consol
Wir unternehmen IT

Ein Leben nach Jenkins. Neue CI/CD-Tools für die Cloud.

ConSol Leitfaden

INHALT

Einführung. Warum überhaupt CI/CD?

Die fortschreitende Digitalisierung ruft nach immer schnellerer und effizienterer Softwareentwicklung. Code soll zügiger geschrieben, integriert, getestet, bereitgestellt und ausgeliefert werden. Kurze Release-Zyklen für Apps oder neue Features sind für einen Großteil der Unternehmen kaum verzichtbar, wenn sie im Wettbewerb stand halten und gleichzeitig Kosten sparen wollen.

Der Einsatz der Methoden **Continuous Integration (CI) und Continuous Delivery (CD)** ist daher mittlerweile Standard im IT-Development, insbesondere weil sie für die kontinuierliche Automatisierung und Überwachung des Entwicklungsprozesses sorgen. Kommt noch ein agiles DevOps-Umfeld hinzu, sind dies gute Voraussetzungen für die Produktivitätssteigerung.

Bei CI befindet sich eine Software oder eine Applikation im Teststadium. Entwickler erzeugen oder ändern Code und integrieren ihn (bestenfalls mehrmals täglich) in den Gesamtcode. Sowohl bei jedem Änderungs-Commit als auch bei der Integration laufen automatisierte Tests, die das Konstrukt damit validieren und auf Funktionsfähigkeit überprüfen.

Den nächsten logischen Schritt stellt **Continuous Delivery** dar, bei dem das Operations-Team mehr in den Fokus rückt. Denn nach dem erfolgreichen CI-Prozess geht es nun darum, die bisherigen Ergebnisse in eine produktionstaugliche Form zu gießen – heißt also, eine auslieferbare Software bereitzustellen. Auch dieser Bereitstellungsprozess läuft automatisiert. Wird diese Software nicht nur bereitgestellt, sondern auch automatisch auf der Zielplattform installiert, so spricht man von Continuous Deployment.

Neue Anforderungen durch Cloud-native

Nach dem Umzug in die Cloud – mit dem wir beispielsweise ad-hoc Verfügbarkeit von skalierenden Rechenressourcen verbinden – birgt nun der Wechsel zu Cloud-native einige Herausforderungen. Die Cloud Native Computing Foundation (CNCF) definiert Cloud-native so:

“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.”

Cloud-native Applikationen basieren heute verstärkt auf Microservices. Diese modernen Architekturen haben eine direkte Auswirkung auf CI/CD. Denn während man zuvor etwa den Release eines Monolithen zu

verantworten hatte, sind nun immer mehr Pipelines notwendig, um Dutzende von Microservices releasefähig zu machen. Zudem werden die Pipelines selbst immer komplexer, um die Möglichkeiten der Cloud zu nutzen. Sie sind inzwischen selbst als Software-Quellcode zu betrachten und behandeln, mit allen Konsequenzen. Was dieses Beispiel aufzeigen soll: CI/CD Tools und Workflows müssen sich an die Geschwindigkeit sowie die Komplexität der cloud-nativen Möglichkeiten anpassen.

Es verwundert also nicht, dass „klassische“ CI/CD-Tools mit Ihren Funktionalitäten an Ihre Grenzen stoßen. Jenkins, eines dieser Tools, hat nach verbreiteter Meinung wohl seine besten Tage gesehen. Ein Entwicklerteam um Jenkins-Erfinder Kohsuke Kawaguchi und CloudBees CEO Sacha Labourey hat diesem Fakt auf der europäischen DevOps World 2018 zwar den Kampf angesagt: Jenkins hätte in Form des Nachfolgeproduktes Jenkins X nun „Superkräfte“ erhalten, um modernen Anforderungen zu genügen. Doch es melden sich inzwischen junge, vielversprechende Tools zu Wort, die direkt aus der cloud-nativen Entwicklung heraus entstanden sind und keine Altlasten mit sich tragen.

Erfahren Sie mehr über neue CI/CD-Tools für die Cloud in den folgenden Kapiteln.

Ein Leben nach Jenkins. Neue CI/CD-Tools für die Cloud.

Jenkins ist der Platzhirsch unter den CI/CD-Lösungen, hat aber mit der Weiterentwicklung der IT-Landschaft in Richtung Cloud wohl seine besten Jahre hinter sich. Neue Tools drängen auf den Markt, welche Technologien wie Kubernetes nicht nur als Option sondern als native Plattform begreifen und daraus immensen Nutzen ziehen. In diesem Artikel machen wir uns Gedanken über zeitgemäße Kriterien für eine Continuous Delivery Plattform auf Kubernetes und beleuchten passende Lösungen.

In die Jahre gekommen

Jenkins genießt in vielen Software-Schmieden den Status eines "De-Facto-Standards" für Continuous Integration, Delivery und Deployment. Das kommt nicht von ungefähr, war die Plattform doch dank einer offenen Plugin-Architektur für lange Zeit flexibel genug, um an geänderte Anforderungen angepasst zu werden. Die Anfänge des Projektes gehen weit in die IT-Historie zurück. Das erste Release fand bereits im Jahr 2005 unter Federführung von Sun Microsystems statt.

Daher ist Jenkins auf modernere Cloud-Plattformen wie Kubernetes eigentlich gut vorbereitet. Dank passendem Plugin lässt sich über Jenkins containerisiert in Kubernetes bauen, testen und deployen. Laut "Jenkins Community Survey 2018"[1] setzen ca. 47 Prozent der teilnehmenden Nutzer Jenkins zusammen mit Googles Cloud-Plattform ein. Hier scheint es also grundsätzlich kein Problem zu geben.

1 https://www.heise.de/developer/meldung/Jenkins-Community-Survey-Kubernetes-und-die-Cloud-machen-das-Rennen-4243181.html

Bei genauerem Hinsehen zeigen sich allerdings Schwächen dieser Kombination. Diese Schwächen haben nicht alle mit modernen Plattformen zu tun, sondern sind Design-Entscheidungen geschuldet, die aus heutiger Sicht ungünstig sind. Eventuell hat man sich inzwischen an sie gewöhnt, auch wenn sie hinderlich sind. Ein paar Beispiele:

- Das Jenkins-eigene Clustering-Modell um Pipelines auf Slave-Rechnern auszuführen ist auf Kubernetes obsolet. Das kann die Cloud-Plattform besser organisieren. Entsprechend wird dieser Ansatz vom Kubernetes-Plugin für Jenkins auch nicht mehr verfolgt. Stattdessen werden Pods ad-hoc für die Pipeline-Ausführung gestartet. Dennoch verbleibt Ballast, z.B. der noch aus dem Slave-Konzept stammende, recht komplizierte "Container-Contract", den Images erfüllen müssen, um so verwendet zu werden. Oder die ungewohnt passive Rolle des resultierenden Containers, der nur Plattform für die Pipeline ist.
- Pipeline-Skripte in Jenkins sind heutzutage oft komplizierte Gebilde, ebenso komplex wie normaler Programmcode. Sie basieren nicht selten auf wiederverwendbaren Pipeline-Modulen Marke Eigenbau. Die Entwicklung dieser Skripte und ihrer Module ist oft problematisch, da sie sich nur im Rahmen eines kompletten Pipeline-Laufs effektiv testen lassen, inklusive Checkout aus dem Code-Repository. Resultat ist häufig eine recht schlechte Qualität dieses Codes, dessen Funktionieren oft mit unverhältnismäßig hohem Testaufwand erkauft wurde. Entsprechend ungern passt man ihn später im laufenden Betrieb an.
- Die Jenkins-Plugins, so flexibel sie die Plattform machen, sind oft mehr Fluch als Segen: Gerade bei langgedienten Installationen zeichnen sie sich häufig durch grassierenden Wildwuchs aus. Den wenigsten Betreibern sind die genau notwendigen Typen und Versionen der Plugins bekannt, die von den eigenen Pipelines benötigt werden. Insofern ist der Plugin-Installationsstand oft ein Stück undokumentierte "Schatten-IT".

CI/CD neu denken auf Basis von Kubernetes

Würde man das Thema CI/CD aus cloud-nativer Perspektive noch einmal unvoreingenommen neu konzipieren, so würde das Ergebnis doch erheblich anders ausfallen als die Jenkins-Architektur. Tabelle 1 listet Vorschläge zu den Kriterien auf, die dabei wichtig sein sollten und die wir im Folgenden erläutern.

Container überall

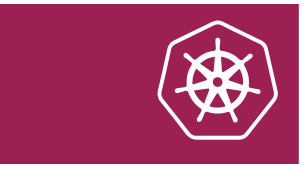
Dreh- und Angelpunkt dieses Konzeptes wären sicherlich moderne Container-Technologien, die sowohl im Testing als auch Deployment heute den Ton angeben. Auch in der Pipeline selbst ist der Container als Pipeline-Modul, bzw. das Image als seine Blaupause, die bessere Wahl gegenüber proprietären Pipeline-Programmiersprachen. Sind die Schritte der Pipeline einfach Linux-Container, so sind Entwickler sehr flexibel beim Einsatz der nutzbaren Funktionalitäten, inklusive aller Shells und Command Line Interfaces, die unter Linux verfügbar sind. Der Shell-Befehl "sh" ist in vielen Jenkins-Pipelines oft das meist genutzte Kommando. Warum also nicht direkt eine Shell für Pipeline-Befehle verwenden? So muss für die Entwicklung der Images keine neue, proprietäre

Sprache gelernt werden, vorausgesetzt die CI/CD-Plattform zwingt uns kein proprietäres Interfacing mit dem Container auf.

Richtig konzipiert ist der Pipeline-Container an beliebiger Stelle ausführbar und dabei nicht abhängig von einer Form des Inputs, z.B. einem Git-Repository unter einer bestimmten URL. Dabei ist darauf zu achten, dass lediglich container-native Interfaces wie Mounts, Umgebungsvariablen und Entrypoint-Commands verwendet werden, um mit dem Container zu kommunizieren. Wird das Git-Repository, mit dem gearbeitet werden soll, im CI/CD-Prozess einfach eingemounted, so ist es dem Container egal woher es stammt. Beim lokalen Container-Testing kann es beispielsweise ein lokal ausgechecktes Projekt von unserer Festplatte sein. Somit lassen sich die Container unkompliziert testen, ohne CI/CD-Umgebung.

Logischerweise ist das Building von Applikations-Images heutzutage eine wichtige Aufgabe einer Pipeline. Vor nicht allzu langer Zeit hätten wir hier eine integrierte Unterstützung des CI-Tools gefordert. Insbesondere hätten wir darauf bestanden, dass die Pipeline ohne Zugriff auf den Docker-Daemon eines Kubernetes-Hosts auskommt. Inzwischen gibt es z.B. mit Kaniko[2] und Buildah[3] gute Image-Building-Lösungen, die keinen Daemon benötigen und die wir genauso wie jedes andere Command Line Interface in Containern verwenden können. Insofern ist diese Funktionalität problemlos integrierbar, vorausgesetzt wir können mit beliebig ausgestatteten Containern arbeiten.

Kubernetes als Plattform und sonst nichts



Ein großes Plus von Jenkins ist sicherlich, dass es unproblematisch einsetzbar ist. Unser Nachfolge-Tool soll ähnlich unkompliziert verwendbar sein, ohne sich Gedanken über Kosten, Lizenzen und Nutzungsquantitäten machen zu müssen. Ebenso wenig möchten wir mit einem speziellen Public Cloud Provider oder einem anderen abhängigen Service verheiratet sein, wenn wir es einsetzen. Ein beliebiges Kubernetes-Cluster an einem beliebigen Ort soll ausreichen.

Hinsichtlich Scheduling von Pipelines benötigt unser Tool, wie bereits erwähnt, keine eigenen Funktionalitäten. sondern baut hier voll auf Kubernetes. Das Tool selbst soll natürlich ebenfalls hier zu betreiben sein. Dabei würden wir auch gerne die immer noch verbreitete Unsitte vermeiden, Container in Kubernetes zwingend unter root-Berechtigungen betreiben zu müssen, was hinsichtlich Sicherheit sehr bedenklich ist. Dieses Kriterium würde die Zahl verfügbarer Lösungen aktuell immer noch drastisch reduzieren, daher betrachten wir es hier (mit einigen Schmerzen) lediglich als "nice to have".

Als Pluspunkt betrachten wir die Verwendung von Custom Kubernetes Resources für die Verwaltung von Pipelines und ihrer Bestandteile. "Pipeline als Code" ist ein weiteres wichtiges Kriterium das als Custom Resource quasi bereits erfüllt ist da sich diese grundsätzlich über ein YAML-Format definieren lassen. Idealerweise kann diese Definition jedoch auch im Software-Projekt welches die Pipeline bauen soll eingebettet werden, wie wir das von Jenkins gewohnt sind. Dadurch kann das Entwicklungsteam die Pipeline unkompliziert weiterentwickeln und mit dem Projekt wachsen lassen.

Zwischen diesen beiden Wünschen, Custom Resources und Pipeline-Datei im Code gibt es einen gewissen Konflikt, sind das doch eigentlich zwei konkurrierende Orte an denen die Definition vorliegt. Man braucht zumindest eine Brückentechnologie, welche die Pipeline-Datei im Code als führendes System betrachtet und

2 https://github.com/GoogleContainerTools/kaniko
3 https://github.com/containers/buildah

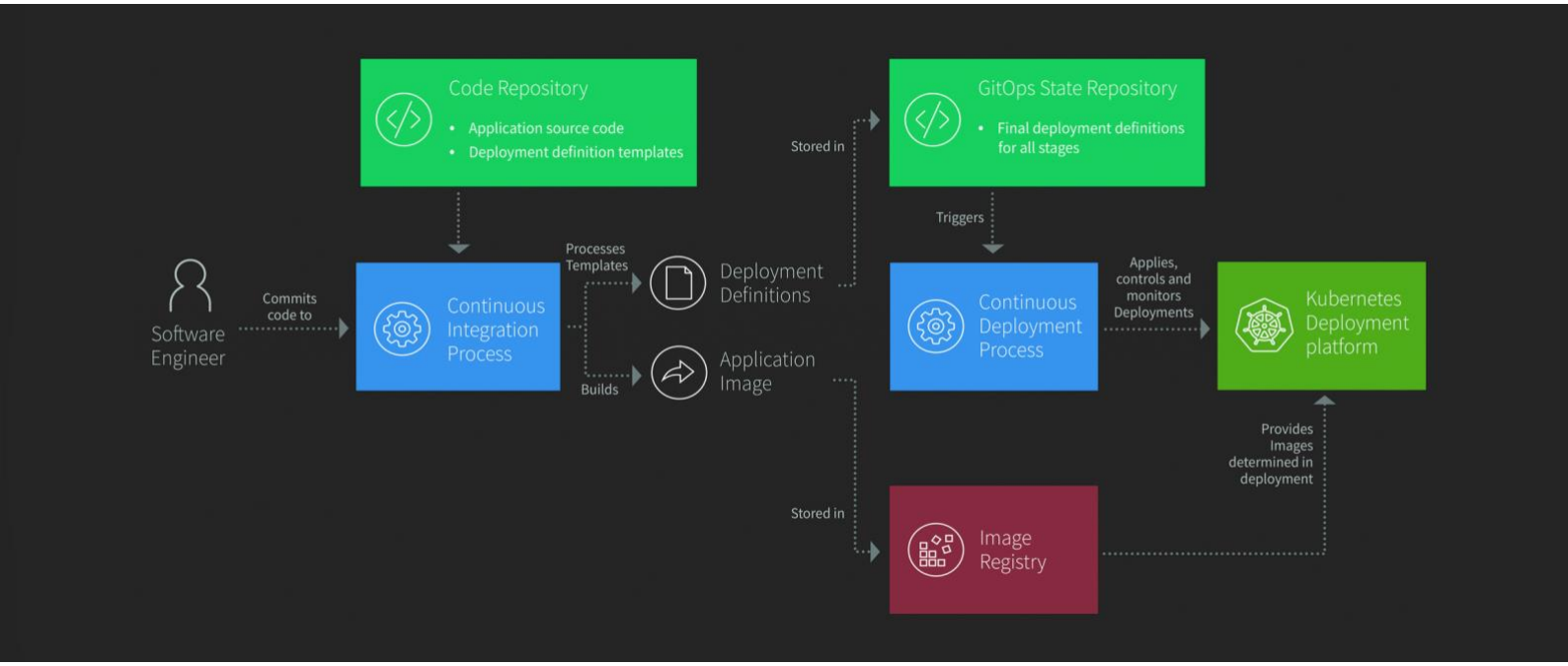
automatisiert in die Custom Resource überführt. Das bekäme man zwar mit vergleichsweise wenig Aufwand über Automationstools selbst hin. Eine direkte Integration dieser Funktionalität im CI-Tool wäre dennoch wünschenswert.

Idealerweise sind die Pipelines für individuelle Projekte dezentral in separaten Namespaces ausführbar. Denkt man nur an dedizierte Kubernetes-Cluster für ein einzelnes Projekt, erscheint dies unnötig. Es macht jedoch Sinn, sobald man große, mandantenfähige PaaS-Cluster ("Platform as a service") betrachtet, auf denen sich viele Mandanten mit vielen Projekten tummeln. Hier wird den Mandanten häufig eine Ressourcen-Limitierung (z.B. Hauptspeicher, CPUs) über Quotas auferlegt, welche auf ihren Namespaces automatisch enforciert wird. Es ist nur logisch, dass auch die Pipelines der Projekte Subjekt dieser Quotas sein sollten. Dazu müssen sie jedoch auch in einem Namespace des Mandanten laufen.

GitOps: CI und CD gehen auf Distanz

Bislang wurden sowohl Continuous Integration als auch Delivery bzw. Deployment oft über dasselbe Tool abgewickelt. Der Deployment-Teil hat sich hierbei seit dem Aufkommen von "Infrastruktur als Code" stark gewandelt. Anstelle das Deployment in Form von prozeduralen Installations-Routinen vorzunehmen, überträgt man hierbei eine Definition der zu erschaffenden Ziel-Architektur auf die Plattform. Diese Definition ist letztlich eine Art Code in einem deklarativen Format. Kubernetes unterstützt dabei direkt ein eigenes Definitionsformat als Eingabe, mit dem die Infrastruktur auf dem Cluster nicht nur definiert sondern auch geändert werden kann.

Bislang ist es oft noch Usus, diesen Definitionscode in einer kombinierten CI/CD-Pipeline aus Schablonen zu generieren und direkt auf die Ziel-Plattform zu übertragen. Eine neue Methodologie namens "GitOps", die zunehmend populär wird, möchte dies entkoppeln. So wird bei GitOps der generierte Definitionscode nicht direkt enforciert sondern zunächst in ein separates Git-Repository, das "State Repository" als neuer Commit übertragen. Ein separater Continuous Deployment Prozess greift nun diese Definitionen aus dem State Repository auf und überträgt sie nach definierten Mustern auf das jeweilige Deployment-Stage im Kubernetes-Cluster.

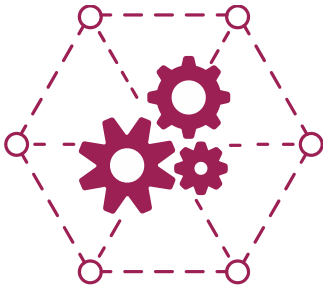


Als essentiellen Vorteil verspricht GitOps eine "Single Source of Truth" für das Operating in Form eben jenes State Repositories, das alle Stati der Infrastruktur in seiner History enthält. Updates laufen über Pull-Requests, Rollbacks sind mit Reverts realisierbar. Diese Herangehensweise hat noch einige weitere Vorteile, jedoch zu dem Preis, dass man nun zwei separate Prozesse hat. Der CD-Prozess wird dabei häufig über dedizierte Tools realisiert, welches das State-Repository überwacht und Änderungen eigentständig auf die Zielinfrastruktur überträgt. Beispiele sind Weaveworks Flux oder Argo CD.

Komplexere Deployment-Prozesse, z.B. Canary oder Blue/Green-Deployments können bei GitOps natürlich nicht mehr durch die CI-Pipeline mehr gesteuert werden da diese keine direkte Verbindung mehr zum Zielsystem hat. Dies wird stattdessen häufig über dedizierte Kubernetes-Operatoren für das Deployment, z.B. Weaveworks Flagger oder Argo Rollouts realisiert die wiederum über den GitOps-Prozess als Kubernetes Custom Resources gesteuert werden.

In diesem Artikel jedoch beschränken wir uns auf die traditionelle CI-Seite der Geschichte und auf ihre Tools. Die Verfügbarkeit einer Anbindung an GitOps-Prozesse, eventuell in einem separaten kompatiblen CD-Tool, sehen wir jedoch als Vorteil.

Neue Lösungen für die Cloud



Tatsächlich ist das Feld der passenden Lösungen trotz Zugeständnissen bei den Kriterien nicht allzu groß, weist aber schon einige interessante Contestants auf. Für unsere Auswahl vorgestellter Produkte haben wir uns für vergleichsweise junge Projekte entschieden. Entsprechend nah sind sie an den hier diskutierten Anforderungen und zeigen auf, wohin die Reise in Zukunft gehen mag. Eine Bewertung der Produkte anhand unserer Kriterien haben wir in *Tabelle **2**: Kriterien-Matrix der* zusammengefasst.

Argo

Das Argo-Projekt besteht im Kern aus der gleichnamigen Workflow-Engine, die explizit für Kubernetes entworfen wurde und daher unseren Kriterien grundsätzlich gut entspricht. Argo-Workflows bestehen im Kern aus einer Reihe von beliebigen Containern, die über die üblichen Container-Schnittstellen parametrisiert werden.
Die Bestandteile der Lösung, die Workflow-Engine selbst und das "Argo Events" getaufte Event-Subprojekt, sind sehr schnell installiert. Im Wesentlichen besteht die Installation aus 4 Controller-Pods und dem UI-Pod, die somit als leichtgewichtig durchgeht. Argo kann problemlos dezentral pro Projekt installiert werden.

Grundsätzlich merkt man dem Projekt an, dass es sich generisch auf Workflows spezialisiert und dass es nicht speziell auf Continous Integration zugeschnitten ist. Dennoch kann man mit etwas Arbeit aus den Argo-Komponenten eine effektive und vor allem flexible CI-Pipeline bauen. Dazu existieren diverse Beispiele in den Repositories des Herstellers. Zum Beispiel kann man via Argo Events eine Webhook-URL einrichten welche vom Code-Repository bei Änderungen am Code aufgerufen wird. Bei Aufruf startet diese einen Argo-Workflow als Pipeline, dessen Definition aus dem Git-Repository zu laden ist. Das Repository wird im Payload des Aufrufs angegeben. So ergibt sich letztlich eine "Pipeline as code"-Funktionalität.

Diese Beziehungen zwischen Hook, Repository und Workflow definiert man freilich manuell. Dies bedeutet zwar Aufwand, dafür lässt sich vieles beliebig feinjustieren. So findet man bei Argo auch eine Reihe speziellerer Features zur Workflow-Steuerung, etwa eine flexible Mutex- bzw- Semaphoren-Unterstützung oder die Organisation von Abhängigkeiten, die man in anderen, eher konkret auf CI/CD ausgerichteten, Produkten eventuell nicht oder nur eingeschränkt zur Verfügung hat.

Für die Konfiguration sowohl der Workflows als auch der Event-Handler verwendet Argo ausschließlich Kubernetes-Objekte und Custom Resources. Das funktionale Web-Dashboard informiert über den aktuellen Stand der Workflows. Ein Command-Line-Client für das Definieren und Starten der Pipelines und ein separates Tool namens "Argo CD" für GitOps-kompatibles Delivery, sowie "Argo Rollouts" für komplexere Release-Prozesse runden das Angebot ab.

InfraBox

InfraBox von der SAP SE wird als "Cloud Native Continuous Integration System" positioniert, und setzt als solches ebenfalls auf Kubernetes auf. Weder der plattform-unabhängige Installations-Prozess noch das resultierende System mit zwölf Pods und zwei externen Abhängigkeiten, ein S3-Object-Storage und eine PostgreSQL-Datenbank, können als leichtgewichtig bezeichnet werden.

Dafür ist es wohl das Tool mit der breitesten CI/CD-Feature-Palette. Es bringt bereits eine eigene Docker-Registry mit. Laut Dokumentation werden viele Einsatzszenarios unterstützt, u.a. auch eine Multi-Cluster-Installation, die auch mehrere Cloud Provider überspannen kann. Es besitzt sogar eine integrierte Unterstützung für die Provisionierung von End-2-End-Testumgebungen. Wer etwas mehr Komfort und schlüsselfertige Funktionalitäten wünscht, sollte sich InfraBox definitiv genauer anschauen.

Auch InfraBox setzt auf Custom Resource Definitions, um Pipelines, Funktionen und die Ausführung dieser Artefakte zu verwalten. Die Pipelines laufen in einem dedizierten Namespace "infrabox-worker", was eher

schlecht zu dezentralen Pipelines passt. Definiert werden die Pipelines in einer Datei "infrabox.json" im Quellcode des zu bauenden Projektes.

Die Pipeline-Schritte nennt Infrabox "Jobs". Es gibt mehrere Job-Typen mit vorgefertigten Funktionen, z.B. um Repositories auszuchecken, oder Images in die Registry zu übertragen. Es existiert aber auch ein generischer Job-Typ "docker-image", der beliebige Images ausführen kann und uns hier speziell wichtig ist.

Erwähnt werden sollte noch das Web-UI, welches vergleichsweise komfortabel ist und die Möglichkeit benutzerdefinierter Darstellungen bietet, sowie das Command Line Interface "infraboxcli", welches den Entwickler beim lokalen Debugging von Jobs unterstützt.

Ein wenig Sorgen bereitet uns die Tatsache, dass das letzte offizielle Release von InfraBox bereits knappe 2 Jahre alt ist, was in diesem progressiven Technologiefeld eine halbe Ewigkeit ist. Die Nutzungsdaten des Githubs-Repositories verraten uns, dass es dennoch aktuelle Entwicklungs-Aktivität gibt, wenn auch in geringem Umfang.

Tekton

Tekton ist ein sehr junges Projekt. Ursprünglich ging es aus Googles "Knative"-Produkt hervor, einer standardisierten Plattform für Serverless-Applikationen auf Kubernetes. Tekton ist der extrahierte Kern der "Build"-Komponente von Knative. Das Projekt wird von der "Continuous Delivery Foundation"[4] verwaltet, welche nach eigenen Aussagen der Fragmentierung der CI/CD-Landschaft entgegen treten will[5]. Neben Google und anderen Unternehmen ist hier auch CloudBees Mitglied. Der Jenkins-Hersteller plant, Tekton als alternative Engine seiner eigenen cloud-nativen CI/CD-Lösung "Jenkins X" zu nutzen die wir hier separat betrachten.

Als dedizierte cloud-native Lösung macht Tekton in unseren Augen einiges richtig. Es basiert vollkommen auf Kubernetes-Scheduling. Pipelines werden als einzelne Pods mit mehreren Containern als Build-Schritte ausgeführt, die unter voller Kontrolle der Pipeline-Definition sind. Das Interfacing ist "sauber", d.h. es gibt keine proprietären Kommunikationswege zwischen CI und Container. Als Image-Build-Tool wird "Kaniko" propagiert. Die Installation ist leichtgewichtig, es gibt 2 zentrale Verwaltungs-Pods. Die Pipelines selbst werden als Custom Resources in den jeweiligen Projekten definiert und dort auch ausgeführt. Insofern lesen sich die Specs von Tekton weitestgehend wie eine genaue Entsprechung der Kriterien dieses Artikels.

Seit wir uns das Projekt das erste Mal in diesem Rahmen angesehen haben ist einige Zeit vergangen, welche das Projekt zur Schärfung seiner Schnittstellen-Definitionen und allgemeinen Reifung der Plattform genutzt hat. Es hat jetzt offiziell "Beta"-Status. OpenShift, die Kubernetes-Variante von Red Hat nutzt Tekton inzwischen als Backend für ihre integrierten CI/CD-Plattform, freilich im Status eines "Technology Preview". Webhooks sind im Rahmen des Subprojektes "Tekton Triggers" inzwischen verfügbar. Das Tekton Dashboard hat einige Fortschritte gemacht, ist aber immer noch vergleichsweise spartanisch.

Ein Command Line Interface "tkn" zur Fernsteuerung der Engine ist ebenfalls vorhanden. Hier vermissen wir aktuell Funktionalitäten um Tekton-Tasks lokal komfortabel zu testen. Das ist aufgrund des strikten Container-Ansatzes der Plattform jedoch auch ohne spezielle Unterstützung machbar.

4 https://cd.foundation/
5 https://opensource.googleblog.com/2019/03/introducing-continuous-delivery-foundation.html

"Pipeline as Code" ist aufgrund der auf Custom Resource Definitions basierenden Pipeline-Definitionen quasi erfüllt, jedoch fehlt eine integrierte Funktion um diesen Code aus dem Repository eines Software-Projektes zu beziehen.

Jenkins X

Die Jenkins-Plattform baut in ihrer neuesten Reinkarnation auf dem oben bereits aufgeführten Tekton-Framework auf. Daher betrachtete die erste Veröffentlichung dieses Leitfadens Jenkins X nicht separat. Inzwischen hat sich jedoch einiges jenseits der Basis-Plattform getan, weswegen wir uns diesem Vertreter nun separat widmen.

Jenkins X hat eine etwas andere Philosophie als die anderen Technologien. Hier ist nicht wirklich der CI/CD-Server Zentrum der Betrachtung sondern der Developer-Workflow. Ein Entwickler der eine Pipeline benötigt soll sich über das Kommandozeilen-Tool "jx" unkompliziert eine eigene Jenkins-X-Runtime installieren können, ohne dass irgendwo jemand einen zentrale Server vorbereitet hat.

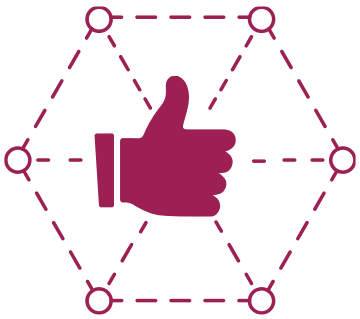
Das Ziel dieser Installation sind in erster Linie die Public-Cloud-Plattformen Google Cloud und AWS. Letztlich findet die Installation jedoch in einem dort gehosteten Kubernetes-Cluster statt. Dieses kann man über vorgefertigte Terraform-Module installieren, dann ist das "jx"-Tool in der Lage auf dieser Basis den Server zu installieren, der letztlich auf einer Tekton-Engine aufbaut.

Dabei stellt sich aus unserer Perspektive natürlich die Frage, inwiefern dieser Prozess spezielle Provider, bzw. extra für diesen Zweck erstellte Kubernetes-Cluster wirklich benötigt, soll unser Tool der Wahl doch grundsätzlich mit jedem Kubernetes auskommen. So existiert auch eine Anleitung für "On Premise"-Installationen die jedoch dafür sensibilisiert, dass in diesem Setup nicht alle Features ohne weiteres funktionieren werden. Das ist einerseits ernüchternd, andererseits aber auch verständlich da Jenkins X viele Integrationen von Haus aus mitbringt, z.B. Anbindung an Secret-Storages, Artifact Repositories, Management von Deployment-Umgebungen etc. Diese pauschal "out-of-the-box" in allen möglichen Umgebungen zum Funktionieren zu bringen ist unrealistisch.

Insofern ist hier die Strategie einfach eine andere: Anstelle maximaler Flexibilität mit den damit verbundenen Umsicherheiten zu bieten definiert Jenkins X eine relativ Enge Auswahl von Zielumgebungen, verspricht dann aber möglichst einfache, fehlerunanfällige Abläufe und nützliche Zusatzfunktionen. Dazu gehört auch weiterhin die Nutzung einer in das Code-Repository eingecheckten Pipeline-Definitionsdatei die inzwischen "jenkins-x.yml" heißt. Eine kompexere Variante mit so genannten "Build Packs", welche den Buildprozess in seiner Gänze definiert, existiert ebenfalls. Deren Besprechung sprengt den Rahmen dieses Leitfadens. Aber auch hier ist offensichtlich: Jenkins X möchte große Teile des Entwicklungsprozesses vordefinieren um dabei so nützlich wie möglich sein zu können. Ob dieser Prozess für das eigene Projekt passt ist eine Frage die man für sich selbst beantworten muss.

Kommen wir noch zu ein paar weniger ambivalenten Fakten: Ein Web-UI ist in der Standardinstallation nicht dabei, kann jedoch separat installiert werden. In den Entwicklungsprozess ist auch ein GitOps Workflow bereits eingebaut. Fragen nach der Natur der Pipeline-Ausführung erledigen sich wegen des Tekton-Unterbaus.

Fazit



Allen Lösungen gemein ist, dass sie sich auf einem guten Weg befinden, das CI/CD-Erbe von Jenkins anzutreten, da sie die aus unserer Sicht erforderlichen Kriterien im Wesentlichen verinnerlicht haben. Dabei verfolgen sie durchaus unterschiedliche Ansätze und bedienen verschiedene Anforderungen und Geschmäcker.

Das Argo-Projekt ist ein hochinteressantes Toolset für Workflows auf Kubernetes, bei dem CI nur ein Anwendungsfall ist, aber kein schlüsselfertiges CI-Produkt. Gerade wenn man Flexibilität sucht oder andere verwandte Anwendungsfelder hat, lohnt sich ein genauerer Blick.

InfraBox ist im Gegensatz dazu eine klassische, zentralistische Lösung mit einem reichen Featureset und dem von Jenkins gewohnten Komfort. Wir sehen hier leider aktuell jedoch nicht viel Momentum.

Tekton entwickelt sich kontinuierlich zu einem sehr guten CI/CD-Framework mit den richtigen cloud-nativen Ansätzen und stabilisiert sich weiter. Es wird sich zeigen ob es weiterhin nur als Engine für größere Systeme betrachtet wird oder als minimalistische Komplettlösung.

Das darauf aufbauende Jenkins X ist eine interessante Software für alle die mit möglichst wenig Aufwand einen in weiten Teilen vordefinierten CI/CD-Workflow in der Public Cloud nutzen wollen und damit klar kommen, dass sie sich dem Framework anpassen müssen und nicht umgekehrt.

Über ConSol und Cloud native

Seit mehr als 30 Jahren begleitet das Münchner IT-Unternehmen ConSol ConSulting & Solutions Software GmbH sowohl lokale als auch internationale Kunden mit passgenauen Lösungen durch den gesamten Software Lifecycle. Dabei stehen wir für eine breit gefächerte technologische Expertise und herstellerunabhängige Beratung – von der Architekturkonzeption über agile Softwareentwicklung und -integration, bedarfsgerechten DevOps Services bis hin zum Betrieb in der Cloud oder On-Premise.

Die ConSol-Experten sind spezialisiert auf **Cloud native Technologien sowie Applikationen** und verfügen über fundiertes Wissen für die Beratung, Entwicklung und den Betrieb in diesem Bereich. Wir haben mit diesem Leitfaden Ihr Interesse geweckt? Sie haben weitere Fragen zum Thema Cloud native? Oder sind Sie auf der Suche nach einem professionellen Partner, der Sie dabei unterstützt, zukunftweisende technologische Neuerungen erfolgreich in Ihrem Unternehmen zu implementieren?

Wir freuen uns, wenn Sie uns an unseren Standorten in **München oder Düsseldorf** kontaktieren!

Must-Have	Hosting und Scheduling auf Vanilla Kubernetes • Das Tool soll vollumfänglich in Kubernetes laufen, sowohl was das Hosting der Tool-Prozesse selbst als auch das Scheduling der CI/CD-Pipelines angeht. • Es darf keine Abhängigkeiten zu speziellen Public Host Providern geben. • Die Installation muss "On Premise" möglich sein.
	Pure Container als Pipeline-Schritte, simple Images als Pipeline-Module • Pure Container sollen als Pipeline-Schritte verwendet werden, die unabhängig vom CI-Tool ausführbar sein sollen. • Der Container wird ausschließlich über container-native Interfaces wie Dateisystem-Mounts, Umgebungs-Variablen und das Container-Entrypoint-Kommando parameterisiert.
	Web-UI • Über eine Webseite muss der Ausführungszustand der gelaufenen Pipelines ersichtlich sein.
	Pipeline as Code • Die Pipeline muss als Quellcode definiert werden können. • Der Pipeline-Code muss aus Sourcecode-Projekten bezogen werden können.
Nice to have	Dezentrale Pipelines • Die Pipelines der Projekte werden in eigenen Namespaces ausgeführt, eventuell auch definiert.
	Leichtgewichtig und einfach installierbar • Die Installation sollte möglichst leicht über gängige Provisioning-Technologien auf Kubernetes möglich sein, z.B. über Helm Charts. • Es soll keine zwingenden Abhängigkeiten zu externen Systemen geben. Ggf. soll ein Installationsmodus verfügbar sein, der die Abhängigkeit über ein automatisch hinzuinstalliertes System löst.
	Keine root-Container • Das Tool sollte auf den Einsatz von Containern, die unter Berechtigungen des "root"-Benutzers laufen, verzichten.
	GitOps-Anbindung • Ein Konzept existiert, um das Tool an eine GitOps-Gesamtlösung anzubinden.
	Konfiguration über Kubernetes-Ressourcen • Das Projekt nutzt Kubernetes-Ressourcen wie z.B. Config Maps oder Custom Resource Definitions, um Projekt-Pipelines zu konfigurieren. Pipeline-Definitionen aus Quellcode-Projekten • Das Projekt unterstützt die Nutzung von Pipeline-Definitionen die als Datei in den Code-Repositories von Software-Projekten hinterlegt sind
Zusätzliche Kriterien	Kostenlos, Open Source, Unlimitiert • Der Einsatz des Tools muss frei von jeglichen Lizenzzwängen sein.
	Production ready • Das Tool soll zum Zeitpunkt dieses Artikels produktiv einsetzbar sein, zumindest auf dem Niveau eines stabilen und feature-kompletten Beta-Releases.

Tabelle 1: Tabelle der Kriterien für cloud-natives CI/CD

	Argo	InfraBox	Tekton	Jenkins X
Must-Have				
Hosting & Scheduling auf K8s	+	+	+	+/-
Pure Container in der Pipeline	+	+	+	+
Web-UI	+	+	+	+
Pipeline as Code	+	+	+	+
Nice to have				
Dezentrale Pipelines	+	-	+	+
Leichtgewichtig	+	-	+	+
Kein root	-	-	-	-
GitOps	+	-	-	+
K8s-Resources als Pipelines	+	+	+	+
Pipeline-Dateien im Repository	-	+	-	+
Zusätzliche Kriterien				
Kostenlos, OpenSource, Unlimitiert	+	+	+	+
Production ready	+	+	-	+

Tabelle 2: Kriterien-Matrix der Lösungen

ConSol
Consulting & Solutions Software GmbH

St.-Cajetan-Straße 43
D-81669 München
Tel.: +49-89-45841-100

Kanzlerstraße 8
D-40472 Düsseldorf
Tel.: +49-211-339903-0

vertrieb@consol.de
www.consol.de
Folgen Sie uns auf Twitter: @consol_de