S&N INVENT

Shift Left – Produktion und Entwicklung sind gleich

Lukas Voß, Armin Vogt

27.06.2023

Inhaltsverzeichnis

1 Software-Entwicklung im Zeitalter der Cloud 3

2 Was bedeutet Shift Left? 3

3 Cloud-native Shift Left 3

4 Werkzeuge 5

5 Wie sieht die lokale Entwicklungsumgebung für Cloud-native Softwareentwicklung aus? 6

6 Wie verändert Cloud-native Shift Left die Entwickler-Arbeitsweise? 6

6.1 Produktionsfehler nachtesten 6

6.2 Onboarding in bestehendes Cloud-natives Projekt 7

6.3 Cloud-Konfiguration ausarbeiten 7

7 Einführung Cloud-nativer Softwareentwicklung 8

1 Software-Entwicklung im Zeitalter der Cloud

Die IT-Industrie ist bekanntermaßen ständig im Wandel. Sie wird agil, cloud-basiert und nicht zuletzt Cloud-native. Ein Hauptgrund für den Wandel sind Kosten und deren Einsparung. Konsequenterweise wird daher angestrebt, die Softwareentwicklung schneller und effizienter zu gestalten.

Vermeidbare Kostentreiber sind unter anderem: Das langwierige Onboarding von Entwicklern, spät gefundene Softwarebugs, langsame Entwicklungszyklen, späte Integrationstests, simulierte Umsysteme und die fehlende Übertragbarkeit von Entwicklungsumgebungen. Dazu kommen Schwierigkeiten, für Entwickler eine Umgebung zu schaffen, die das isolierte Nachtesten von Produktionsfehlern ermöglicht.

Diese Kosten können verringert werden, indem Cloud-native Technologien und Arbeitsweisen bis hinunter zum Entwicklerarbeitsplatz etabliert werden. Was das ist und warum es die Arbeitsweise effizienter machen kann, wird im Folgenden erläutert.

2 Was bedeutet Shift Left?

Shift Left ist ein Begriff aus der Software-Qualitätssicherung (auch: Shift Left Testing). Er bezeichnet das Verlagern von qualitätssichernden Tätigkeiten zu einem früheren Zeitpunkt – auf dem Zeitstrahl nach links!

Ziel ist es, Integrationstests schon vor einem Deployment auf der Integrationsumgebung durchzuführen. Fehler sollen früher gefunden werden. Denn je später ein Fehler gefunden wird, desto teurer wird seine Behebung. Um das zu erreichen, werden neben Unit Tests auch Integrationstests in der CI-Pipeline durchgeführt. Dabei wird die fehlende Integrationsumgebung ersetzt durch ein Gerüst von Ersatz-Modulen: diese simulieren andere Komponenten. Ihr Verhalten ist in einem Drehbuch vorgeschrieben oder durch vereinfachte Algorithmen nachgebildet.

Diese Ersatz-Module weichen im Verhalten immer ab von den echten Modulen. In sie hinein codiert sind Annahmen über das Verhalten. Häufig werden diese Annahmen geschrieben von einem Entwickler, der selber gar nicht Experte für das Modul ist. Die Ersatz-Module müssen gepflegt werden und auch den Versionszyklus der echten Module mitgehen.

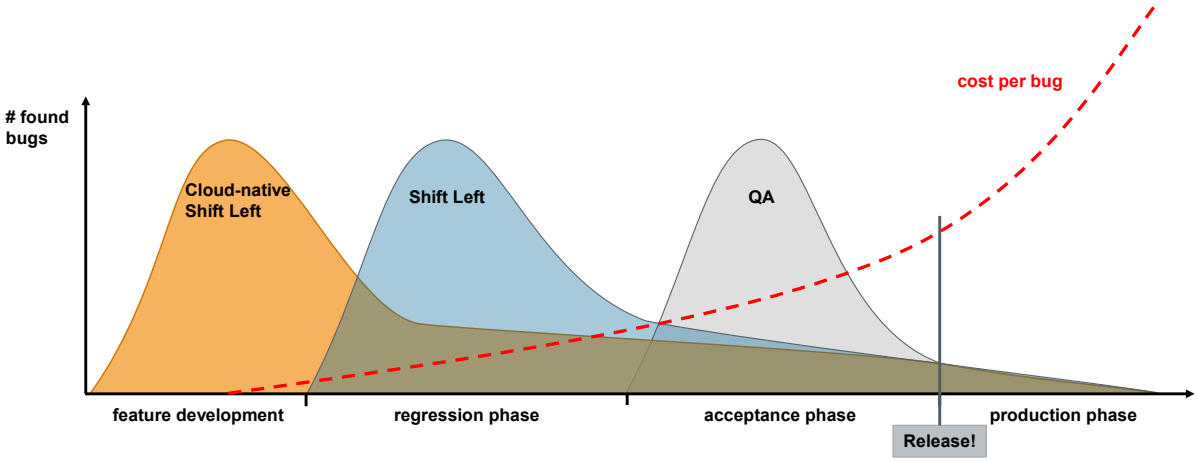
3 Cloud-native Shift Left

Wünschenswert war schon immer der Test mit der gesamten, echten Umgebung. Aus Kostengründen war diese bisher beschränkt auf eine dedizierte Testphase, in einer Methodik, die viel Vorbereitung in allen beteiligten Modulen voraussetzt.

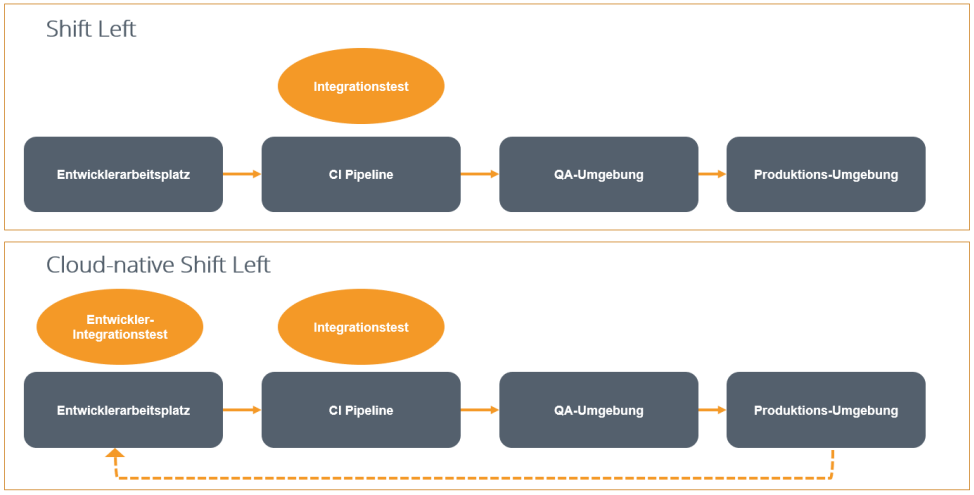
Heute wird viel für den Betrieb in der Cloud entwickelt, denn mit der Cloud sollen Kosten gespart werden. Selbst entwickelte Software wird als image bereitgestellt und auf einer Container-Plattform ausgebracht.

Als Image paketierte Software ist leicht übertragbar und ausführbar auf jeder Container-Plattform (Kubernetes). Container-Plattformen finden sich beim Cloud-Anbieter gemietet, im unternehmenseigenen Rechenzentrum und auch lokal auf dem eigenen Arbeitsplatz.

Jetzt werden die echten Module transportabel. Auf die Ersatz-Module können wir verzichten.



In der Abbildung ist dargestellt, wann bzw. wo der Integrationsaufwand bei herkömmlicher Softwareentwicklung, Shift Left und Cloud-native Shift Left stattfindet. Bei Cloud-native Shift Left finden die Integration und die Integrationstests bereits bei dem Entwickler statt, noch vor der CI-Pipeline. Wie das geht, wird nachfolgend erläutert.



Shift Left hilft uns, Fehler früher zu erkennen. Aber so früh wie möglich? Das kann nur ein Setup schaffen, welches Cloud-native ist. Die Integrationstests können in der Entwicklungsumgebung durchgeführt werden, nicht erst in der CI-Pipeline. Das wird dadurch erreicht, dass

der Entwickler die Produktionsumgebung lokal startet. Das hat viele Vorteile. Zum einen kann der Entwickler sein neues Feature direkt integrieren und testen. Fehler kann er sofort identifizieren. Dank Containerisierung kann er gezielt Versionen der anderen Module auswählen. Zum anderen kann er in der Produktion oder Testumgebung gefundene Fehler lokal nachstellen und beheben – mit der Gewissheit, dass es auch in den weiteren Umgebungen gelöst ist.

Die Entwicklung wird dadurch schneller und zuverlässiger.

Das Aufsetzen einer Entwicklungsumgebung darf dabei nicht länger als ein paar Minuten dauern. Die Einrichtung der lokalen Entwicklungsumgebung darf nicht durch die Installation von Bibliotheken oder durch „works-on-my-machine“ geprägt sein, um unnötigen Zeitverlust zu vermeiden. Ein Kontextwechsel zwischen zwei unterschiedlichen Projekten darf nicht erfordern, dass seitenweise Dokumentation gelesen werden muss, um zu verstehen wie die Umgebung lokal eingerichtet wird und welche Pakete / Software zu installieren und zu konfigurieren sind.

4 Werkzeuge

Die wichtigsten Werkzeuge im Cloud-native(n) Werkzeugkasten sind dabei Linux und Docker. Linux nutzen wir unter Windows über WSL (Windows Subsystem for Linux) und Docker über Docker for Windows/Mac.

Linux ist ein wichtiger Baustein, da (fast) alle Systeme, die wir heute entwickeln, in einem Linux-System ausgeführt werden. Container Plattformen laufen mit Linux und die meisten Container sind Linux-basiert. Da ist es naheliegend, bereits in der Entwicklungsumgebung Linux zu benutzen. Das Arbeiten mit dem Linux-Terminal kann viele Abläufe wesentlich beschleunigen. Zudem erleichtern die Linux-Kenntnisse auch den Umgang mit Containern und Container-Plattformen.

Docker als zweiter Baustein ermöglicht uns die Übertragbarkeit (und Reproduzierbarkeit) der Entwicklungsumgebung zu anderen Entwicklern und in die Cloud. Docker integriert sich nahtlos in die Linux Umgebung. Wir können fixierte Bibliotheken nutzen und verhindern so das „works on my machine“-Problem. Docker Compose ermöglicht uns, komplexe Systeme mit einem Kommando zu starten und zu steuern.

Eine Cloud-native IDE wie Visual Studio Code integriert sich perfekt mit dem Linux-Terminal und Docker. Visual Studio Code unterstützt außerdem Devcontainer, welche die Flexibilität der Entwicklungsumgebung auf ein neues Level heben. Das gesamte Projekt wird dann nicht in Windows oder im WSL-Linux zur Ausführung gebracht, sondern mit allen Abhängigkeiten in einem Container. Die Entwicklung selbst findet also im Container statt und steigert die Übertragbarkeit der Entwicklungsumgebung noch weiter.

Ein Proxy wie Traefik kann auch sehr wertvoll für die lokale Umgebung sein. Traefik reagiert auf Labels, die in einem Docker Compose File gesetzt werden können und kann so flexibel das Problem des Routings zwischen den verschiedenen Komponenten lösen. Es imitiert den Edge Router / Ingress Controller der Cloud, da die Anwendung(en) an einem Punkt

zusammenlaufen und per name-based routing erreicht werden. Mithilfe von Traefik umgeht man umständliche Setups mit unzähligen Port-Weiterleitungen auf der Entwicklungsmaschine.

Je nach Arbeitsauftrag kann es sinnvoll sein, lokal ein Kubernetes-Cluster zu starten. Die Workloads, die wir entwickeln, werden auf Kubernetes-Clustern deployed. Daher ist es in manchen Fällen notwendig, ein Kubernetes-Setup lokal zu testen, bevor man es auf dem Entwicklungscluster anwendet. Hier hilft uns wieder Docker for Windows/Mac, mit welchem wir auf Knopfdruck ein Kubernetes-Cluster starten. Hier können wir ein Service Mesh deployen und einrichten, Ingress Routing testen oder Manifeste für unsere Anwendung schreiben und testen.

5 Wie sieht die lokale Entwicklungsumgebung für Cloud-native Softwareentwicklung aus?

Die exemplarische lokale Entwicklungsumgebung (unter Windows) nutzt eine in Linux integrierte Docker-Installation und Visual Studio Code.

Das Projekt wird im Linux-System ausgecheckt und Visual Studio Code („VSC“) gestartet. Unter den Projektdateien befindet sich eine Devcontainer-Definition, welche VSC nutzt, um den Devcontainer zu bauen und zu starten. Mit dem Microservice verbundene Services werden mithilfe von Docker Compose in der konfigurierten Version über eine Image-Registry abgerufen und gestartet. Schon kann der Entwickler mit der eigentlichen Entwicklungstätigkeit beginnen. Er muss nicht erst Java, Node, Maven oder andere Pakete in der korrekten Version einrichten.

Bevor der Entwickler Code-Änderungen committet, laufen lokal die Code- und Dependency-Analysen (Auch integriert in der IDE). Gebaute Images werden mit einem Vulnerability-Scanner wie Trivy überprüft. Der Entwickler kann dann vor dem push nachbessern, da er weiß, dass Images mit Sicherheitslücken nicht ausgerollt werden.

Die Entwicklung wird unterstützt durch das Linux-Terminal, welches mit seinen vielfältigen Möglichkeiten den Entwickler entlastet.

6 Wie verändert Cloud-native Shift Left die Entwickler-Arbeitsweise?

6.1 Produktionsfehler nachtesten

Ein Fehler im Produktivsystem wurde gefunden und muss schnellstmöglich behoben werden. Ein Umsystem war kurzzeitig nicht erreichbar, was in der Produktion zu einem Datenverlust geführt hat.

Nicht-Cloud-native:

Ein Entwickler bekommt den Auftrag, den Fehler zu finden und zu beheben. Logs werden analysiert und die lokale Entwicklungsumgebung verändert, um den Fehler lokal nachstellen zu können. Das nachgelagerte System zu starten ist nicht möglich, da es von einem anderen

Team entwickelt und nicht containerisiert ist. Es wird ein Unit-Test geschrieben, in dem ein Systemausfall simuliert wird und der Fehler wird behoben. Der Fix wird auf einer Testumgebung ausgerollt und nachgetestet. Hier scheitert der Test aufgrund eines nicht bedachten Fehlers in der Netzwerktopologie. Der Entwickler schaut sich die Logs an … Insgesamt entsteht sehr viel Aufwand, da die Produktionsumgebung nicht lokal gestartet werden kann. Im Anschluss muss der Entwickler seine Entwicklungsumgebung wieder verändern, um den aktuellen Codestand bearbeiten zu können, da dieser gegebenenfalls andere Abhängigkeiten wie eine andere Java-Version erfordert.

Cloud-native:

Der Entwickler bekommt den Auftrag, den Fehler zu finden und zu beheben. Er analysiert die Logs, um die Ursache zu finden und checkt den Codestand der Produktion aus. Er öffnet das Projekt im Devcontainer, führt ein Kommando aus und startet lokal die komplette Produktionsumgebung, da die Umsysteme in einer Image-Registry bereitgestellt sind. Er kann den Fehler nachstellen und behebt ihn, mit der Gewissheit, dass es auch in der Produktion funktionieren wird.

Das spart viel Zeit, Geld und Nerven. Im Anschluss kann die gewohnte Entwicklertätigkeit wieder aufgenommen werden, die Entwicklungsumgebung ist containerisiert und muss nicht neu aufgebaut werden.

6.2 Onboarding in bestehendes Cloud-natives Projekt

Ein zweiter Anwendungsfall besteht im Onboarding neuer Entwickler.

Nicht-Cloud-native:

Ein neuer Entwickler wird eingearbeitet. In vielen Fällen bedeutet das vor allemdie Entwicklungsumgebung einrichten. Dazu gehört das Auschecken des Codes, die Installation von Abhängigkeiten (z. B. Java, Node, Maven in der richtigen Version) sowie die Einrichtung von zu installierender Software, wie Datenbanken.

Dieser Vorgang kann mitunter nicht nur Stunden, sondern auch Tage in Anspruch nehmen. Vermeintliche Kleinigkeiten können dazu führen, dass mehrere Projektmitglieder gemeinsam eruieren, warum es auf dem einen Computer funktioniert, auf dem anderen aber nicht.

Cloud-native:

Der neue Entwickler wird eingearbeitet. Docker und Visual Studio Code hat er bereits installiert. Er checkt den Code aus und startet Visual Studio Code im Devcontainer. Datenbanken und andere Software werden mit einem Befehl gestartet – fertig.

6.3 Cloud-Konfiguration ausarbeiten

Ein dritter Anwendungsfall betrifft das Ausarbeiten von Cloud-Konfigurationen. Damit gemeint sind Kubernetes-Manifeste wie Deployments, Network-Policies oder Service Mesh-Konfigurationen.

Nicht-Cloud-native:

Der DevOps-Engineer nutzt die gemeinsame DEV-Stage, um die Konfigurationen zu testen. Die DEV-Umgebung weicht von der Produktionsumgebung ab, da letztere hochverfügbar ist. Für die anderen Entwickler bedeutet das Downtime von Anwendungen und für den DevOps-Engineer einen gewissen Druck, keine Fehler machen zu dürfen. Außerdem hat er die Ungewissheit, ob die Konfiguration trotz der Systemunterschiede zwischen DEV und PROD funktioniert.

Cloud-native:

Der DevOps-Engineer startet lokal sein Kubernetes. Er installiert die Manifeste der aktuellen Produktionsumgebung und nimmt Änderungen vor. Die gemachten Änderungen testet er umfangreich bereits lokal, damit keine Ausfälle der nachfolgenden Stages drohen.

7 Die Einführung von Cloud-nativer Softwareentwicklung

Richtig eingesetzt fördern Cloud-native Techniken Effizienz und erhöhen die Testbarkeit.

Entwicklung unter Vollintegration wird zur neuen Normalität für die Softwareentwickler. Wenn Fehler früher im Softwareentwicklungsprozess gefunden werden können, wird Softwareentwicklung günstiger und kommt früher zu einem Ergebnis.

Die ersten Schritte in Richtung Cloud-nativer Softwareentwicklung sind schnell beschritten. Für den Erfolg entscheidend ist, die betroffenen Entwickler auf diesem Weg mit einzubeziehen. Dazu gehören gemeinsame Gespräche, Schulungen und eine Phase des Kennenlernens der neuen Tools. Auch der organisatorische Rahmen muss dabei betrachtet werden, unter Umständen sind Freigaben von Images und Projektdokumentationen der nebenläufigen Projekte notwendig, um die Software in einer vollintegrativen, exklusiven Umgebung starten zu können.

Wir beraten Sie gerne auf Ihrem Weg zur Cloud-nativer Softwareentwicklung. Sprechen Sie uns an!