

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: 13. September 2021

Inhaltsverzeichnis

1	Diskussion: Dev(Sec)Ops	1
1.1	Warum Dev(Sec)Ops	1
1.2	Für wen ist Dev(Sec)Ops?	2
2	Der „ideale“ Dev(Sec)Ops Prozess	3
2.1	Welche Werkzeuge kommen zum Einsatz?	3
2.2	Wie ordnet man seine Pipeline am besten an?	3
3	Dev(Sec)Ops Werkzeug: License Checker	4
3.1	Auswahl des geeignetsten Werkzeugs	4
3.2	Inbetriebnahme	5
4	Literaturverzeichnis	7
5	Anhänge	9

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 in dem 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 der Teams gesteigert und sie bleiben 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 erraten können: „Für JEDEN!“. Die ausführlichere, dafür aber auch zufriedenstellendere Antwort ist komplizierter.

2 Der „ideale“ Dev(Sec)Ops Prozess

2.1 Welche Werkzeuge kommen zum Einsatz?

2.2 Wie ordnet man seine Pipeline am besten an?

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 [8] [9].

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* [10], das „Open Source License Compliance Management Tool“ *Fossa* [11], das in der Versionskontrolle *GitLab* integrierte Tool *License Compliance* [12] und der *Open-Source Ansatz*, einem Adapter für die Integration bereits existierender Lizenz-Überprüfungswerkzeuge in eine *GitLab Pipeline* [13].

Verglichen wurde nach mehreren Parametern. Eine tabellarische Übersicht ist auf [Seite 13](#) gegeben. Die Informationen auf dieser Tabelle stammen allesamt aus den Quellen [10] [11] und [12]. Der Open-Source Adapter wurde speziell für dieses Seminar geschrieben. Er nutzt dabei zwei Open-Source Pakete [14] und [15].

Die Entscheidung, welches der Werkzeuge das geeignetste ist, hängt 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. Die Möglichkeit 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 direkt 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 Aufsetzen 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 das GitLab interne 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 und 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 mit einem Open-Source Paket. Diese sind lediglich 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 [16] 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-Analysewerkzeuge in eine bestehende GitLab Pipeline sein.

Die Skripte zusammen mit einer Kurzdokumentation auf Englisch findet sich auch unter: [\[13\]](#). 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 zudem im Anhang. Für den Paketmanager *npm* auf [Seite 10](#) und für den Paketmanager *composer* auf [Seite 11](#).

Damit die Skripte in einer Pipeline richtig ausgeführt werden, muss ein entsprechender Job in der Pipeline definiert werden. Vorkonfigurierte Beispiele für Jobs, die die Skripte ausführen und die Ergebnisse an GitLab zurückgeben findet man auf [Seite 9](#) 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)*, Amazon Web Services, Inc., <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] Redhat, *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*, ITProPortal, (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] Validatis, *Compliance: Definition & Bedeutung für Unternehmen*, Validatis, <https://www.validatis.de/kyc-prozess/news-fachwissen/compliance/> (besucht am 13.09.2021).
- [9] 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).
- [10] Snyk, *Licensing Compliance Management* | Snyk, <https://snyk.io/product/open-source-license-compliance/> (besucht am 13.09.2021).
- [11] Fossa, *Open Source License Compliance Management* | FOSSA, <https://fossa.com/product/open-source-license-compliance> (besucht am 13.09.2021).
- [12] GitLab, *License Compliance* | GitLab, https://docs.gitlab.com/ee/user/compliance/license_compliance/ (besucht am 13.09.2021).

- [13] J. Kell, *OpenSource License Checker Adapter*, (2021) <https://github.com/jonas-kell/seminar-dev-ops-2021> (besucht am 13.09.2021).
- [14] D. Glass, *NPM License Checker*, npm, <https://www.npmjs.com/package/license-checker> (besucht am 13.09.2021).
- [15] D. Bauernfeind, *Composer License Checker*, <https://github.com/dominikb/composer-license-checker> (besucht am 13.09.2021).
- [16] 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).

5 Anhänge

.gitlab-ci.yml

```
1  check-npm-licenses:
2    image: node:16
3    script:
4      - /bin/bash ./license-checker/npm-licenses.sh
5        ↪ "MIT;ISC;Apache-2.0;0BSD;BSD-2-Clause;BSD-3-Clause;CC0-1.
6        ↪ 0;CC-BY-4.0;Unlicense"
7    artifacts:
8      when: always
9      paths:
10        - npm_license_summary.txt
11        - npm_license_detailed.json
12      expire_in: 3 days
13      reports:
14        junit:
15          - npm_licenses.xml
16
17  check-composer-licenses:
18    image: composer:2.1.6
19    script:
20      - /bin/bash ./license-checker/composer-licenses.sh
21        ↪ "MIT;LGPL-2.1;BSD-3-Clause;LGPL-3.0;Apache-2.0"
22    artifacts:
23      when: always
24      paths:
25        - composer_license_summary.txt
26        - composer_license_detailed.txt
27      expire_in: 3 days
28      reports:
29        junit:
30          - composer_licenses.xml
```

license-checker/npm-licenses.sh

```
1  # install the npm dependency
2  npm install
3  npm install -g license-checker
4  npm install -g yui-lint
5
6  # list the summary of licenses used
7  license-checker --summary --out "npm_license_summary.txt"
8  # generate a overview of licenses
9  license-checker --json --out "npm_license_detailed.json"
10
11
12  fails=0
13  # Generate Output.xml
14  echo '<?xml version="1.0" encoding="UTF-8"?><testsuites><testsuite
   ↳ id="NPM-CUSTOM-LICENSE-CHECKER">' > npm_licenses.xml
15  echo '<testcase name="NPM check allowed Licenses">' >>
   ↳ npm_licenses.xml
16
17  # execute test and capture error stream in variable
18  ERROR=$(license-checker --onlyAllow "$1" 2>&1 >/dev/null)
19
20  if [ $? != 0 ];
21  then
22      let "fails++"
23      echo '<failure type="FAILURE">' >> npm_licenses.xml
24      echo $ERROR >> npm_licenses.xml
25      echo '</failure>' >> npm_licenses.xml
26  else
27      echo '<passed type="PASSED"></passed>' >> npm_licenses.xml
28  fi
29
30  echo '</testcase>' >> npm_licenses.xml
31  echo '</testsuite></testsuites>' >> npm_licenses.xml
32
33  # 0 if all succeeded, larger 0 otherwise
34  exit $fails
```

license-checker/composer-licenses.sh

```

1  # install the composer dependency
2  composer require dominikb/composer-license-checker
3
4  # list the summary of licenses used
5  ./vendor/bin/composer-license-checker report >
   ↪  "composer_license_summary.txt"
6
7  # generate a overview of licenses
8  # as there's no real option for this, run a check that lists
   ↪  everything and remove the Error-messages
9  ./vendor/bin/composer-license-checker check --allowlist
   ↪  NAMEOFNOLICENSE > "composer_license_detailed.txt"
10 sed -i '/\[ERROR\]/d' composer_license_detailed.txt
11
12 # format the first operand correctly to be used in the command
13 OPTIONS=""
14 IFS=';' read -ra ADDR <<< "$1"
15 for i in "${ADDR[@]}; do
16     OPTIONS="$OPTIONS --allowlist $i"
17 done
18
19
20 fails=0
21 # Generate Output.xml
22 echo '<?xml version="1.0" encoding="UTF-8"?><testsuites><testsuite
   ↪  id="COMPOSER-CUSTOM-LICENSE-CHECKER">' > composer_licenses.xml
23 echo '<testcase name="Composer check allowed Licenses">' >>
   ↪  composer_licenses.xml
24
25 # execute test and capture output stream in variable
26 OUTPUT=$(./vendor/bin/composer-license-checker check $OPTIONS)
27
28 if [ $? != 0 ];
29 then
30     let "fails++"
31     echo '<failure type="FAILURE">' >> composer_licenses.xml
32     echo $OUTPUT >> composer_licenses.xml
33     echo '</failure>' >> composer_licenses.xml
34 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)
Konfigurations 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