Was Entwickler über Beobachtbarkeit wissen müssen

von Chris Engelbert,

Developer Advocate bei Instana

Inhalt

01

Einführung

02

Beobachtbarkeit – Terminologie

03

Verteiltes Tracing

04

Leistungsstarkes Engineering

05

Beobachtbarkeits-gesteuerte Entwicklung (Observability-driven Development, ODD)

06

Open-Source … sein oder nicht sein

07

Unser Leben leichter machen

01 Einführung

Als Entwickler können wir uns noch an die Zeit erinnern, als Nagios technologisch ausgereifte Technologie war. Seitdem ist der Begriff "Überwachung" mit einigen schlechten Assoziationen behaftet. Dashboards voller verschiedener Diagramme und Zahlen haben uns nie wirklich dabei geholfen, Probleme zu lösen; dennoch sollten wir sie uns immer ansehen. Natürlich sagten uns diese Dashboards, dass etwas nicht in Ordnung war, aber unser Job war ja nicht die Systemausführung, oder? Schon nach kurzer Zeit stellte sich bei uns etwas wie eine Metrikmüdigkeit ein.

Warum also hat Nagio für uns Entwickler nicht besser funktioniert? Abgesehen von den Leistungsmetriken, die direkt von den in den Verarbeitungsablauf für Anwendungscode integrierten Zählern erfasst wurden, ging es dabei mehr oder weniger um eine reine Überwachung von Funktionseinheiten, meist bestehend aus Pings zu Hosts und Verbindungen zu Services – in der Regel begleitet von einer Log-Regex-Analyse.

Erschwerend kommt hinzu, dass früher bei Serviceoperationen Entwicklungs- und Operations-Teams strikt voneinander getrennt agierten. Wir hatten Datenbankadministratoren, die alle anderen gezielt davon abhielten, auch nur eine Select-Anweisung selber auszuführen.

Das Operations-Team, das für den Betrieb des System in der Produktion, die Bearbeitung von Systemausfällen und das Weiterleiten von Informationen an die Entwicklungsabteilung zuständig war, hatte es sich zur Aufgabe gemacht, Entwickler von Produktionssystemen fernzuhalten. Im besten Fall ließen sie nur eine sehr begrenzte Anzahl von Entwicklern auf diese Systeme zugreifen.

Schließlich gab es da auch die Entwicklungsabteilung. Unsere Aufgabe war es, Funktionen zu entwickeln, sie zu "testen" und das war's. Als Nächstes durften wir dann erst wieder bei der Fehlerbehebung ran. Für uns war Quellcode eine Kunst für sich. Natürlich haben auch wir jeden, der kein Entwickler war, von unserem eigenen Bereich ferngehalten.

Die Bedienung und Überwachung des Systems war einfach nicht Teil unseres Jobs. Bis auf einen bestimmten Bereich, nämlich das Hinzufügen einiger Überwachungsmechanismen in unseren schönen Quellcode – was fast so war, als würde man die Sixtinische Kapelle mit Aluminium auskleiden. Wir hatten also unsere guten Gründe, es einfach nicht zu mögen. Es war wie ein Buschfeuer.

Die Verantwortlichkeiten waren geteilt und lange Zeit schrieben Software-Engineers einfach Code und gingen dann nach Hause. An diesem Punkt übernahm dann das Operations-Team und musste sich mit allen anfallenden Problemen auseinandersetzen.

Doch die Zeiten haben sich geändert. Heute sehen Systeme anders aus, Bereitstellungen funktionieren anders und die Grenzen zwischen Teams sind aufgeweicht oder sogar vollständig aufgehoben. Das heißt, dass wir als Entwickler nun näher an den Unternehmensprozessen sind als je zuvor und ein indirekter Teil der Systeme sind.

Mit der Komplexität moderner Systeme, auf denen Mikroservices ausgeführt werden, wird die "aussagefähige Überwachung" ein wichtiger Teil unserer Arbeitswelt. Um eine Überflutung mit Metriken, wie wir sie in der Vergangenheit erlebt haben, zu verhindern, müssen wir uns mit den Vorteilen der Beobachtbarkeit beschäftigen.

Dieses E-Book schaut hinter der Kulissen dieser neuen Welt. Wir lassen dabei alle unguten Gefühle zur Überwachung hinter uns und wagen unsere ersten Schritte in die Welt der Beobachtbarkeit und ihrer immer größeren Bedeutung für Entwickler.

02

Beobachtbarkeit – Terminologie

Sehen wir uns als Erstes die grundlegende Terminologie zur Beobachtbarkeit an. Um tiefer in die Welt der Beobachtbarkeit einzutauchen, ist es wichtig, die Unterschiede zwischen Überwachung und Beobachtbarkeit zu verstehen und zu wissen, wie Daten dem Benutzer dargestellt werden.

Monitoring

In der Vergangenheit lag der Fokus der Überwachung auf Zeitreihenmetriken. Der Prozess war immer der gleiche: Zuerst wurden verschiedene Metriken erfasst, diese dann in Diagramme in Dashboards eingefügt, anschließend wurden für einige dieser Metriken Alerts eingerichtet und schließlich Schwellenwerte für die Alertausgabe ausgewählt. Dieser Ansatz war zwar besser als nichts, aber bei weitem nicht ideal, da entweder zu viele und zu wenige Alerts ausgegeben wurden, ein falsches Gefühl von Sicherheit vermittelt wurde und nicht gerade wenig Zeit auf den Fehlerbehebungsprozess verwendet wurde.

Auch wenn wir den Allgemeinzustand und die Leistung von Services als Bestandteil der Überwachung nehmen, verfolgt die traditionelle Überwachung dennoch einen symptombasierten Alertansatz. Die Probleme sind die Symptome, auf die die Überwachung reagiert. Wenn ein externer Service nicht mehr erreichbar ist, ist dies ein Symptom. Statt auf Symptome zu reagieren, sollte der Fokus auf der Erfassung relevanter Daten im entsprechenden Kontext liegen.

Beobachtbarkeit

Auch wenn Beobachtbarkeit nichts Neues ist, so ist sie doch die perfekte Ergänzung zur Überwachung. Sie kann als eine Art Überbegriff verstanden werden.

Der Begriff Beobachtbarkeit impliziert, dass wir etwas beobachten. Wichtig hierbei ist eine etwas differenziertere Betrachtung als bei einer Überwachung. Neben den ganzen Zahlen und Diagrammen als Teil der Überwachung liefert die Beobachtbarkeit den verschiedenen, zuvor getrennt agierenden Teams Informationen darüber, was für die Überwachung wichtig ist.

Niemand braucht Hunderte von Diagrammen und Tabellen, vor allem dann nicht, wenn sie keine Probleme lösen. Andererseits ist es möglich, dass die notwendigen Daten, die erforderlich sind, um ein bestimmtes Problem zu lösen, gar nicht erfasst wurden.

Zusätzlich fügt die Beobachtbarkeit verteiltes Tracing hinzu, also im Grunde einen Stack-Trace für Mikroservices. Wir werden darüber sprechen, was das bedeutet. Und nicht zu vergessen: Aussagefähige Protokollanalysen gehen weit über einfache, auf regulären Ausdrücken basierende Fehlersuchen hinaus.

02 Beobachtbarkeit – Terminologie

Die letzten beiden Aspekte sind so konzipiert, dass sie die Überwachung aus Anwendungen und Services heraus ermöglichen. Im Gegensatz zur einfachen Überwachung werden Informationen direkt aus dem Service erfasst und mit dem gesamten Wissen über das System zusammengeführt.

Beobachtbarkeit ist mit dem Wissen um bekannte Fehlerdomänen unseres Systems konzipiert. Beispielsweise kann für einen Service, der sich über HTTP mit einem anderen Service verbindet, diese Verbindung aus verschiedenen Gründen fehlschlagen; dies ist eine bekannte Fehlerdomäne. Fehlerdomänen können davon abhängen, wie der Service oder das System als Ganzes wahrgenommen wird. Aus diesem Grund ist es wichtig, bei der Gestaltung der Fehlerdomänen und dem damit verbundenen Kontext unterschiedliche Perspektiven (z. B. nach Zielgruppe) für ein leichtes Debugging zu berücksichtigen.

Die Beobachtbarkeit wird von drei Grundpfeilern gestützt, die Einblicke und Erkenntnisse für unser Problem liefern: Metriken zu Allgemeinzustand und Leistung, verteilte Traces und Protokolle.

Beobachtbarkeit im Unternehmen
Im Unternehmen muss Beobachtbarkeit hoch skalierbar sein, um der Komplexität und Skalierbarkeit transienter Systeme mit kurzlebigen Mikroservices oder serverlosen Anwendungen gerecht zu werden. Systeme, die für die Beobachtbarkeit in Unternehmen konzipiert sind, halten automatisch Schritt mit der sich ständig ändernden Infrastruktur oder Serviceumgebung. Teil dieses Prozesses ist es, neue Instanzen oder Services zu erkennen oder Instanzen oder Services zu beenden, und zwar ganz ohne manuellen Eingriff. Normalerweise werden Wrapper-Bibliotheken für nicht invasive, minimale Änderungen am Quellcode bereitgestellt, manchmal sogar bis hin zu einer vollautomatischen Codeinstrumentalisierung, um Mess- und Statussonden zu einer bereits aktiven Anwendung hinzuzufügen.

Eine Codeinstrumentalisierung bietet eine Möglichkeit, die notwendigen Start- und Endpunkte von Operationen in die laufende Codebasis einzufügen, ohne diese manuell dem Quellcode hinzuzufügen. Das bedeutet, dass die Entwickler von der mühsamen Aufgabe befreit sind, Elemente zur Überwachung und Nachverfolgung in ihre Codebasis einzufügen, wodurch mehr Zeit für die eigentliche Arbeit an den Anwendungsfällen des Unternehmens bleibt.

Darüber hinaus liefert das hoch verfügbare (jedoch mit geringem Aufwand verbundene) Code-Profiling für Entwicklungs-, Staging- und Produktionssysteme umfassendere Einblicke in die Leistung von Services unter Live-Bedingungen – etwas, das Entwicklern lange Zeit gefehlt hat. Systeme haben sich in Produktionsumgebungen immer anders als erwartet verhalten.

Nach der Erfassung all der Daten ist es jedoch wichtig, diese nicht nur auf die herkömmliche Art in Form von Zahlen, Tabellen und Diagrammen darzustellen. Beim Blick über den Tellerrand wird klar, dass es der Kontext der Daten ist, der wichtig für das Lösen von Problemen ist, insbesondere für Entwickler.

Um die riesigen Datenmengen nutzbar zu machen, müssen Beobachtbarkeitslösungen für Unternehmen automatische Korrelationsfunktionen umfassen, was zu einem besseren Verständnis und umsetzbaren Informationen führt. Diese Funktionen helfen oft auch bei der Ursachenanalyse, indem sie den notwendigen Kontext zu den korrelierten Ereignissen und Informationen liefern.

03 Verteiltes Tracing

Verteilte Traces sind für verteilte Systeme das, was Stack-Traces für Anwendungen und Ausnahmen oder Panics sind. Es handelt sich um ein Verfahren zur Erfassung und zeitlichen Bedienung von Handlern und internen Aufrufen, während sich eine Anforderung ihren Weg durch eine Systemlandschaft zum Generieren einer Antwort bahnt. Aus diesem Grund wird dieses Verfahren manchmal auch als verteiltes Anforderungs-Tracing bezeichnet.

Verteiltes Tracing hilft Entwicklern, Anforderungsflüsse zu analysieren und die Ursache von Problemen oder Leistungsengpässen zu ermitteln.

Angenommen, ein Benutzer ruft einen Benutzerservice auf und fragt seine eigenen Kontodaten ab. Der Service selbst ruft einige Services weiter unten im Stack auf, um verschiedene Arten von Informationen abzurufen. Abbildung 1 zeigt den grundlegenden Ablauf des Aufrufs. Obwohl Informationen, wie z. B. wer wen aufruft, sichtbar sind, fehlen wichtige Daten (wie beispielsweise Timing-Informationen) im Diagramm.

Dieses Szenario zeigt, wo verteiltes Tracing ins Spiel kommt. Die Abbildungen 2 und 3 zeigen den Aufrufablauf auf zwei verschiedene Arten: als Timing-Balken und als Stack-Trace-ähnliche Baumstrukturanzeige.

Diese beiden Ansichten stellen die notwendigen Informationen für Entwickler bereit, um langsame Aufrufe, lange laufende Operationen, fehlgeschlagene Services und den Ort dieser Ausfälle zu finden.

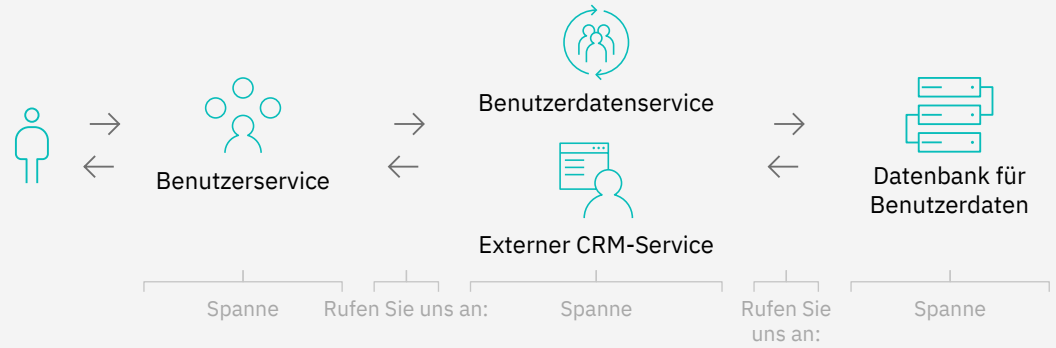


Abbildung 1. Eine Anforderung an einen Benutzerservice, der sich entlang der internen Services bewegt

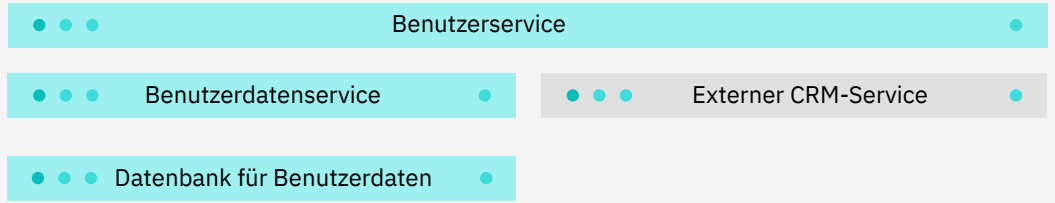


Abb. 2. Der Aufrufablauf als Timing- und Hierarchiediagramm

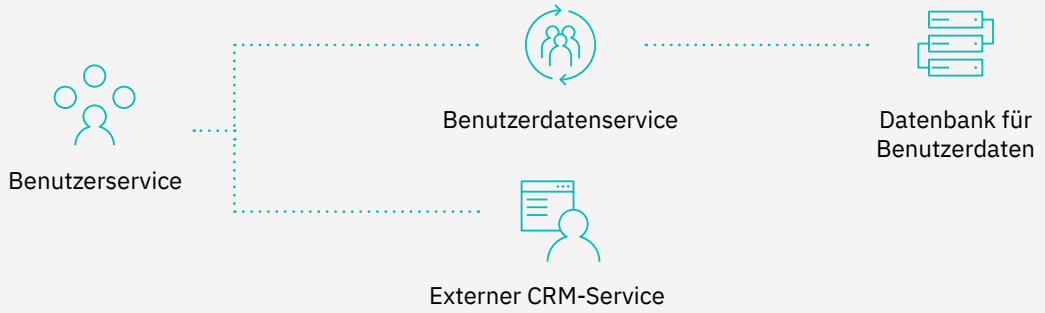


Abb. 3 Hierarchiediagramm des Codeflusses

03 Verteiltes Tracing

Zur Darstellung dieser Informationen ist der Trace in Elemente, wie Spannen und Aufrufe, unterteilt.

Eine Spanne, manchmal auch als Zeitspanne bezeichnet, stellt die Laufzeit einer einzelnen Operation innerhalb des Verarbeitungsablaufs des verteilten Trace dar. Eine Operation in diesem Kontext ist eine Codeausführung mit einer Startzeit und einer Endzeit sowie möglichen über- und untergeordneten Spannen.

Zusätzlich enthält eine Spanne Metadaten, um den Kontext für die tatsächliche Spanne selbst bereitzustellen. Um mehrere Spannen zu verbinden, werden Kennungen ('span-id' und 'trace-id') zur Erstellung der über-/untergeordneten Hierarchien verwendet.

Die Kommunikation zwischen zwei Spannen wird durch Aufrufe dargestellt. Aufrufe enthalten Informationen darüber, welche Art von Verbindung verwendet wurde, Header-Informationen und Antwortdaten, wie HTTP-Statuscodes, wenn ein Aufruf fehlerhaft war. Auch angepasste Kontextinformationen können hinzugefügt werden.

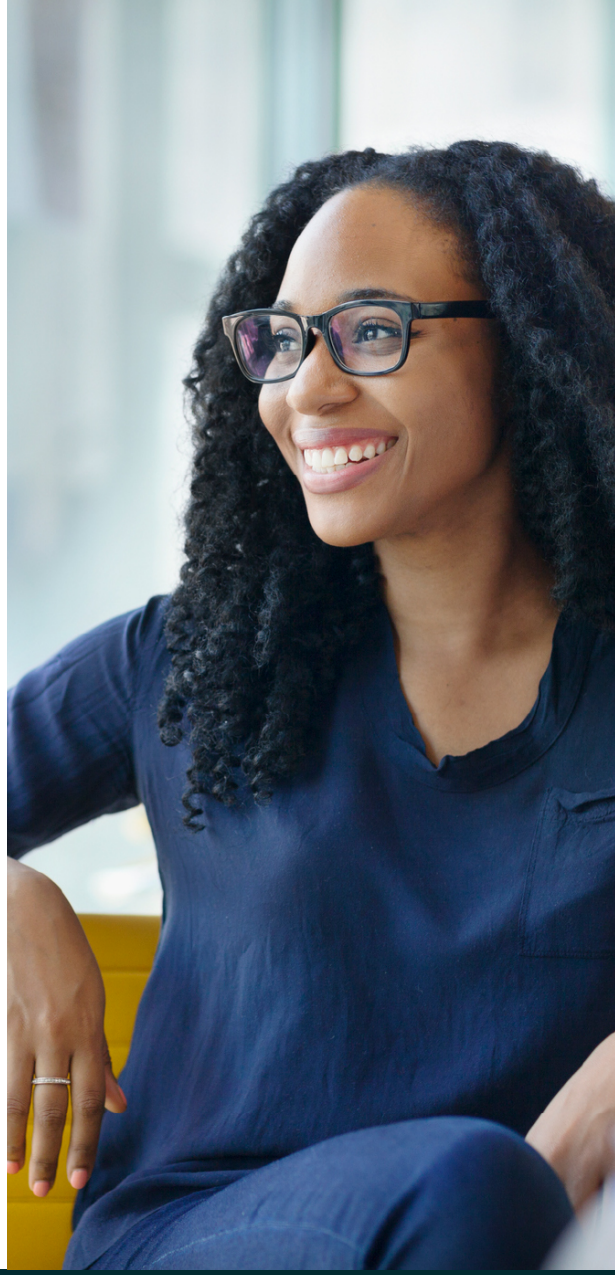
Die Wichtigkeit der Beobachtbarkeit

Bei der Arbeit mit verteilten Systemen sind die Herausforderungen ganz anders als bei den monolithischen Anwendungen, die wir in der Vergangenheit erstellt haben. Obwohl wir uns unsere Welt gerne ganz einfach vorstellen, sieht die Realität ganz anders aus – manchmal auch ohne unser Wissen.

Viele verschiedene Systeme und Services werden unabhängig voneinander und gleichzeitig ausgeführt. Die meisten Services haben noch einen oder mehrere Unteraufrufe zu anderen Services oder Datenbanken. Das Verständnis des Wechselspiels zwischen diesen Services ist eines der wichtigsten Elemente für einen Entwickler, wenn er versucht, einen Fehler zu beheben oder ein Leistungsproblem zu mindern.

Es ist auch wichtig, nicht zu viele Diagramme mit Metriken in Dashboards zu haben; das würde uns dahin zurückführen, wo wir vor Jahren waren. Was wir suchen, sind relevante Informationen, die uns schnellstmöglich zur Ursache eines Problems führen. Das System muss uns zeigen, welche Metriken für die Fehlerdomäne jedes einzelnen Service wichtig sind. Unser Fokus auf die wirklich wichtigen Dinge ist das, was hier zählt.

Und dieser Fokus macht Sinn, denn es gibt keine Garantie, dass wir am Ende mit einem fehlerfreien System dastehen, ganz gleich, wie viele Metriken wir hinzufügen. Es ist eher umgekehrt: Fehler in unserem System können nicht vermieden werden. Ich habe hierzu einen Blogbeitrag mit dem Titel Building Resilient Applications - Embrace the Failure (nur auf Englisch verfügbar) geschrieben, in dem im Detail auf dieses Thema eingegangen wird. Das Beste, was wir tun können, ist, uns auf diesen Fall vorzubereiten, indem wir das Prinzip der bereits erwähnten Fehlerdomäne verstehen.



03 Verteiltes Tracing

Am Ende ist ein wichtiger Punkt für uns als Entwickler, uns wieder auf unsere Kreativität und Motivation zu besinnen. Wir wollen an den Teilen arbeiten, die das Unternehmen voranbringen: unseren Anwendungsfällen. Wir wollen unseren Quellcode nicht mit Millionen von Metriken oder Tracing-Punkten überladen. Und das Wichtigste: Wir wollen keine ständige Brandbekämpfung oder Fehlerbehebung.

Abschließend hierzu sei aber gesagt, dass ein unverfälschtes, hilfreiches und Einblick verschaffendes Beobachtbarkeitssystem bei all diesen Aufgaben helfen kann. Durch die Bereitstellung von Slim-Wrapper-Bibliotheken oder, noch besser, einer automatischen Instrumentierung können wir unsere Quellbasis ganz auf den Geschäftsanwendungfall abstimmen. Wir erhalten auch Zeit zurück – Zeit, die wir für die Analyse komplexer Probleme in verteilten Systemen aufgewendet haben, Zeit, die wir auf das Hinzufügen von Metriken verwendet haben, Zeit, die wir brauchten, um die Interaktion und Kommunikation zwischen Systemen zu verstehen und nicht zuletzt Zeit, die wir auf der Suche nach Engpässen verschwendet haben. Heute kann uns ein verteilter Trace helfen, Unzulänglichkeiten und Entwicklungen des aktuellen Systems zu verstehen, bevor es zu Engpässen oder Skalierbarkeitsproblemen kommt.

Zeit bis zur Problemlösung

Wenn eine Bereitstellung fehlschlägt oder unsere neue Version sich unberechenbar verhält, müssen wir schnell handeln. Entweder führen wir einen Rollback auf eine ältere Version durch oder analysieren das Problem, finden ein Fix, implementieren diesen und machen einfach weiter..

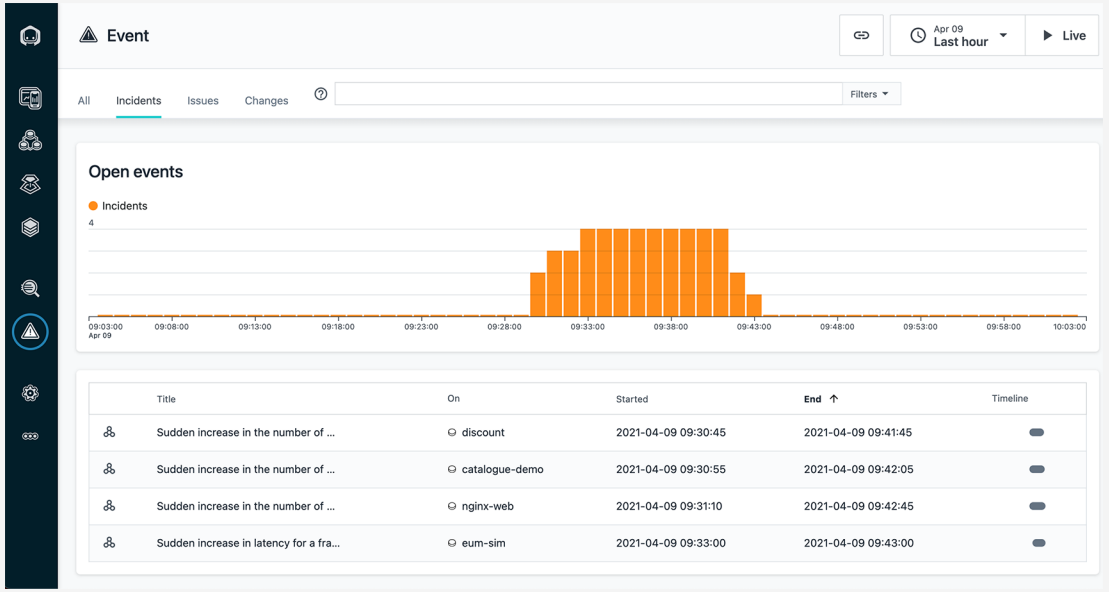


Abbildung 4. Schneller Einblick in den Status eines Service mit Release-Markern. Quelle: Instana.

03 Verteiltes Tracing

Manchmal rührt das Problem tatsächlich nur aus einer kleiner Unachtsamkeit her und kann lokal reproduziert und schnell behoben werden. Die ganze Sache wird jedoch komplexer, wenn wir das Problem nicht direkt lokal reproduzieren können. In diesem Fall können sich Beobachtbarkeitstools als große Hilfe erweisen. Tatsächlich lässt sich nicht vorhersagen, wie komplex die Probleme sind, die in unseren Produktionsumgebungen auftreten können. Wegen dieser fehlenden Vorhersagbarkeit müssen wir daher die erforderlichen Daten aus einer aktiven Produktionsumgebung sammeln.

Durch die Bereitstellung einer vereinfachten Darstellung des Anforderungsablaufs mithilfe eines verteilten Trace ist es leicht, Informationen darüber zu sammeln, an welcher Stelle die Anforderung fehlgeschlagen ist. Es wird außerdem deutlich, ob der Fehler im Stack nach oben gestiegen ist oder irgendwo dazwischen verarbeitet wurde und ob der Benutzer vom Fehler betroffen war.

Verteilte Traces stellen auch Informationen zu aufrufender und aufgerufener Komponente gemeinsam mit den jeweiligen Headern sowie Timing- oder Wiederholungsinformationen bereit.

Diese Informationen helfen als erster schneller Schritt bei der Ursachenanalyse und geben Aufschluss darüber, welche Kollegen möglicherweise bei der Problemlösung helfen könnten. Unfundierte Schuldzuweisungen werden so vermieden, denn Schlussfolgerungen zu vorliegenden Problemen werden auf der Grundlage von Daten gezogen.

Dieser Top-Down-Analyseansatz ist nicht nur viel schneller, sondern bietet auch die Möglichkeit, zügig die tatsächliche Ursache des Problems zu finden. Man erhält einen Überblick über alle Aspekte des Problems und ermittelt basierend auf den tatsächlichen Upstream- oder Downstream-Abhängigkeiten diejenigen, die für die Untersuchung von Interesse sind.

Kurz gesagt: Bei der Suche nach dem Problem liefert die Beobachtbarkeit die Erkenntnisse, die wir brauchen, um schnell und im besten Fall mit minimaler Abhängigkeit von anderen Teams oder externen Partnern wie Hosting-Unternehmen zu agieren. Letztere reagieren häufig bei der Untersuchung eines Problems, bei dem der Faktor Zeit eine wichtige Rolle spielt, zu langsam.

Eine schiefgelaufene Bereitstellung
Historisch gesehen könnte ich als Entwickler diesen Abschnitt überspringen. Bereitstellungen fielen früher nicht in meinen Zuständigkeitsbereich.

03 Verteiltes Tracing

Heute jedoch schreiben Entwickler Dockerfile-Konfigurationen zum Erstellen von Docker-Images und stellen häufig auch Implementierungskonfigurationen in Form von Kubernetes (K8s)-YAML-Deskriptoren bereit, die die Container, Services und mehr – oder zumindest Teile davon – beschreiben. Und das aus gutem Grund.

Es ist nicht die alleinige Zuständigkeit von DevOps-Teams, diese Deskriptoren bereitzustellen, da eine vollständige Bereitstellung meist eine Kombination der Kubernetes-Bereitstellungsdeskriptoren von den Entwicklungs- und DevOps-Teams ist.

Jedenfalls ist unsere letzte Bereitstellung fehlgeschlagen und es gibt kein offensichtliches Problem beim Bereitstellungsprozess. Trotzdem fällt der Service direkt nach der Bereitstellung aus. Und jetzt können wir endlich zeigen, was wir können.

Der erste Schritt ist, sich klarzumachen, was das Problem ist. Waren das wirklich wir? Ein Blick auf einen verteilten Trace kann diese Frage oft schnell beantworten. Ist es unser Service, der zu einem Fehler führt (z. B. HTTP-Statuscode 500), oder kommt das Problem von weiter unten im Stack? Gibt die Downstream-Serviceabhängigkeit den Fehler zurück? Vielleicht senden wir die falschen Daten. Auch auf diese Frage könnte ein schneller Blick eine Antwort geben.

Beobachtbarkeitslösungen stellen den notwendigen Nachweis, der ursprünglich von den Systemen gemeldet wurde und auf unserer Fehlerdomänenanalyse basiert, als präzise und kontextbezogene Beobachtungen bereit. Verteilte Traces sind der wichtigste Baustein, wenn es darum geht, sich schnell einen Überblick über noch unbekannte Situationen zu verschaffen. Die mittlere Reparaturzeit (MTTR) und die durchschnittliche Zeit bis zur Wiederherstellung des Service (MTRS) sind hier die wichtigsten Metriken.

Erinnern Sie sich an die Regel "Keine Bereitstellungen am Freitag"? Ich auch, und ich wette, da sind wir nicht die Einzigen. Heute würde ich behaupten, dass man mit guten Beobachtbarkeitsmechanismen und der Möglichkeit, sofort kleine Änderungen beim Verhalten, der Latenz oder Fehlerrate zu erkennen, freitags Bereitstellungen durchführen kann und sollte, jedoch vielleicht gleich frühmorgens. In diesem Fall hätte man noch einige Stunden, um ein Problem zu beheben oder einen Rollback für den Fall durchzuführen, dass etwas schiefgeht.

Wenn ein Problem nicht sofort behoben werden kann, weil wir uns zu weit vorgewagt haben, besteht die einzige Möglichkeit darin, einen Rollback auf einen früheren Build durchzuführen. Mit einem automatisierten Bereitstellungsablauf ist dieser Vorgang relativ einfach. Wir starten unsere CI/CD-Pipeline (Continuous Integration/Continuous Deliver) einfach mit einem anderen Tag oder einer anderen Release-Version und schon läuft's. Wir haben auf einmal viel mehr Zeit, um noch genauere Untersuchungen anzustellen, Probleme noch gewissenhafter zu lösen und haben keinen zusätzlichen Stress mehr, besonders schnell sein zu müssen.

04 Leistungsstarkes Engineering

Nach all den Informationen zum Beschleunigung des Entwicklungsprozesses und dazu, wie Zeit optimal genutzt werden kann, müssen wir uns mit der Entwicklung an sich befassen und damit, wie sie sich verändert oder in den vergangenen Jahren verändert hat.

Auf Seite 18 des 2019 State of DevOps Report sind vier Punkte aufgeführt, die für ein leistungsfähiges Entwicklungsteam zukünftig von Bedeutung sind. Die aufgeführten Aspekte verweisen auf die Tatsache, dass die Grenzen zwischen Engineering-Teams, Operations-Teams und Datenbankadministratoren mittlerweile aufgeweicht sind.

Aber was bedeutet diese Aussage wirklich? Werden wir Teil des Operations-Teams? Sicherlich nicht. Ich würde mich nie als Teil des Operations-Teams sehen, auch wenn ich täglich einiges an Arbeit für diesen Bereich erledige. Die Unterscheidung besteht jedoch nach wie vor. Ich bin keineswegs ein Experte in Sachen Bereitstellungs-Engineering; es gibt aber genug davon. Und diese Aussage trifft auch jedes andere Thema zu, auch beim Engineering. Nicht jeder ist ein Performance Engineer. Es ist immer zwar noch notwendig, diese hochspezialisierten Leute zu haben, aber das große Ganze zu verstehen ist wichtig für alle und bedeutsamer als je zuvor.

04 Leistungsstarkes Engineering

Die vier Fragen, die es zu beantworten gilt, sind Folgende:

- Wie häufig führen wir Bereitstellungen durch?
- Wie sieht unsere Durchlaufzeit für Änderungen aus?
- Wie schnell können wir Services wiederherstellen?
- Wie sieht unsere Fehlerrate bei Änderungen aus?

Die Antworten auf diese Fragen sagen uns, wie schnell wir Iterationen durchführen können. Unsere Antworten sind zwar eine gute Richtlinie, aber seien Sie ehrlich. Das Gleiche gilt für unsere Codebasis; eine Optimierung ist nur möglich, wenn wir das Problem verstehen.

Wie oft müssen wir Bereitstellungen durchführen?

Führen wir Bereitstellungen einmal pro Jahr, einmal pro Monat, einmal pro Tag oder mehrmals pro Tag durch? Häufigere Bereitstellungen bedeuten, dass wir bessere und schnellere Einblicke in das Verhalten neuer Versionen haben müssen.

Wie lange dauert es, bis Code live geschaltet wird?

Wie viel Zeit brauchen wir eigentlich für eine Bereitstellung? Handelt es sich um eine größere Operation, eventuell ein Release-Train zwischen allen Teams? Vielleicht sind Bereitstellungen unabhängig voneinander möglich. Oder können wir Bereitstellungen automatisch durchführen, sobald alle Tests erfolgreich waren?

Wie schnell ist eine Wiederherstellung nach einer Betriebsunterbrechung möglich?

Wer kann das Problem analysieren, wenn etwas schief geht? Meine Lieblingsfrage lautet: "Wer kann das Problem für meinen Service beheben, wenn ich im Urlaub bin, kein Telefon oder keine Internetverbindung habe und wie lange wird die Person brauchen, um das Problem überhaupt erst einmal zu verstehen?" Wie viele Bereitstellungen schlagen fehl?

Zu guter Letzt die am häufigsten gestellte Frage, die die meisten Personen wohl ohne viel Nachdenken beantworten können: Schlagen Bereitstellungen fehl und wenn ja, wie viele und warum?

Diese Liste ist ein ausgezeichneter Ausgangspunkt, sich mit dem Thema der Geschwindigkeit von Iterationen in unserem Unternehmen, Team und Service oder in unserer Anwendung auseinanderzusetzen.

Ich würde aber noch einen Schritt weiter gehen und sagen, dass eine wichtige Frage fehlt, nämlich die Frage zu den Unterschieden zwischen Releases. Es ist natürlich schön, dass nur wenige Bereitstellungen fehlschlagen und wir sind vielleicht schon an dem Punkt, dass wir mehrere Bereitstellungen pro Tag durchführen. Es gibt jedoch noch eine offene Frage: Welchen Unterschied gibt es in Bezug auf Fehler und Leistung zwischen den einzelnen Releases?

Die besten Bereitstellungen und schnellsten Iterationen sind nichts wert, wenn jedes Release schlechter abschneidet als das vorherige oder die Fehlerrate erhöht. Am Ende wollen wir doch nicht zurück zur Brandbekämpfung, oder?

Entwicklungsprozesse verändern sich

Die Art und Weise, wie wir Anwendungen entwickeln, ändert sich offensichtlich. In den letzten Jahren war der Trend nach horizontal skalierten Systemen in noch nie da gewesenen Größenordnungen unverkennbar. Abgesehen von den Lösungen für das Internet der Dinge steigt die Zahl der Nutzer, Kunden oder Personen, die wir mit unseren Entwicklungsanstrengungen zufriedenstellen möchten, immer weiter.

Viele Unternehmen befinden sich hier zwar noch in den Anfängen, es ist aber nahezu sicher, dass neue Systeme nicht mehr auf einen monolithische Ansatz ausgerichtet sind. Skalierbarkeit ist zu wichtig geworden und die vertikale Skalierbarkeit ist zu kostenintensiv und in manchen Fällen nicht realisierbar.

Mikroservices sind zwar nicht die von vielen Leuten als solche angesehene Patentlösung, definitiv aber auf dem Vormarsch. Größere Arbeiten in mehrere kleinere Arbeitsprozesse aufzuteilen und diese unabhängig voneinander bereitzustellen klingt großartig und ist für die Skalierbarkeit ein großes Plus. Das Gefühl, Teile des Systems unabhängig voneinander nach Bedarf skalieren zu können, ist unglaublich. Andererseits hat diese Skalierbarkeit jedoch auch ihren Preis, da viele Netzoperationen zwischen den verschiedenen Services hinzukommen. Operationen können Zeitlimits überschreiten, fehlschlagen oder fehlerhafte Daten zurückgeben. Wir kämpfen hier gegen eine ganz neue Art von Feind.

04 Leistungsstarkes Engineering

Außerdem erinnere ich mich an die Zeit, als Daten in relationalen Datenbanken gespeichert wurden. Das war einfach so, denn es gab keine Alternativen. Obwohl relationale Datenbanken uns erhalten bleiben werden, haben wir heutzutage viele verschiedene Systeme zur Auswahl, von einfachen Schlüssel-Werte-Datenbanken oder Dokumentspeichern über Graph- oder Zeitseriendatenbanken bis hin zu spaltenorientierten Speichern für extrem schnelle Aggregationen.

Alle diese Systeme sind unterschiedlich und alle verhalten sich auch unterschiedlich. Niemand kann Experte für alle diese Systeme sein. Das Schöne als Entwickler ist jedoch, dass wir die Chance haben, das beste Tool für den Job zu wählen und gleichzeitig neue Dinge lernen.

Und nicht zu vergessen stellen wir Systeme und Services heute anders als noch vor einigen Jahren bereit. Die Bereitstellung dedizierter Systeme macht nur noch einen geringen Anteil neuer Installationen aus. Fast jeder setzt heute auf virtuelle Maschinen. Viele Unternehmen führen ihre Bereitstellungen bereits in der Cloud oder selbstgehosteten Umgebungen mit Kubernetes oder Cloud Foundry aus. Docker und andere Container-Runtimes wie CRI-O sind auf Entwicklermaschinen kein seltenes Phänomen.

All diese Technologien sind wunderbar und bringen die Entwicklungsarbeiten auf meiner Maschine näher an die Umgebung, in der sie schließlich ausgeführt werden. Ich mochte allerdings den Satz "Es funktioniert auf meiner Maschine" noch nie, obwohl er irgendwie stimmte. Der Grund, warum ich ich ihn nicht mochte, ist ganz einfach. Es bedeutete, dass etwas im Busch war – etwas, das ich nicht sofort verstehen oder erklären konnte.

Entwicklungspipelines

Wenn wir schneller iterieren wollen, müssen wir über einen guten Support verfügen. Obwohl Einheitentests hoffentlich nichts Neues sind und grundlegende Integrationstests häufig eingesetzt werden, wird alles, was danach kommt, oft noch als zu komplex, zu kostenintensiv oder nicht "entwicklerisch" genug angesehen.

Mit kurzen Iterationszyklen und verteilten Systemen sind Integrationstests heute jedoch wichtiger denn je. Die kontinuierliche Integration auf einem System möglichst kurz vor der Einführung in eine Staging- und Produktionsumgebung ist ein Muss für schnelle Feedback-Zyklen während der Entwicklung. Dieser Zeitpunkt ist auch ideal, um erste Leistungs- und Fehlerrateninformationen zusammenzustellen. Das Ermitteln potenzieller Probleme in einem frühen Stadium in der Entwicklung eines Features kann langwierige Neuerstellungszyklen verhindern und Erwartungshaltungen schnell erfüllen – oder aber widerlegen.

Ein weiteres Teil des Puzzles ist, dass regelmäßige Arbeitslasttests am besten, wann immer möglich, automatisiert werden sollten. Die Gründe sind die gleichen wie zuvor: Kurze Feedback-Schleifen und eine frühzeitige Überprüfung von Modellen, Erwartungen und leistungsbezogenen Fragen.

Der letzte Schritt bei der Optimierung der Arbeitsschritte eines erfolgreich agierenden Entwicklungsteams ist Continuous Delivery, für die alle zuvor erwähnten Aspekte umgesetzt sein müssen. Bei diesem Schritt muss man aber auch Mut zum Scheitern haben. Mit dem Wissen, dass das Verstehen von Problemen, eine schnelle Ursachenanalyse und Implementierung von Fehlerkorrekturen nicht nur möglich ist, sondern von allen Teammitgliedern und Tools im gesamten Produktionsprozess unterstützt wird, ist eine mutige Herangehensweise automatisch viel leichter. Wir fühlen uns bei unserem Vorgehen viel sicherer.

Das heißt aber auch, dass problematische Situationen schnell erkannt werden müssen. Kontextsensitive Beobachtbarkeits-Tools müssen den Fehleranalyseprozess unterstützen und bei der schnellstmöglichen Ermittlung der Ursache eines Problems helfen. Auch sollten alle Services möglichst widerstandsfähig gegen ausfallende Abhängigkeiten sein.

Nach der Entwicklung und Übergabe einer Korrektur beginnt der Pipelineprozess mit dem Erstellen, Testen und Bereitstellen der neuen Version in der Produktionsumgebung. Kleine, unabhängige Services sind bei einem solchen Prozess leichter zu handhaben als umfangreiche monolithischen Umgebungen.

Um es abschließend noch mal einmal zu betonen: Alle Schritte sollten voll automatisiert sein, um die Auswirkungen einer fehlgeschlagenen Bereitstellung oder unterbrochenen Versionierung zu minimieren.

04 Leistungsstarkes Engineering

Und wir sind live

Gleich nach der Live-Schaltung einer Bereitstellung war die wichtigste Frage nach "Funktioniert es?" die Frage "Wie sieht es mit der Leistung aus?"

Wie bereits erwähnt, ist ein schneller Bearbeitungszyklus nur dann aussagefähig, wenn neue Releases in Bezug auf die Leistung nicht allgemein schlechter sind als alte Versionen. Ausnahmen von dieser Regel gibt es, aber es sollte nur ein im Voraus bekannter und erwarteter Leistungsverlust sein, also etwas, das bereits in eine mögliche Infrastrukturskalierung einkalkuliert ist.

Wichtige Schlüsselmetriken, die man direkt nach einer Bereitstellung im Blick haben sollte, sind jede Art von unerwarteten Änderungen bei der durchschnittlichen Latenzzeit, Fehlerraten oder nachgeschalteten Aufrufen, zum Beispiel die Anzahl von Datenbankaufrufen oder ähnliches. Großartige Beobachtbarkeitstools bieten für einen leichten Zugriff direkte Vergleichsmöglichkeiten vor und nach der Bereitstellung.

Unsere Bereitstellung ist jedenfalls gut gelaufen, die unmittelbaren Zahlen sehen vielversprechend aus und das System verhält sich in den erwarteten Bahnen. Ist dies das Ende der Geschichte? Existiert unsere Bereitstellung glücklich bis an ihr Lebenszyklusende?

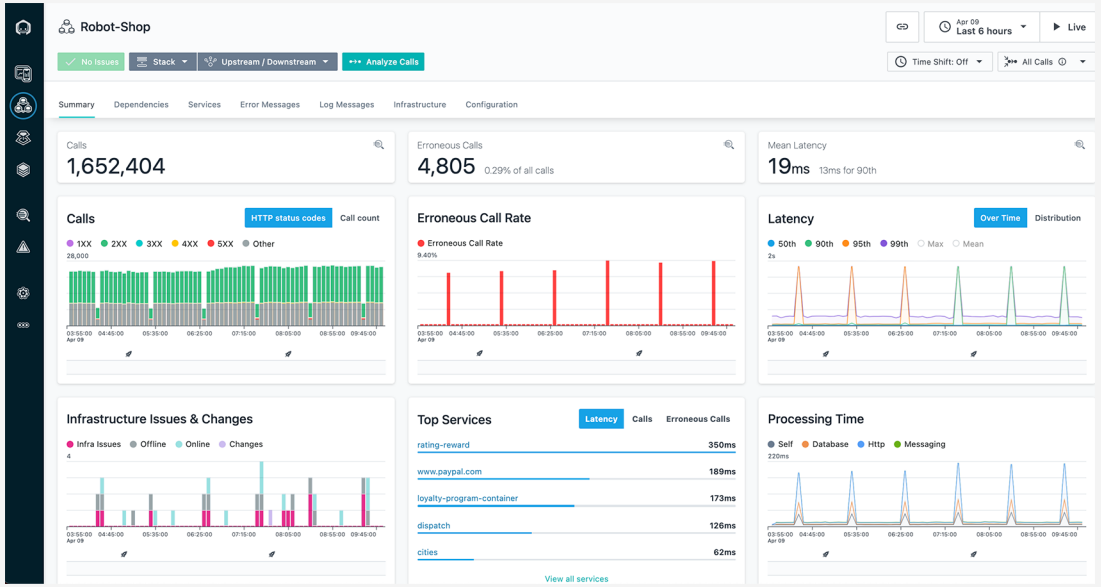


Abbildung 5. Release-Marker im Services-Dashboard für sofortige Einblicke nach und zwischen Releases. Quelle: Instana

04 Leistungsstarkes Engineering

Leider wissen wir, dass dieses Ergebnis selten für Produktionsumgebungen zutrifft. Es kann immer irgendetwas passieren. Seien Sie vorbereitet. Auch wenn die Zeiten der Brandbekämpfung um ein Vielfaches zurückgegangen sind, werden sie nie ganz verschwinden. Denken Sie daran: Ein unfehlbares System zu erstellen ist praktisch unmöglich.

Es stimmt, dass es an diesem Punkt im Prozess weniger direkte Beteiligung des Entwicklungsteams gibt, wenn etwas schief geht. Um es noch einmal auf den Punkt zu bringen: Wenn unsere Unterstützung benötigt wird, hilft uns ein verlässlicher Überblick über alle beteiligten Komponenten, Netze, Maschinen, Anwendungen und deren Verbindung untereinander bei einem schnellen Einstieg in die Problemfindung. Es ist eine Aufgabe, die mit jeder Art von Abstraktion oder architektonischer Komplexität, die wir dem System hinzufügen, nur noch komplizierter wird. Was uns bei der Entwicklung das Leben vereinfacht und erleichtert, erhöht die Komplexität bei der Analyse von teilweise unbekannten Systemen.

Es gibt nur eine Metrik, anhand der wir uns selbst beurteilen sollten, vor allem in Situationen, in denen wir auf Abruf bereitstehen: die MTTGBTB – Mean Time to Get Back to Bed – also die durchschnittliche Zeit, bis wir wieder ins Bett dürfen. Vielen Dank an Karthik Kumar für diese sehr wichtige Metrik.

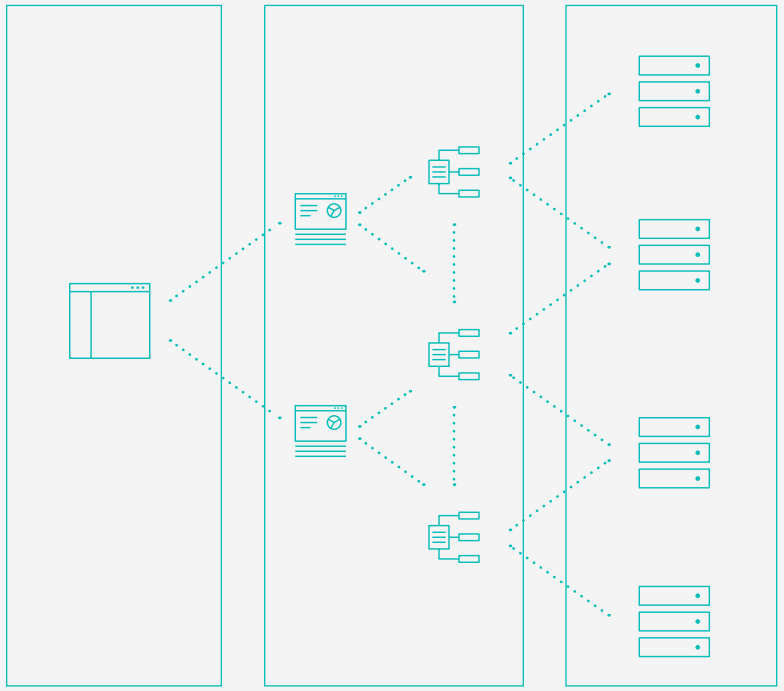


Abbildung 6. Architektonische Komplexität verhindert manchmal schnelle Schlussfolgerungen

05 Beobachtbarkeits-gesteuerte Entwicklung (Observability-driven Development, ODD)

Nachdem wir nun die Wichtigkeit von Beobachtbarkeit verstanden haben, lassen Sie uns einen Blick auf den Wert der beobachtbarkeitsgesteuerten Entwicklung (Observability-driven Development, ODD) werfen. Fassen wir schnell die wichtigsten Elemente zusammen, die als ODD beschrieben werden.

ODD ist eine direkte Erweiterung der verhaltensgesteuerten Entwicklung (Behaviour-driven Development, BDD). Sie erweitert Tests um das Verhalten einer Komponente mit Erwartungen zu Leistung und Allgemeinzustand. Die Tests sollten frühzeitig implementiert werden und während des gesamten Entwicklungszyklus durchgeführt werden. Diese Tests stellen Einblicke in die Leistung vor, während und nach der Feature-Entwicklung bereit. Sie helfen, Erwartungen in einem frühen Stadium der Entwicklung mit der Realität zu vergleichen und stellen entsprechende Informationen bereit, wenn für das Produktionssystem vor der endgültigen Feature-Bereitstellung eine Skalierung erforderlich ist.

Bauen Sie den Entwicklungszyklus um das Prinzip von kurzen Feedback-Schleifen auf. Der eigentliche Schlüssel des Ganzen ist, Beobachtbarkeit zu einem zentralen Teil des Entwicklungsprozesses zu machen. Agieren Sie vorausschauend und arbeiten Sie gemeinsam mit den anderen Teams. Grenzen Sie sich nicht voneinander ab.

Entwickeln Sie bei diesen Interaktionen Metriken zum Allgemeinzustand des Features und setzen Sie sie frühzeitig im Implementierungsprozess um. Denken Sie daran, die Anzahl der Metriken im Auge zu behalten. Die optimale Auswahl umfasst 3 bis 5, maximal 10 Metriken.

Wenn die Beobachtbarkeitslösung eine manuelle Integration mittels Wrappern erfordert, fügen Sie diese ebenfalls sofort ganz am Anfang hinzu. Wenn sich unser Service automatisch instrumentieren lässt, umso besser. Unser Code – unsere "Kunst" – bleibt davon unberührt.

Was Optimierungen anbetrifft, lassen Sie sich nicht auf Mutmaßungen ein. Um Optimierungspunkte auszuwählen, berücksichtigen Sie alle von der Beobachtbarkeitslösung bereitgestellten Daten, Metriken, verteilten Traces und Infrastrukturinformationen. Finden Sie die größten Verursacher von Latenz und Fehlerraten und beseitigen Sie diese zuerst. So etwas sorgt für schnelle Erfolge und Motivation.

Sehen Sie sich auch die Daten zum Code-Profiling an. Mit einem hoch verfügbaren Profiler auf Produktionsniveau werden wir in bisher kaum vorstellbare Bereiche vordringen. Bis vor kurzem hatte ich noch nie die Gelegenheit, ein Profil auf meinem Computer zu haben, das auch nur annähernd dem ähnelte, wie es in einer Produktionsumgebung aussehen würde. Denken Sie aber bitte daran, dass Sie nicht einfach irgendeinen Profiler verwenden sollten. Produktionsfertige Profiler sind maßgeschneidert für ihren jeweiligen Anwendungsfall, mit extrem niedrigem Systemaufwand.

Der letzte Tipp wird wahrscheinlich allein beim Lesen schon wehtun. Testen ... Warten ... in Produktion. Ja, Sie haben das ganz richtig gelesen. Testen in der Produktion, aber mit Feature-Flags. Aktivieren Sie das neue Verhalten (das neue Feature) basierend auf einer Gruppe von Benutzer-IDs, speziellen Parametern und seien Sie kreativ.

Die Überlegungen hinter diesem Schritt – und die totale Abkehr von der alten Regel, niemals Tests in der Produktion durchzuführen – ist die Einzigartigkeit jeder einzelnen Umgebung. Der Infrastrukturstatus, Abhängigkeiten, Datum und Uhrzeit, Bereitstellung, die Umgebung an sich und die Mondphase... alles kann sich auf unseren Code auswirken.

Nach all der Entwicklungsdiskussion und den Ausführungen, warum die Beobachtbarkeit den Lebenszyklus des Entwicklers erleichtert, stellt sich jetzt die Frage, wie wir diese anwenden können? Fragen Sie zunächst das DevOps-Team, ob es bereits eine Lösung hat, die von einem anderen Team eingesetzt wird.

06
Open-Source … sein oder nicht sein

Wie bei fast allem hat die Open-Source-Gemeinschaft großartige Arbeit geleistet und die Welt mit verschiedenen Optionen versorgt. Optionen sind gut, oder?

Ja und nein. Eine große Hürde bei den meisten Lösungen ist, dass sie entweder eine metrische Überwachung oder verteiltes Tracing anbieten, nicht aber beides. Das grundlegende Problem ist, dass wir den Zusammenhang zwischen den Informationen verlieren. Open-Source-Software-Tools (OSS) haben unabhängige Dashboards, unabhängige Datensilos und verfügen meist nicht über eine Möglichkeit, für eine einzelne Anforderung zwischen Metriken, Traces und Infrastrukturen zu wechseln.

Beobachtbarkeit manuell einrichten

Um Beobachtbarkeit einzurichten, werden in der Regel zwei getrennte Tools verwendet: eines für den Metrik- und Überwachungsteil (häufig Prometheus) und eines für das verteilte Tracing (häufig Jaeger oder Zipkin).

Diese Open-Source-Lösungen sind "kostenlos" verfügbar – ich werde gleich den Grund für die Anführungszeichen erläutern – und können in einen Anwendungsservice integriert werden. Integration in diesem Fall bedeutet, dass wir den Tracer und Metrik-Collector zu unserem Quellcode hinzufügen müssen. Wrapper sind für die meisten allgemeinen Bibliotheken in vielen Programmiersprachen verfügbar und können direkt verwendet werden.

Ein einfaches Beispiel für eine Prometheus-Metrik, die die Anzahl der Anfragen pro Sekunde misst, würde unter Verwendung einer Zählerinstanz zum Zählen der Anzahl der Anfragen dem folgenden Snippet ähneln. Siehe Abbildung 7.

Nicht allzu schlecht für die direkte Verwendung. Zum Glück stellen viele Frameworks diese Art von Metriken mit Prometheus ohne Vorbereitungs- oder Anpassungsaufwand bereit.

Der andere Teil ist die Verwendung von Prometheus für die Messung der Nutzung von Ressourcen wie Datenbankverbindungen oder Abfragelaufzeiten. Hier haben wir eine Vielzahl von Integrationen aus der Prometheus-Community. In der Java-Welt kann die berühmte Hibernate ORM-Lösung (ORM, Object-relational Mapping) mit nur wenig Code verwendet werden. Siehe Abbildung 8.

Die Prometheus-Integration verarbeitet alle kniffligen Details der Implementierung für uns.

Für das verteilte Tracing stehen uns Zipkin oder Jaeger zur Verfügung. Mit beiden Lösungen können wir verteiltes Tracing zu unserer Anwendung hinzufügen und auch hier ist die Integration nicht allzu kompliziert, wie das folgende Snippet, basierend auf unserem vorhandenen FooHandler, zeigt. Siehe Abbildung 9.

```
public class FooHandler {
  Counter counter = Counter
      .build()
      .namespace(“my-app”)
      .name(“foo-handler”)
      .help(“number of requests”)
      .register();

public ResponseEntity handler(Request request) {
  counter.inc(1);
  // do some business thing here
  }
}
```

Abbildung 7. Beispiel für eine Prometheus-Metrik

```
new HibernateStatisticsCollector()
   .add(sessionFactory, “my-app”)
   .register();
```

Abbildung 8. Beispiel für Hibernate mit Prometheus

06
Open-Source … sein oder nicht sein

Wir sehen, dass es immer noch nicht wirklich kompliziert ist. Es fühlt sich allerdings irgendwie so an, als ob wir uns auf dem Stand von "früher" befinden. Das Schlimme daran ist, dass unser Code nun wahrscheinlich weitaus mehr Features für die Überwachung und das Tracing enthält als tatsächliche Geschäftslogik.

Daten verstehen

Neben den in Tools wie Logstash oder Splunk gespeicherten Protokolldateien gibt es nun große Datenmengen, die man durchforsten muss, wenn ein Problem auftritt. Es gibt jedoch noch ein weiteres Problem, und zwar den Beobachtbarkeits-Stack selbst.

Bei drei unabhängigen Systemen, die keine Daten oder Datenkorrelationen teilen, ist es schwierig und zeitintensiv, die verschiedenen Datenquellen in Relation miteinander zu setzen. Das Abgleichen von Zeitstempeln, das Erkennen und Herstellen von Zusammenhängen und schließlich das Aufbauen des Kontextes, um das vorliegende Problem zu lösen, sind allesamt Sache des Benutzers.

Denken Sie daran, dass genau dies bei Open-Source-Beobachtbarkeits-Tools das größte Problem darstellt. Wir haben jetzt mehr einzelne Systeme und mehr Daten ohne den notwendigen zusammengefassten Kontext selbst.

```
public class FooHandler {
  Counter counter = Counter
        .build()
        .namespace("my-app")
        .name("foo-handler")
        .help("number of requests")
        .register();

private JaegerTracer tracer;

public FooHandler() {
  Configuration.SamplerConfiguration samplerConfig =
   Configuration.SamplerConfiguration
    .fromEnv()
    .withType("const")
    .withParam(1);

  Configuration.ReporterConfiguration reporterConfig =
   Configuration.ReporterConfiguration
    .fromEnv()
    .withLogSpans(true);

  Configuration config = new Configuration()
   .withSampler(samplerConfig)
   .withReporter(reporterConfig);

   this.tracer = config.getTracer();
  }

public ResponseEntity handler(Request request) {
   Span span = tracer.buildSpan("foo handler").start()
   try {
    counter.inc(1);
     // do some business thing here
   } catch (Exception e) {
     span.setTag("http.status_code", 500);
   } finally {
     span.finish();
   }
  }
}
```

Abbildung 9. Snippet von Java-Tracing mit Jaeger

06

Open-Source … sein oder nicht sein

Die Ist-Kosten von Open-Source-Lösungen
Erinnern Sie sich, dass ich, als ich vorhin "kostenlos" erwähnte, sagte, ich würde später noch etwas dazu sagen? Jetzt ist der ideale Zeitpunkt.

Entgegen der allgemeinen Meinung sind Open-Source-Angebote nicht kostenlos. Es stimmt zwar, dass es für ihre Nutzung keine Lizenzgebühr gibt. Der größte Kostenfaktor bei der OSS-Beobachtbarkeit ist Zeit – Zeit, die von Unternehmen bezahlte Personen damit verbringen, sich mit der Funktions- und Anwendungsweise des Beobachtbarkeits-Stacks auseinanderzusetzen und damit verbundene Daten in die richtige Relation zu setzen. Diese Zeit, oft das, von dem wir am wenigsten haben, fehlt uns dann bei der Bereitstellung funktionierender Geschäftsabläufe.

Es kann notwendig sein, zusätzliche Entwickler zu bezahlen, um auch OSS-Beobachtbarkeitslösungen anzupassen. Nicht zu vernachlässigen ist auch die Zeit, die für die Integration der Überwachung und des verteilten Tracing erforderlich ist. Diese Aufgaben sind alle zusätzliche Kosten neben der Zeit, die es braucht, um die OSS-Beobachtbarkeit in der Quellbasis zu implementieren.

Diese Integrationen mit unseren bisherig gesetzten Zielen im Einklang zu halten, kann ganz leicht zu 5 bis 10 % mehr Entwicklungszeit führen; Zeit, die wir eigentlich nicht mehr aufwenden wollten, als wir die Ära von Zabbix und Nagios hinter uns ließen.

Wenn man sich die DevOps- und Operations-Teams anschaut, erhöht sich diese Zahl nur noch, da wir jetzt mehrere Systeme betreiben, was wieder eine andere Geschichte ist. Wir sehen uns nur unsere Seite der Geschichte an.

Zu guter Letzt fallen auch Kosten für den Betrieb des Stacks, den erforderlichen Speicherplatz für alle Daten und die Verarbeitungszeit an, die für die erneute Implementierung selbst der einfachsten automatischen Korrelationen benötigt wird.

Plötzlich wurden diese "kostenlosen" Open-Source-Tools sehr teuer und weisen zudem immer noch Defizite in Bezug auf Beobachtbarkeit und Kontext auf.

Die Anbieterlösung
Kommerzielle Lösungen hingegen stellen viele, wenn nicht sogar alle, der oben genannten Features in einer einzelnen Lösung bereit. Wenn bestimmte Teile nicht direkt bereitgestellt werden, integrieren sie externe Services in einer Art und Weise, als ob diese intern gespeichert sind. Die Datenkorrelation von internen und externen Systemen ist dabei inbegriffen.

Die Korrelation von Daten zwischen allen Teilen unserer Systeme ist jedoch das wichtigste Element. Infrastruktur, Anwendungen, Services, Netz, Protokollierung… alle Daten werden so aufbereitet, dass die notwendigen Informationen leicht zusammenzustellen und zu verstehen sind.

Einige Anbieter gehen noch einen Schritt weiter und korrelieren Probleme aus verschiedenen Teilen des Systems, um umsetzbare und zielgruppenspezifischere Alerts zu generieren. Das Korrelieren aller zusammengehörigen Probleme oder Ereignisse verschafft einen noch schnelleren Einblick in das Ausmaß und die Ursache eines Vorfalls, die zugehörigen Services und ob es Auswirkungen auf den Benutzer gibt.

Darüber hinaus stellen einige Anbieter sofortige Einblicke in Änderungen vor und nach Releases bereit, wie in Abbildung 5 dargestellt. Für solche Einblicke ist die Integration mit allgemeinen CI/CD-Pipelines, DevOps-Services und Entwicklungstools erforderlich.

06
Open-Source … sein oder nicht sein

Möglichkeiten jenseits von Open-Source
Als Entwickler sind wir allerdings auf der Suche nach der Entwicklungsarbeit. Anstatt alles manuell einzurichten, stellen einige kommerzielle Lösungen eine vollautomatische Codeinstrumentierung bereit.

Erinnern Sie sich an unser Codebeispiel von vorhin und fragen Sie sich, wie dieser Code bei Verwendung einer vollautomatisierten Codeinstrumentierung aussehen würde?

Richtig, es sind keine Codeänderungen erforderlich, um Überwachungsfunktionen, verteiltes Tracing oder Leistungserfassungsmechanismen hinzuzufügen. Die Automatisierung erfasst die Services. Dabei ist es unerheblich, ob diese direkt, in einem Docker-Container oder innerhalb von Kubernetes, auf der Red Hat® OpenShift®-Plattform oder in ähnlichen Umgebungen gestartet wurden. Nach der automatischen Erfassung wird der Service während des Betriebs instrumentiert, Dashboards werden basierend auf bekannten Best Practices für den gegebenen Service oder das Framework generiert und dann kann es losgehen. Es ist kein manueller Eingriff notwendig. Was gäbe es Besseres?

Der Vorteil der automatischen Korrelation
Wie bereits erwähnt, ist der Hauptvorteil einer Beobachtbarkeitslösung für Unternehmen die Fähigkeit, Metriken für Maschinen, Infrastrukturen, Anwendungen und Services sowie Traces zu korrelieren. Verteilte Traces liefern eine Erklärung des Verarbeitungsablaufs der Anfrage, während Metriken die notwendigen Leistungspunkte bereitstellen.

Das manuelle Korrelieren ist jedoch recht umständlich. Der Hauptgrund, warum mehrere Dashboards für verschiedene Services bei Nutzern nicht so beliebt sind, ist, dass es fast unmöglich ist, irgendwelche Zeitspannen miteinander abzugleichen und den Gesamtkontext des Problems zu erfassen.

Der größte Vorteil der automatischen Korrelation sind, wie bereits beschrieben, die sofortigen Einblicke. Beim Betrachten eines Problems oder eines Vorfalls hat die Anbieterlösung bereits die ganze Ermittlerarbeit geleistet, um die wichtigen Informationen als kontextuellen Nachweis bereitzustellen und uns sofort an die richtigen Stellen zu führen.

```
public class FooHandler {
   public ResponseEntity handler(Request
request) {
      // do some business thing here
   }
}
```

Abbildung 10. Code mit voll automatisierter Instrumentierung mit Instana AutoTrace

07

Unser Leben leichter machen

Die Kosten von Anbieterlösungen
Im Gegensatz zu Open-Source-Tools erheben Anbieterlösungen nicht den Anspruch, kostenlos zu sein.

Beim Open-Source-Ansatz müssen wird das System bedienen und für Datenspeicherung und Rechenleistung bezahlen. Umgekehrt bieten Anbieterlösungen alle notwendigen Tools, um eine umfassende Unternehmensbeobachtbarkeit in einer per Hosting zur Verfügung gestellten Software-as-a-Service (SaaS)-Umgebung oder vor Ort in Situationen bereitzustellen, in denen dies eine Anforderung ist.

Da die Kosten für die Integration von Überwachung und Tracing in Services und Infrastruktur in Anbieterlösungen wegfallen, sind diese im Endeffekt aufgrund niedrigerer Betriebs- und Entwicklungskosten oft kosteneffizienter. Und lassen Sie uns nicht die Speicherkosten vergessen, die für die Speicherung aller Informationen und Daten anfallen.

Eine fundierte Entscheidung treffen
Bei der Auswahl zwischen einer Open-Source- und einer kommerziellen Lösung stellt sich die Frage, wie viel Zeit wir in die Entwicklung von nichtgeschäftsbezogenen Funktionalitäten stecken möchten, und zwar unabhängig von den Kosten, da keine der beiden Lösungen kostenfrei ist.

Bisher haben wir uns auf Entwickler konzentriert, die nur ungern Überwachung und Tracing in ihre Codierung einbauen; und das ist sicherlich die größte Gruppe. Es gibt aber auch Entwickler, die gerne tüfteln. Ich für meinen Teil betrachte mich als jemanden, der das Zeug zum Bastler hat. Ich bevorzuge es noch immer, neue Technologien auszuprobieren, statt mich mit einer Leistungsmetrik auseinanderzusetzen.

Abgesehen davon sagt mir meine Erfahrung, dass wir beim manuellen Erfassen von Daten das Wichtigste vergessen, und das nicht nur einmal. Oder wir fallen in alte Muster zurück, indem wir einfach alles Mögliche hinzufügen, anstatt uns auf unsere zuvor definierten Fehlerdomänen zu konzentrieren.

Anbieterlösungen stellen immer mehr Best Practices für automatisch erfasste Metriken bereit und verstehen Programmiersprachen, Frameworks und Serviceintegrationen. Durch diese Best Practices entfällt ein Großteil der Arbeit bei der Definition der möglichen Fehlerdomänen für neue oder unbekannte Systeme. Mit Anbieterlösungen kommt es kaum noch vor, dass uns wichtige Informationen fehlen.

07

Unser Leben leichter machen

Die heutige Softwareentwicklung unterscheidet sich stark von den traditionellen Methoden, die wir in den letzten Jahrzehnten verwendet haben. Anwendungen werden in handliche Mikroservices aufgegliedert, während Bereitstellungen auf der Basis von Docker, Kubernetes und automatischen CI/CD-Pipelines implementiert werden.

Mit all diesen Veränderungen während des Geschäftsbetriebs Schritt zu halten und Probleme zu analysieren wird zunehmend komplizierter. Neue Wege, um Einblicke in das Geschehene zu erhalten, sind wichtig, besonders dann, wenn man im Bereitschaftsdienst ist und versucht, ein Problem in einer unbekannten Komponente zu verstehen.

Die Beobachtbarkeit von Unternehmen ist das Wichtigste. Entscheiden Sie sich für den richtigen Weg; kämpfen Sie nicht gegen Ihre Beobachtbarkeitslösung.

© Copyright 2021 Instana, an IBM Company

IBM Deutschland GmbH
IBM-Allee 1
71139 Ehningen
ibm.com/de

Hergestellt in den USA
April 2021

IBM und das IBM Logo sind eingetragene Marken oder Marken der IBM Corporation in den USA und/oder anderen Ländern. Weitere Produkt- und Servicenamen können Marken von IBM oder anderen Unternehmen sein. Eine aktuelle Liste der IBM Marken finden Sie unter ibm.com/trademark.

Instana ist eine Marke oder eingetragene Marke von Instana, Inc., an IBM Company.

Red Hat und OpenShift sind Marken oder eingetragene Marken der Red Hat, Inc. oder ihrer Tochtergesellschaften in den USA und/oder anderen Ländern.

Dieses Dokument ist zum Datum seiner Erstveröffentlichung aktuell und kann jederzeit von IBM geändert werden. Nicht alle IBM Angebote sind in jedem Land, in welchem IBM tätig ist, verfügbar.

Vertragsbedingungen und Preise erhalten Sie bei den IBM Geschäftsstellen und/oder den IBM Business Partnern. Die Produktinformationen geben den derzeitigen Stand wieder. Für IBM Produkte gelten die Gewährleistungen, die in den Vereinbarungen vorgesehen sind, unter denen sie erworben werden.