST) | GitLab*, https://docs.gitlab.com/ee/user/application_security/dast/ (besucht am 16. 09. 2021).

[19] CraftQuest, *What Are Zero Downtime Deployments?*, CraftQuest, https://craftquest.io/articles/what-are-zero-downtime-atomic-deployments (besucht am 16. 09. 2021).

[20] Sentry, *Application Monitoring and Error Tracking Software*, https://sentry.io/welcome/?utm_source=google&utm_medium=cpc&utm_campaign=10512507309&utm_content=g&utm_term=sentry&gclid=Cj0KCQjw1ouKBhC5ARIsAHXNMI9N-1vGu-8wVkL04qCvCck6zAOl2LioOQ4eyOYoTDJA6IxjqXsu95oaAkiVEALw_wcB (besucht am 16. 09. 2021).

[21] I. Nemytchenko, *GitLab CI: Run Jobs Sequentially, in Parallel or Build a Custom Pipeline*, (29. Juli 2016) https://about.gitlab.com/blog/2016/07/29/the-basics-of-gitlab-ci/#run-jobs-sequentially (besucht am 16. 09. 2021).

[22] Validatis, *Compliance: Definition & Bedeutung für Unternehmen*, https://www.validatis.de/kyc-prozess/news-fachwissen/compliance/ (besucht am 13. 09. 2021).

[23] Haufe-Lexware GmbH & Co KG, *Bedeutung von Compliance für Unternehmen*, Haufe.de News und Fachwissen, https://www.haufe.de/compliance/management-praxis/compliance/bedeutung-von-compliance-fuer-unternehmen_230130_474234.html (besucht am 13. 09. 2021).

[24] Snyk, *Licensing Compliance Management | Snyk*, https://snyk.io/product/open-source-license-compliance/ (besucht am 13. 09. 2021).

[25] Fossa, *Open Source License Compliance Management | FOSSA*, https://fossa.com//product/open-source-license-compliance (besucht am 13. 09. 2021).

[26] GitLab, *License Compliance | GitLab*, https://docs.gitlab.com/ee/user/compliance/license_compliance/ (besucht am 13. 09. 2021).

[27] J. Kell, *OpenSource License Checker Adapter*, (2021) https://github.com/jonas-kell/seminar-dev-ops-2021 (besucht am 13. 09. 2021).

[28] D. Glass, *NPM License Checker*, https://www.npmjs.com/package/license-checker (besucht am 13. 09. 2021).

[29] D. Bauernfeind, *Composer License Checker*, https://github.com/dominikb/composer-license-checker (besucht am 13. 09. 2021).

[30] IBM, *JUnit Standard*, https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/de/adfz/developer-for-zos/14.1.0?topic=formats-junit-xml-format (besucht am 13. 09. 2021).

Die Abbildung 2.1 wurde mit der Software Inkscape erstellt.

5 Anhänge

.gitlab-ci.yml

```yaml
check-npm-licenses:
  image: node:16
  script:
    - /bin/bash ./license-checker/npm-licenses.sh
      ↪ "MIT;ISC;Apache-2.0;0BSD;BSD-2-Clause;BSD-3-Clause;CC0-1.
      ↪ 0;CC-BY-4.0;Unlicense"
  artifacts:
    when: always
    paths:
      - npm_license_summary.txt
      - npm_license_detailed.json
    expire_in: 3 days
    reports:
      junit:
        - npm_licenses.xml

check-composer-licenses:
  image: composer:2.1.6
  script:
    - /bin/bash ./license-checker/composer-licenses.sh
      ↪ "MIT;LGPL-2.1;BSD-3-Clause;LGPL-3.0;Apache-2.0"
  artifacts:
    when: always
    paths:
      - composer_license_summary.txt
      - composer_license_detailed.txt
    expire_in: 3 days
    reports:
      junit:
        - composer_licenses.xml
```

license-checker/npm-licenses.sh

```
# install the npm dependency
npm install
npm install -g license-checker
npm install -g yui-lint

# list the summary of licenses used
license-checker --summary --out "npm_license_summary.txt"
# generate a overview of licenses
license-checker --json --out "npm_license_detailed.json"


fails=0
# Generate Output.xml
echo '<?xml version="1.0" encoding="UTF-8"?><testsuites><testsuite
 ↪  id="NPM-CUSTOM-LICENSE-CHECKER">' > npm_licenses.xml
echo '<testcase name="NPM check allowed Licenses">' >>
 ↪  npm_licenses.xml

# execute test and capture error stream in variable
ERROR=$(license-checker --onlyAllow "$1" 2>&1 >/dev/null)

if [ $? != 0 ];
then
    let "fails++"
    echo '<failure type="FAILURE">' >> npm_licenses.xml
    echo $ERROR >> npm_licenses.xml
    echo '</failure>' >> npm_licenses.xml
else
    echo '<passed type="PASSED"></passed>' >> npm_licenses.xml
fi

echo '</testcase>' >> npm_licenses.xml
echo '</testsuite></testsuites>' >> npm_licenses.xml

# 0 if all succeeded, larger 0 otherwise
exit $fails
```

license-checker/composer-licenses.sh

```
# install the composer dependency
composer require dominikb/composer-license-checker

# list the summary of licenses used
./vendor/bin/composer-license-checker report > "composer_license_summary.txt"

# generate a overview of licenses
# as there's no real option for this, run a check that lists everything and remove the Error-messages
./vendor/bin/composer-license-checker check --allowlist NAMEOFNOLICENSE >  "composer_license_detailed.txt"
sed -i '/\[ERROR\]/d' composer_license_detailed.txt

# format the first operand correctly to be used in the command
OPTIONS=""
IFS=';' read -ra ADDR <<< "$1"
for i in "${ADDR[@]}"; do
  OPTIONS="$OPTIONS --allowlist $i"
done


fails=0
# Generate Output.xml
echo '<?xml version="1.0" encoding="UTF-8"?><testsuites><testsuite id="COMPOSER-CUSTOM-LICENSE-CHECKER">' > composer_licenses.xml
echo '<testcase name="Composer check allowed Licenses">' >> composer_licenses.xml

# execute test and capture output stream in variable
OUTPUT=$(./vendor/bin/composer-license-checker check $OPTIONS)

if [ $? != 0 ];
then
    let "fails++"
    echo '<failure type="FAILURE">' >> composer_licenses.xml
    echo $OUTPUT >> composer_licenses.xml
    echo '</failure>' >> composer_licenses.xml
else
```

```
35      echo '<passed type="PASSED"></passed>' >> composer_licenses.xml
36  fi
37
38  echo '</testcase>' >> composer_licenses.xml
39  echo '</testsuite></testsuites>' >> composer_licenses.xml
40
41  # 0 if all succeeded, larger 0 otherwise
42  exit $fails
```

Tabelle: Vergleich Lizenzwerkzeuge

Tool	SNYK	FOSSA	GitLab	OpenSource
Preis Version:	**$195 /5 Entwickler/Monat** (Team)	**$230 /5 Entwickler/Monat (Bereits Fähige Free Version)**	**$99 /Entwickler/Monat** (Ultimate)	**Gratis** (Developer Kosten)
Installations Aufwand	**Mittel** (gibt Snyk einen Application Access Token)	**Mittel** (quick-Import per Token, oder lokales Tool (Aufwand))	**Sehr Niedrig** („Out of the box“ bei bestehendem GitLab)	**Hoch** (Muss eigens geschrieben werden)
Konfiguration s Aufwand	Einfache Regeln in Team Komplexe in teurerer Version	Vorkurierte Lizenz-Regel Vorlagen	**Hoch** (Einmalig, muss Eigene Regelfestlegung pro Lizenz)	**Hoch** (Einmalig, Muss Eigene Regelfestlegung pro Lizenz)
Unterstützte Sprachen	8 (Alle benötigten)	17 ++ (Alle benötigten)	6 + 9 experimentell (Alle benötigten)	Praktisch Alle, aber jeweils extra Aufwand
Statische Checks	Ja + IDE-Integration	Ja, lokales CLI Tool	Nein aber Übersicht	Ja
Checks bei Merges	Ja + detaillierte Verbesserungsvorschläge	Ja + detaillierte Verbesserungsvorschläge	Ja + Übersicht über Violation	Ja + etwas Übersicht über Violation
Deployment	SaaS, (SelfHost in High Tier)	Tool lokal -> SaaS (nur Infos), Oder On-Prem -> SaaS	SaaS oder SelfHost	Zwangsweise in der gleichen Konfiguration wie GitLab
Dependency Manager Integration	Integriert direkt mit GitLab	Integration für Code Import und Webhooks Export	Ist der Dependency Manager	Ja, Bleibt hinter der normalen GitLab Version zurück
Regel komplex.	Hoch	Hoch	Niedrig	Mittel (frei, aber aufwändig)
Dashboard	Sehr Detailliert	Sehr Detailliert	Ja, aber wenig Details	Nein