Sonderdruck aus JavaSPEKTRUM

Reden ist Gold

Chatbots als Beschleuniger des Cloud-Betriebs

Frederik Bäßmann, Tobias Unger

Bei richtiger Verwendung können Bots ein Chat-Tool in eine Betriebszentrale für eine cloud-basierte DevOps-Umgebung verwandeln. Leider ist die Umsetzung eines ChatOps-Ansatzes alles andere als trivial: Die Bots müssen sich nahtlos in die DevOps-Prozesse integrieren, müssen ein verständliches Interaktionsmodell bieten und selbst nicht übermäßig Wartungsaufwand erzeugen. Dieser Artikel bietet eine Unterstützung bei der Einführung von Bots in einer DevOps-Umgebung. Welche Chat-Tools sollen unterstützt werden? Sollen existierende Bots verwendet oder sollen die Bots selbst entwickelt werden? Welches Interaktionsmodell soll ein Bot bieten? Diese und weitere Fragen werden im Folgenden auf Basis von Projekterfahrungen diskutiert.

Ein zentraler Aspekt bei der Umsetzung des DevOps-Ansatzes ist die Kommunikation der Teammitglieder untereinander, also der Austausch und die Übermittlung von Informationen im alltäglichen Geschäft der Softwareentwicklung. Ferner spielen die Aspekte Automatisierung und Fortschrittsmessung eine Rolle. Dennoch liefern Retrospektiven – auch in Projekten mit funktionierender DevOps-Umgebung – das Ergebnis, dass die Kommunikation verbessert werden muss. Oft fühlen sich Entwickler nicht genug darüber informiert, warum zum Beispiel außerplanmäßig ein Notfall-Deployment durchgeführt wurde. Auch eine Post-mortem-Analyse liefert meist kaum Aufschluss darüber, da die Informationen in E-Mails und in zig Systemen der DevOps-Umgebung verteilt sind.

Von DevOps zu ChatOps

Dies führt oft zum Wunsch, die Informationen an einem zentralen Ort – einer Art Betriebszentrale – zu bündeln und, falls möglich, die DevOps-Umgebung auch von diesem zentralen Ort zu steuern. Es wird also ein Ansatz gewählt, bei dem der Informationsaustausch, die projektrelevante Dokumentation und die technischen

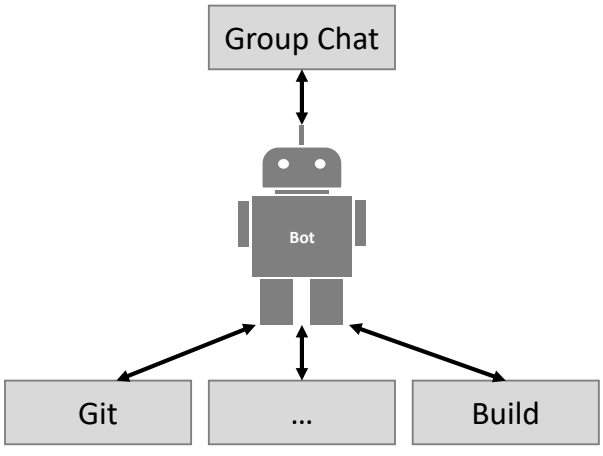
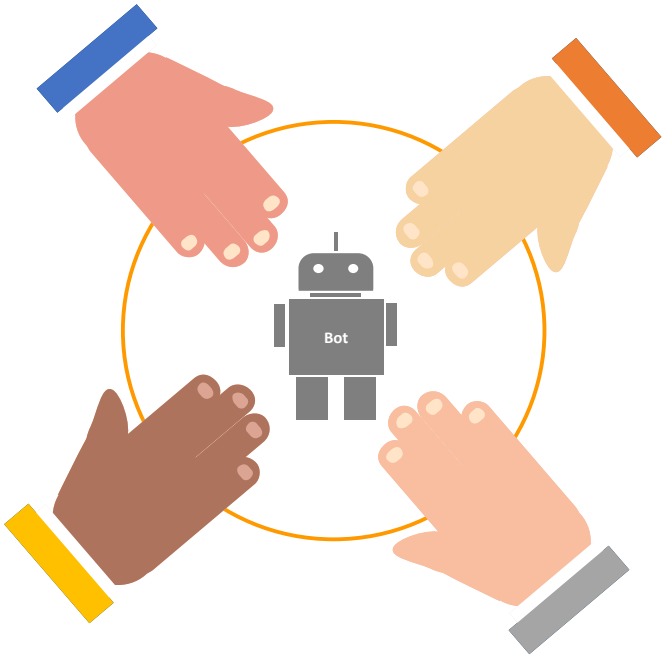


Abb. 1: Chatbot als Middleware



Prozesse mithilfe eines als Einheit verstandenen Systems abgewickelt werden (one system). Aus dieser Überlegung heraus reifte im Projektteam der Entschluss, die DevOps-Umgebung zu einer ChatOps-Umgebung zu erweitern.

Dadurch ist es nicht mehr notwendig, jeden einzelnen Arbeitsschritt anzukündigen (z. B. per E-Mail), da das mit den ausgeführten Arbeitsschritten korrespondierende Feedback sofort automatisch im Chat persistiert wird (persist and document), um allen Teilnehmern die Möglichkeit zu geben, sich zu informieren und zu reagieren (notify and share). Für die Initialisierung und Ausführung von technischen Prozessen wird ein Bot eingesetzt, den sich alle Chat-Teilnehmer teilen und dem sie Kommandos eingeben können, deren Ausführungszeit, Syntax und Ergebnis allen Teilnehmern sichtbar sind. – Ein Bot fungiert also als Middleware (s. Abb. 1) zwischen verschiedenen Diensten, die alle zentral gesteuert werden können (control and interact).

Eine erste Analyse des ChatOps-Marktes ergab, dass sehr viele Tools, Frameworks und Bots existieren, die jedoch nicht komplett zu unserer Umgebung passen. Da das Thema parallel auch in anderen Projekten an Relevanz gewann, entschieden wir uns, einen Proof-of-Concept (PoC) aufzusetzen mit dem Ziel, einen Entscheidungskatalog für den Einsatz von Bots zu entwickeln.

ChatOps in a Nutshell

„ChatOps“ [Han16] bezeichnet ein Konzept, das aus der DevOps-Bewegung hervorgeht, und es beschreibt die Herangehensweise der Informationsverbreitung innerhalb von Teams und Organisationen durch Gruppen-Chat-Tools. Kontextbezogene Diskussionen und Aktionen, die sich aus dem Chat-Verlauf ergeben, werden dem Team durch ein einheitliches Interface zugänglich gemacht und erlauben die Ausrichtung der Arbeitsweise anhand von Informationen, die aus dem Chat-Verlauf gewonnen werden.

Frederik Bäßmann verfügt über neun Jahre Erfahrung in den Bereichen Java SE/EE, BPM und Softwarearchitektur. Sein aktueller Fokus liegt auf der Konzeption und Entwicklung von Individualsoftware in wirtschaftlichen und behördlichen Umfeldern. E-Mail: frederik.baessmann@opitz-consulting.com

Tobias Unger verfügt über mehr als zehn Jahre Erfahrung in den Bereichen Enterprise Architecture, BPM und Java Enterprise. Sein aktueller Fokus liegt auf Design und Implementierung von Integrations- und Prozessautomatisierungslösungen. Ein weiterer Fokus liegt im Aufbau von DevOps-Umgebungen und dem Einsatz von Chatbots. E-Mail: tobias.unger@opitz-consulting.com

Ein weiterer Mehrwert besteht darin, dass technische Prozesse direkt über den Chat gesteuert werden können. Teammitglieder und Bots können auf Nachrichten im Chat reagieren; Bots können automatisch Aktionen auslösen, das heißt, technische Prozesse anstoßen, ablaufen lassen und anschließend umgehend ein automatisch generiertes, aus der Aktion beziehungsweise aus dem Prozessablauf resultierendes Feedback im Chat persistieren. Das Ziel ist demnach, den Chat neben dem Nachrichtenaustausch zwischen den Teilnehmern auch als Interface und Kontrollmechanismus für andere Systeme einzusetzen.

Ziel definieren

An vorderster Stelle steht die Zieldefinition. Die Fähigkeiten eines Bots unterscheiden sich sehr: Ein bestimmter Bot kann nur Statusinformationen (z. B. über den Build-Status) in den Chat einfügen, ein anderer Bot bietet einem Entwickler die Möglichkeit, Statusinformationen aktiv anzufordern, und eine weitere Bot-Variante kann Prozesse und Systeme im Sinne der oben beschriebenen Betriebszentrale aktiv steuern.

Szenarien definieren

Um die Ziele noch feingranularer definieren zu können, definiert das Projekt zu den umgesetzten Entwicklungsprozessen passende Szenarien. Wichtig ist, dass hierbei die eigenen Entwicklungsprozesse berücksichtigt werden, die teils sehr individuell sind und dadurch spezielle Anforderungen an einen Bot stellen. Des Weiteren liefert die Retrospektive gute Anhaltspunkte für die Definition der Szenarien. Hierbei sollte aber in der Retrospektive darauf geachtet werden, den abstrakten Punkt „Kommunikation“ auf den konkreten Bedarf herunterzubrechen.

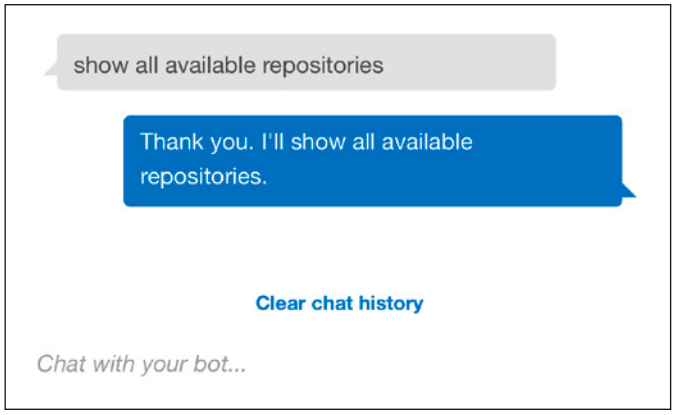


Abb. 2: Befehl in natürlicher Sprache

Die Projektmitarbeiter wünschen sich zum Beispiel die Möglichkeit, sich schnell einen Überblick darüber verschaffen zu können, welche Pull-Requests offen sind, damit diese Information nicht ständig im Browser auf der GitHub-Oberfläche gesucht werden muss. Der Bot muss also die Möglichkeit bieten, dass alle Projektbeteiligten über den Chat die aktuellen Pull-Requests abfragen können.

Des Weiteren ergibt sich auch der Bedarf für konkrete Prozessoptimierungen. Funktionalitäten der Benutzungsoberfläche müssen laut Prozess gesondert vom Fachbereich abgenommen werden. Derzeit erfolgt dies durch viel Telefonkommunikation. Dieser Prozess soll dahin gehend optimiert werden, dass er semi-automatisiert über den Bot gesteuert wird.

Dazu soll der Bot auf die Erstellung eines Pull-Requests reagieren und durch Setzen eines Status-Checks in GitHub das sofortige Mergen verhindern. Danach fragt der Bot im Chat, ob dieser Pull-Request durch den Fachbereich getestet werden soll. Bei positiver Bestätigung deployt der Bot den Code in eine Testumgebung und wartet auf Feedback, andernfalls wird der Pull-Request wieder freigegeben.

Entscheidung 1: Interaktionsmodell

Nachgelagert zur Definition der Szenarien schließt sich die Frage an, wie mit dem Bot interagiert werden soll. Eine Möglichkeit stellt die Interaktion mittels natürlicher Sprache dar. Hierbei kann dem Bot direkt ein Befehl wie „Zeige mir alle Pull-Requests“ gegeben werden (s. Abb. 2). Der Vorteil dabei ist, dass Entwickler dem Bot einfach ein Befehl stellen können, ohne sich an syntaktische Vorgaben halten zu müssen. Ein Nachteil ist, dass der Bot unter Umständen den Befehl nicht sofort versteht und nachfragen muss. Dazu muss er in der Lage sein, Konversationen zu verwalten und den Status der Konversation zu halten, damit der Fragesteller nicht immer die ganze Frage von Neuem eingeben muss, und somit einfach direkt auf die Nachfragen des Bots antworten kann (s. Abb. 3).

Etwas einfacher gelagert ist die Verwendung eines Command-Styles, das heißt, der Bot funktioniert wie eine Kommandozeile, bei der Befehle in vorgegebener Syntax eingegeben werden müssen. Der Vorteil hierbei ist, dass die Eingabe exakt ist und auch kein Status der Konversation gehalten werden muss, da der Bot auf eine falsche Eingabe mit einer Beschreibung der Syntax antwortet. Der Nachteil ist, dass die Entwickler sich weitere Kommandos merken müssen und nicht intuitiv mit dem Bot interagieren können. Aus diesem Grund wurde für das Szenario zur Abfrage der Pull-Requests natürliche Sprache als Interaktionsmodell gewählt.

Die letzte Möglichkeit ist die Verwendung von UI-Elementen, wie sie beispielsweise Slack erlaubt. Dabei kann der Bot zum Beispiel eine Frage stellen und die möglichen Antworten als Buttons vorgeben (s. Abb. 4). Die oben genannte Entscheidung, ob ein Deployment erfolgen soll oder nicht, erfordert eine exakte Antwort – ja oder nein –, um den Prozess fortsetzen zu können. Deshalb fiel hier die Entscheidung zugunsten von UI-Elementen.

Generell eignet sich der Einsatz von UI-Elementen vor allem für Entscheidungen innerhalb von Prozessabläufen. Zum Starten von

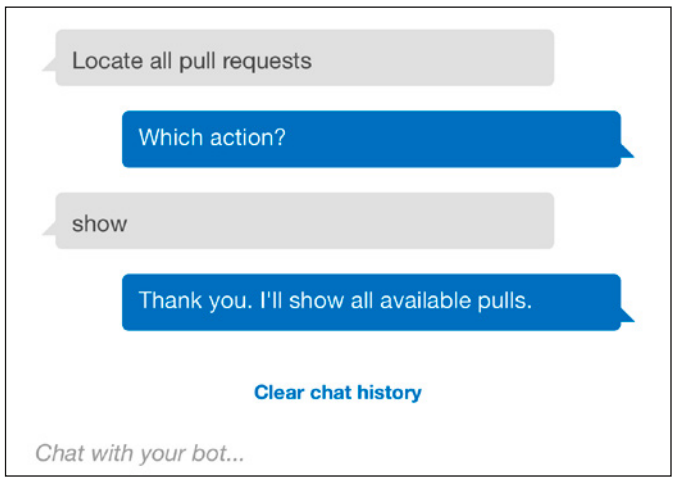


Abb. 3: Nachfrage des Bots

Prozessen und Aktionen hat sich die Nutzung des Command-Styles bewährt. Einerseits können hier viele Varianten abgebildet werden, was mit UI-Elementen aufgrund der Kombinatorik der Optionen kaum zu gewährleisten ist. Andererseits müssen die Informationen exakt eingegeben werden, was bei der Eingabe in natürlicher Sprache unter Umständen zu einer komplexen und langwierigen Konversation führt. Zur Beschaffung von Informationen eignet sich natürliche Sprache, da einfach eine Frage an den Bot gerichtet werden kann und der Entwickler anschließend durch eine Konversation zur Antwort geleitet wird. Die Kenntnis von Kommandos und die Einhaltung einer speziellen Syntax sind somit nicht nötig.

Entscheidung 2: Tool-Unterstützung

Nach Klärung der nicht-technischen Themen stand die Klärung der technischen Themen an. Zuallererst steht die Klärung, welche Systeme der Bot ansteuern soll. Generell gilt, dass die Komplexität mit der Anzahl der zu steuernden Systeme wächst. Architektonisch betrachtet spielt der Bot hier die Rolle einer klassischen Integrationsmiddleware.

Eine weitere Komplexität entsteht in Ermangelung von Standardformaten und APIs zur Integration. Zwar bieten die meisten Systeme ein API zur Integration, jedoch unterscheiden sich diese Programmierschnittstellen in großem Maße in den verwendeten Datenformaten, Interaktionsmustern und Sicherheitsmechanismen. Dadurch muss gewissermaßen jedes System getrennt angebunden werden. Selbst Systeme oder Dienste wie GitHub oder Bitbucket, die vergleichbare Funktionalitäten bieten, unterscheiden sich in ihren APIs sehr stark.

Entscheidung 3: Make or Buy/Use

Ein Ausweg aus dieser Integrationsproblematik kann die Verwendung vorgefertigter Bots sein. Dabei ist genau abzuklären, ob der Bot wirklich die gewünschten Szenarien abdecken kann. Meistens

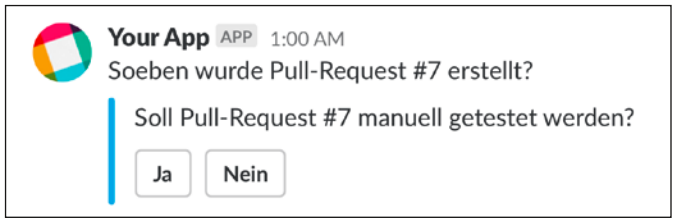


Abb. 4: Interaktion mit Buttons in Slack

bieten diese Bots nur einen Teil der gewünschten Funktionalität oder unterstützen nicht das gewünschte Interaktionsmodell. Des Weiteren ist eine Unterstützung aller zu integrierenden Systeme zu prüfen. Ist diese nicht gegeben, gilt es abzuwägen, ob der parallele Einsatz mehrerer Bots eine Lösung ist oder eine Eigenentwicklung nötig wird.

Eigenentwicklungen bieten den Vorteil, dass sie auf die eigenen Szenarien und Prozesse maßgeschneidert werden können. Andererseits entsteht dadurch Software, die ebenfalls erweitert und gewartet werden muss, beispielsweise bei Updates eines verwendeten APIs, was im aktuellen Beispiel bei Verwendung von Slack, GitHub und AWS CodeBuild [AWS] mit hoher Wahrscheinlichkeit eintritt.

Entscheidung 4: Bereitstellungsmodell

Einer Entscheidung bedarf auch die Bereitstellung des Bots. Manche Bots sind nur „as a Service“ verfügbar und können nicht selbst gehostet werden. Soll der Bot selbst gehostet werden, muss die passende Infrastruktur bereitgestellt werden. Das kann sowohl on-premise als auch in der Cloud geschehen. Stehen Systeme sowohl on-premise als auch in der Cloud zur Verfügung, muss gewährleistet werden, dass der Bot alle Systeme erreichen kann. Da die meisten Systeme (z. B. GitHub) Webhooks nutzen, um Status-Updates zu melden, muss auch die Erreichbarkeit des Bots gewährleistet sein.

Im aktuellen Beispiel erfolgt das Hosting des Bots in der Cloud. Jedoch existiert die Einschränkung, dass die Implementierung weitgehend dem AWS Servless Application Model (SAM) entspricht. Dieser Ansatz erlaubt eine Wiederverwendung des existierenden Know-hows, da die in der ChatOps-Umgebung entwickelten Applikationen ebenfalls dem SAM folgen.

Entscheidung 5: Nutzung von Frameworks

Wesentliche Erleichterung bei der Implementierung versprechen auch Bot-Frameworks, wie HubBot [HUB] oder das Microsoft Bot Framework [MBF]. Hier gilt es zu prüfen, ob die Frameworks die benötigten Konnektoren zu den anzubindenden Systemen unterstützen und ob die benötigten Interaktionsmodelle unterstützt werden. Nicht alle Frameworks bieten derzeit eine Unterstützung für UI-Elemente. Manche Frameworks beschneiden zusätzlich die möglichen Bereitstellungsmodelle. Das MBF erlaubt zwar das Hosting des Bots on-premise, die Konnektoren zu den Chat-Systemen laufen aber in der Cloud in Azure. Deshalb werden momentan auch nur öffentlich verfügbare Chat-Systeme wie Slack oder Microsoft Teams unterstützt.

Für die Implementierung der Szenarien entschied sich das Projekt für den Verzicht auf ein existierendes Framework, hauptsächlich wegen mangelnder Unterstützung für AWS Lambda, welches als Umsetzungsplattform genutzt werden soll.

Entscheidung 6: Mandantenfähigkeit

Wird der Bot von mehreren Teams oder Organisationen eingesetzt, sollte eine Mandantenfähigkeit vorhanden sein, um Arbeitsbereiche sauber trennen zu können. Prinzipiell lässt sich Mandantenfähigkeit auf zwei verschiedene Weisen implementieren:

Die Mandantenfähigkeit kann direkt in die Implementierung eingebaut werden und weitere Mandaten können dann per Konfiguration, zum Beispiel über ein Web-Interface, hinzugefügt werden. Dieser Ansatz nennt sich „Single-Instance“.

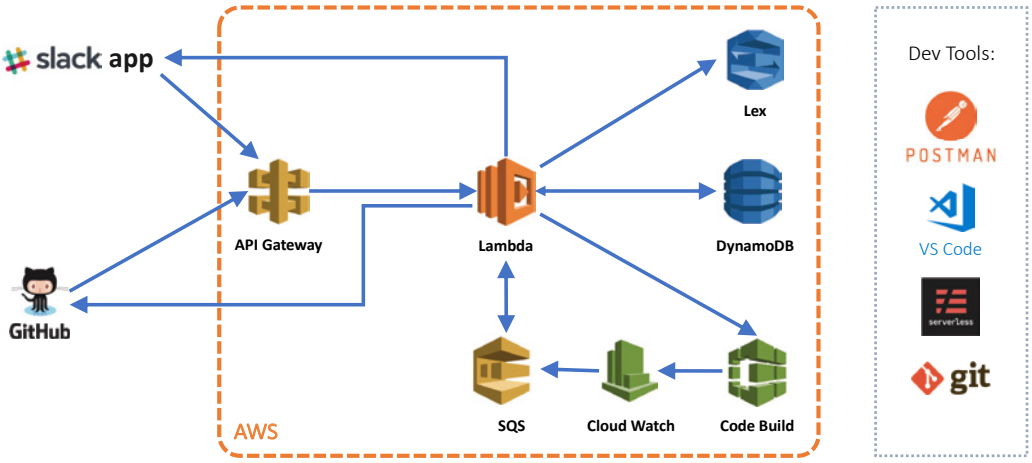


Abb. 5: Architektur des Bots

Der Multi-Single-Instance-Ansatz besteht darin, dass für jeden Mandanten eine eigene Instanz des Service bereitgestellt wird. Für die Realisierung der zwei oben genannten Szenarien wurde dieser Ansatz gewählt. Da mehrere Entwickler gleichzeitig mit der Implementierung betraut waren, muss die Möglichkeit einer Mehrfachbereitstellung sowieso unterstützt werden.

Sicherheit, Identitätsmanagement und Datenschutz

Bei der Implementierung von Bots muss schon beim Design auf Sicherheit geachtet werden. Auch sollten alle Sicherheitsmechanismen der integrierten Systeme genutzt werden. GitHub beispielsweise verlangt beim Anlegen eines Webhooks einen Token, den nur GitHub und der Bot kennen sollten. Mithilfe des Tokens berechnet GitHub einen HMAC hexdigest aus der Nachricht und liefert diesen als http-Header mit aus. Die Bot-Implementierung kann die Nachricht dadurch validieren und somit GitHub als Absender der Nachricht verifizieren.

Damit der Bot aus allen Systemen Nutzer zuordnen kann, müssen alle Identitäten in allen Systemen bekannt sein. Hier empfiehlt es sich, ein Identitätsmanagement zu implementieren, um die Benutzerkonten zentral verwalten zu können. Eine separate Verwaltung der Benutzerkonten pro System führt auf Dauer meist zu Inkonsistenzen.

Beispielarchitektur mit SAM

Auf Basis der Entscheidungen konzipierte das Projektteam die Architektur des exemplarischen Bots (s. Abb. 5). Die Logik wird mittels Lambda-Funktionen realisiert. Statusinformationen speichert der Bot in einer DynamoDB. Die Aktionen von UI-Elementen und Command-Style-Eingaben in Slack werden mittels API Gateway an die Lambda-Funktionen weitergeleitet. Natürliche Sprache wird mittels AWS Lex verarbeitet. AWS Lex ist ein Service zur Erstellung von Konversationsschnittstellen unter Nutzung von Deep Learning und somit bestens zur Implementierung eines Chatbots geeignet. Lex übernimmt auch die komplette Verwaltung der Kompensation und stellt selbstständig Rückfragen an den Fragesteller.

Die Einbindung von GitHub erfolgt über Webhooks und das API. Etwas komplizierter erwies sich die Einbindung von AWS CodeBuild und AWS CodePipeline. Diese Services unterstützen keine Webhooks. Die Statusrückmeldung ist deshalb über einen CloudWatch-Alarm realisiert, der über eine SQS-Queue den Status an die Lambda-Funktionen zurückmeldet.

Lessons Learned

Auf dem Weg zum eigenen Chatbot lernte das Projektteam einige wichtige Lektionen.

Trotz aller Möglichkeiten, die Chatbots bieten, verwarf das Team viele Szenarien, weil sich bei genauer Betrachtung herausstellte, dass eine gänzliche Automatisierung möglich ist. Diese wurde dann auch umgesetzt (Automation First).

Bei der Definition der Interaktionsmodelle stellte sich als wichtiges Akzeptanzkriterium heraus, dass die Nutzer des Bots diese verstehen und verwenden können. Der Nutzen und die damit verbundene Effizienzsteigerung stehen im Vordergrund. Deshalb sind einfache Interaktionsmodelle mit UI-Elementen (z. B. Buttons) oftmals besser als natürliche Sprache (Simple Interaction).

Auch sollte das Einführen von Systemen vermieden werden, die nur dazu dienen, dass der Bot funktioniert. Es ergibt beispielsweise keinen Sinn, zum Beispiel zu Microsoft Teams zusätzlich Slack einzuführen, da am Ende sämtliche Informationen auf mehrere Chat-Tools verteilt sind und die Situation sich verschlechtert anstatt sich zu bessern (Existing Tools First).

Wenn Mandantenfähigkeit in der Architektur berücksichtigt werden muss, sollte von vornherein bedacht werden, ob sie von Beginn an oder erst später nötig ist (Multi-tenancy first).

Auch in einer ChatOps-Umgebung stehen weiterhin Menschen im Mittelpunkt. Deshalb sollte darauf geachtet werden, den Change aktiv zu gestalten. Nicht umsonst werden ChatOps auch als eine Kultur bezeichnet, und eine Kultur lässt sich nicht per Dekret einführen. Widerstände sind so gut wie sicher: sei es, dass sich Entwickler zunehmend überwacht fühlen oder, dass eine prinzipielle Ablehnung besteht. Diese Widerstände müssen aufgenommen und auch im weiteren Verlauf einer ChatOps-Einführung adressiert werden.

Fazit

Die Einführung einer ChatOps-Umgebung oder die Weiterentwicklung einer DevOps-Umgebung bedeutet mehr als die Installation von ein paar Bots.

Die oben aufgeführten Entscheidungen können aber eine erfolgreiche Einführung begleiten. Im Projekt wurde bis jetzt noch keine Entscheidung als komplett falsch widerrufen.

Nicht außer Acht sollten auch die kulturellen Aspekte gelassen werden. Die schönste DevOps-Umgebung nützt nichts, falls sie nicht genutzt wird. Bis jetzt hat das Projektteam allerdings durchweg positive Erfahrungen mit dem Bot gemacht. Entwicklung und Betrieb von Applikationen mit der DevOps-Umgebung konnten in großem Maße beschleunigt werden.

Literatur und Links

[AWS] Amazon Web Services, https://aws.amazon.com
[Han16] J. Hand, ChatOps, O'Reilly Media Inc., 2016
[HUB] HUBBOT, https://hubot.github.com/
[MBF] Microsoft Bot Framework, https://dev.botframework.com/