Performance Measurement & Tuning einer eCommerce-Plattform

Andreas Badelt
Infosys Limited
Frankfurt a.M. / Düsseldorf

Schlüsselworte
Performance Engineering, Performance Tuning, Oracle ATG Commerce, Weblogic, LoadRunner, jvisualvm, JProfiler.

Einleitung
Das Messen und Tunen der Performance großer eCommerce-Systeme ist vielseitig und anspruchsvoll: Verschiedene Sichten auf das System müssen unter einen Hut gebracht werden, dabei sind die Anforderungen selten von Beginn an klar. Der Nutzer ist häufig ein unbekanntes Wesen, das sich in seinem Verhalten bestenfalls annäherungsweise statistisch erfassen lässt, das System ist in ein komplexes Netz anderer Applikationen und Infrastruktursysteme eingebettet.
Der vorliegende Text soll einen Überblick und Hilfestellung bei dieser Aufgabe bieten – Erfahrungen und „best practices", die meine Kollegen und ich beim Entwickeln, Testen und Tunen großer, auf ATG Commerce basierender Systeme gesammelt haben. Die Vorgehensweise ist strukturiert anhand des Six Sigma-Zyklus „Definieren – Messen – Analysieren – Verbessern – Steuern".

Wie kann Performance definiert werden?

Beteiligte Akteure und ihre Anforderungen
Die (angestrebte) Performance wird im Zusammenspiel verschiedener Akteure definiert, insbesondere:
- Die Fachabteilung liefert Mengengerüste, das Budget und damit den Rahmen für die einsetzbaren Ressourcen, und ggf. zentrale nichtfunktionale Anforderungen (Ausfallsicherheit, erwartetes Verhalten bei Server-Ausfall oder Überlast).
- User Experience-Spezialisten (häufig in den Fachabteilungen) definieren Anforderungen bezüglich Antwortzeiten und Informationen zu den Haupt-„Flows" durch die Applikation.
- Die Betriebsabteilung liefert Vorgaben zur Ressourcennutzung (z.B. Obergrenzen für CPU-Auslastung, Hauptspeicher- und Netzwerknutzung) und weitere nichtfunktionale Anforderungen.

Aufgabe der Applikationsentwicklung ist es, all diese Anforderungen miteinander zu vereinbaren bzw. bei Nichtvereinbarkeit rechtzeitig Feedback zu geben und Lösungen vorzuschlagen. Dazu muss ein weitgehend vollständiges Nutzungs-Modell erstellt werden, um sinnvolle Aussagen darüber treffen zu können, ob die Applikationen die erforderliche Last innerhalb tragen kann.

Nutzungsmodell
Ein Nutzungsmodell ist nie fertig, es wächst mit jedem Release und mit der Zeit (Nutzerverhalten). Für eine neue Applikation sollte versucht werden, Modelldaten aus vergleichbaren oder Vorgängersystemen zu erhalten, um nicht alles neu „erfinden" zu müssen. Idealerweise kann das Modell später, angereichert mit Kennzahlen aus den Tests bzw. aus der Produktion, direkt Voraussagen über Veränderungen liefern, z.B. „wie verändert sich die CPU-Last, wenn Seite X 10% öfter besucht wird".
Im Folgenden soll eine typische Herangehensweise skizziert werden. Beginnen wir mit den wichtigsten Daten, dem Verhalten der „externen" Nutzer. Sie ergeben sich aus
- Annahmen: Brainstorming mit User Experience-Verantwortlichen und der Fachabteilung. Welche relevanten Nutzer-„Flows" sind zu erwarten? Für eine noch nicht produktive Applikation bzw. für

neue Features müssen hier auch Annahmen über die statistische Verteilung getroffen werden.
- Repräsentativen Tests: Eine geschlossene Gruppe von Testnutzern wird in ihrem Nutzerverhalten beobachten. Dies setzt natürlich mindestens ausgereifte „Click Dummys“ voraus.
- Logfile-Analyse und „Web Analytics“ aus der Produktion (wenn die Applikation bereits „live“ ist): Requests und Sessions pro Tag, Dauer und Requests pro Session, Anzahl paralleler Sessions, Verteilung über Seiten/Bereiche, Verteilung über den Tag/das Jahr, „think time“ pro Seite etc.

Aus diesen Basis-Modelldaten können dann alle relevanten Nutzer-“Flows“ und ihre Verteilung in Sessions, untereinander und über die Zeit ermittelt werden. Hier werden meistens vereinfachende Annahmen getroffen, um das Modell und die daraus resultierenden Tests nicht zu komplex werden zu lassen. Auf die angenommenen oder bereits aus der Produktion stammenden Zahlen wird eine Sicherheitsreserve aufgeschlagen (meist zwischen 10 und 30%). Typischerweise wird der lasttechnisch „schlimmste“ Tag des Jahres zur Grundlage genommen.
Dazu muss auch noch die Verarbeitung berücksichtigt werden, die intern angestoßen wird (z.B. das „Publishen“ geänderter Katalogdaten oder automatische Batch-Jobs). Manchmal sind dies keine unabhängigen Zahlen, sondern vom Rest des Modells abhängige (z.B. „Bei mehr als 10 Millionen Nutzerprofilen startet der Löschvorgang und löscht ‚alte‘ Profile“).

Antwortzeiten und Langzeitstabilität
Das Nutzungsmodell definiert, welche Last das System grundsätzlich aushalten können muss. Aber über welchen Zeitraum? Und wie schnell muss es Anfragen beantworten?
Der Zeitraum ist eigentlich klar: Unendlich. Keine Betriebsabteilung möchte eine Applikation betreuen, die unter Last irgendwann zusammenbricht. Aus praktischen Überlegungen heraus (Zeitbedarf für Tests) reicht es normalerweise, im Test die Stabilität über einige Tage zu zeigen.
Die Grenze für das Laden von Seiten im Web wird heute bei höchstens 3-5 Sekunden gezogen. Diese Zeit lässt sich an verschiedenen Punkten messen. Aus UX-Sicht ist die am Application oder Web Server gemessene Antwortzeit für die Seitenanfrage eher uninteressant. Wichtiger ist, wann dem Nutzer relevante Informationen angezeigt werden (oder zumindest eine Fortschrittsanzeige), wann alle Bilder und sonstigen aus der Seite referenzierten Inhalte vollständig geladen sind, und wann die Seite wieder „aktiv“ benutzbar ist. Die Messklammer sollte zu Beginn klar definiert werden. Die erwarteten Antwortzeiten werden in „Key Performance Indikatoren“ (KPIs) festgelegt, z.B. in Perzentilen („90% der Seite müssen in maximal 2 Sekunden vollständig geladen sein, 98% in 5 Sekunden“). Diese können bei Bedarf seitenspezifisch festgelegt werden (nach Wichtigkeit oder Komplexität der Seite). Auch für asynchrone Requests sollten KPIs festgelegt werden, wenn sie die Interaktion mit dem Nutzer beeinflussen können. Wenn Backend-Systeme Einfluss auf die Antwortzeiten der Applikation haben, muss dies auch in die Definitionen mit einfließen: Akzeptable Antwortzeiten der jeweiligen Services, und was passieren soll, wenn sie „gerissen“ werden, insbesondere in Bezug auf die eigenen KPIs.

Ressourcen und Hardware Sizing
Vorgaben zur Ressourcen-Nutzung werden auch als KPIs definiert, i.d.R. CPU-Auslastung und ggf. Speicherbedarf. Häufig gibt es Nutzungsbeschränkungen für Backend-Systeme, z.B. der Connections zur DB oder Anzahl der Aufrufe eines Services pro Tag. Eine Variante sind Einschränkungen, die nur Testsysteme betreffen und deren passende Skalierbarkeit bzw. Aussagekraft einschränken.
Über unbeschränktes Hardware Sizing ließen sich die Anforderungen leicht erfüllen, wenn Skalierbarkeit gegeben ist. Für die Budgetplanung werden aber früh konkrete und belastbare Zahlen verlangt. Customization verhindert meist die Nutzung von Referenzwerten z.B. vom Hersteller. Was bleibt, ist der Vergleich mit ähnlichen Systemen und möglichst frühzeitige Tests unter realistischen Bedingungen. Diese können sich auf eine Untermenge der Transaktionen und Seiten der Applikation beschränken, die zuerst implementiert werden und eine gute Extrapolation erlauben.
Sobald das Nutzungsmodell mit tatsächlich gemessenen Kennzahlen gefüttert ist, lassen sich die Auswirkungen von Änderungen am System dann meist gut abschätzen. Die Kennzahlen kommen dabei

i.d.R. aus Tests und nicht aus der Produktion, da sich hier der Einfluss einzelner Seiten und Transaktionen gut in Isolation messen lässt.

Wie kann Performance gemessen werden?

Arten von Tests

Die oben beschriebene Vorgehensweise führt zu *Lasttests*, die das System auf die Fähigkeit hin testen, eine realistische oder auf bestimmten Annahmen beruhende Last auszuhalten. Eine Unterart dieser Lasttests sind *Stabilitätstests*, bei denen das System über einen längeren Zeitraum (normalerweise einige Tage) der definierten Last ausgesetzt wird.
Bei Lasttests können Fehler und Schwachstellen unentdeckt bleiben, die sich erst bei extremer Last oder „ungewöhnlichem" (d.h. nicht vorhergesagtem) Nutzerverhalten auswirken. Dafür sind *Stresstests* nötig. Hauptunterschied zu den Lasttests ist die fehlende „think time". Darüber hinaus basieren sie weniger auf abgestimmten Modellen, sondern auf den Erfahrungen des Entwicklungs- und Testteams – eventuell werden nur bestimmte Seiten oder andere Flows getestet.
Kapazitätstests – auch „Abrisskantentests" genannt – werden eingesetzt, wenn es nicht darum geht, eine vordefinierte Last zu testen, sondern Grenzen zu messen: Wie viele gleichzeitige Nutzer, wie viele Requests pro Minute verkraftet das System? Wie viele Aufträge können innerhalb einer Stunde verarbeitet werden? Diese Art von Tests wird normalerweise nur bei größeren Änderungen ausgeführt.
Im Folgenden wird der Fokus auf Last- und Stabilitätstests liegen.

Test-Tools

Sinnvoll sind Tools, mit denen Tests aufgezeichnet, aber auch noch angepasst werden können. Das macht das erste Erstellen eines Tests einfach, garantiert aber auch Flexibilität bzw. das schnelle Wiederverwenden von Tests für neue, aber ähnliche Situationen, statt immer neu aufzeichnen zu müssen. Wie oben erwähnt sollten die Antworten funktional ausgewertet werden können.
Wichtig ist auch, wie einfach und flexibel Test-Suites mit Ramp-Up und Ramp-Down-Phasen und Kombinationen verschiedener Test Cases definiert werden können. Anzahl der parallelen Nutzer pro Test Case, Think Time, sowie die Möglichkeit, Timings und Eingaben über Zufalls-Funktionen und Listen variieren zu können, sind hier entscheidend. Wenn das Tool hierfür keine eingebauten Features hat, sollte es zumindest möglich sein, dies selbst in den Testskripten zu implementieren.
Darüber hinaus sind die Reporting-Funktionen wichtig, wobei dies auch extern erfolgen kann.
Wir haben gute Erfahrungen mit HP LoadRunner gemacht, das alle genannten Kriterien erfüllt. Nachteile des Tools sind sicherlich die entstehenden Lizenzkosten (pro Virtual User) und die Bindung an Windows-Systeme für die Test Agents. Eine gute Open Source-Alternative ist Apache JMeter. Es gibt weitere kommerzielle und freie Tools, aber ein Vergleich würde hier den Rahmen sprengen.

Was ist bei den Tests zu beachten?

Die Tests basieren auf dem Nutzungsmodell. Sie sollten weitestgehend automatisiert sein – aus Kostengründen, aber auch für die Nachvollziehbarkeit. Allerdings lässt sich nicht alles automatisieren. Die normalen externen Requests werden i.d.R. mit einem Test-Tool automatisiert und können dann hochgradig parallel ausgeführt werden. Das Anstoßen flexibler Batch Jobs ist damit nicht immer möglich und muss ggf. manuell erfolgen. D.h. ein „Regieplan" ist nötig, der definiert, in welchen Zeiträumen welche Test Cases in welcher Parallelität erfolgen sollen und wann besondere Tätigkeiten ausgeführt werden, wie z.B. das Publishen von Katalogänderungen unter Last.
Die Tests sollten nicht nur die Zeiten messen, sondern auch die Funktionalität mit testen (eine HTTP 404-Seite kann sehr schnell sein). Wichtig für Nachvollziehbarkeit und Vergleichbarkeit ist das exakte Aufzeichnen und Archivieren aller relevanten Daten zu einem Test: Die Definition des Tests selbst inkl. exakter Start-/Endzeit und Regieplan; die Konfiguration des getesteten Systems (welche Version, welche Instanzen, welche Startup-Parameter, sonstige variable Parameter). Idealerweise relevante Systemkennzahlen, z.B. zur Netzwerk- und CPU-Auslastung, parallel laufende Prozesse, Nutzung von

DB Connection Pools etc. (dazu später mehr). Dies sollte nicht nur bei Problemen im Test erfolgen, da es hilfreiche Vergleiche zwischen erfolgreichen und fehlgeschlagenen Tests ermöglicht.
Vergleichbarkeit ist entscheidend: Um Voraussagen für die Produktion treffen zu können, muss die Lasttest-Umgebung eine möglichst exakte Kopie der Produktion sein. Aus Kostengründen wird man eine herunterskalierte Version nutzen, aber dieser Ansatz hat Grenzen: Wie wird z.B. die eine aktive Datenbankinstanz realistisch skaliert? Meist ignoriert man daraus entstehende Verfälschungen, weil sie keinen signifikanten Einfluss auf das Ergebnis haben. Das Risiko ist, dass die Test-Ergebnisse „zu gut" sind, weil z.B. die Testdatenbank viel schneller antwortet und so Bottlenecks nicht erkannt werden. Ein weiterer wichtiger Aspekt ist Failover: Die Test-Umgebung sollte immer noch das Testen aller relevanten Failover-Mechanismen unterstützen. Benchmarking der Umgebungen mittels Referenztests hilft, die Vergleichbarkeit und Aussagekraft von Tests zu verbessern.

Wie kann Performance analysiert und verbessert werden?

Ermitteln und Aufbereiten von Kennzahlen

Wichtig für die Analyse von Performance-Problemen und den Vergleich zwischen Performance-Testläufen ist das regelmäßige Protokollieren und Aufbereiten nicht nur der Testergebnisse, sondern aller relevanten Systemkennzahlen. Dies umfasst u.a.:

- Host: Speicher- und CPU-Auslastung; Netzwerk- und Disk-I/O; parallele Prozesse
- Application Server: Connection Pools (Größe, Anzahl wartender Threads); Threads / Work Manager (Weblogic); Pending Requests; parallele Sessions
- JVM: Garbage Collection-Statistiken (Zeiten, Speichernutzung, ggf. Speicher-Details)
- Applikation: Spezielle Testtransaktionen durch das komplette System („Business Activity Monitoring"); Cache-Statistiken; Ausführungs- und Laufzeiten von Batch Jobs
- Backend-Systeme: Antwortzeiten und ggf. weitere Statistiken der DB, Web Services etc.
- Test Cases: Requests/Sekunde (durchschnittlich/gesamt); Anzahl Iterationen (erfolgreich / fehlgeschlagen, getrennt nach funktionalen Fehlern und Time-Outs/Verbindungsabbrüchen o.ä.); Zeitmessungen für KPIs

Alle Kennzahlen sollten in eine Datenbank fließen. Von dort sollten sie auf Knopfdruck für variable Zeitabschnitte, mit Min-/Max- und Durchschnittswerten, graphisch aufbereitet zur Verfügung gestellt werden: Das menschliche Auge ist perfekt in der Mustererkennung. Häufig genug lassen sich Ursachen für Performance-Probleme oder nur „merkwürdige" Anstiege von Connection Pools und Request Queues direkt über zeitliche Zusammenhänge zwischen verschiedenen Kennzahlgraphen finden.

Weitere Daten und Tools für die Analyse

Bei Performance-Problemen geben die oben genannten Kennzahlen manchmal schon hinreichende Informationen zur Ursache. Zumindest hat man Anhaltspunkte für die weitere Analyse und kann nun weitere Datenquellen und Tools für eine detailliertere Analyse nutzen. Die wichtigsten werden im Folgenden betrachtet, da eine auch nur annähernd vollständige Liste den Rahmen sprengen würde.

Datenbankzugriff

Die Connection Pools sollten ausreichend dimensioniert sein, die initiale gleich der maximalen Größe, um ständiges Auf- und Abbauen von Verbindungen zu vermeiden. Beim Zugriff auf eine Oracle-DB sind Reports aus dem Automated Workload Repository eine wertvolle Informationsquelle. Hier werden z.B. langlaufende Statements oder spezielle Wait Conditions aufgelistet. Um Informationen zu allen abgesetzten DB-Statements zu erhalten, lässt sich in ATGs SQL Repositories SQL Logging aktivieren. Eine Alternative ist das Loggen im JDBC-Treiber (erfordert den Austausch des Treiber-Jars). Viele Profiling-Tools auf dem Markt bieten auch JDBC-Analyseoptionen. Manchmal schlägt hier aber Schrödingers Katze zu: Bei der Analyse mit JProfiler können z.B. manchmal SQLExceptions geworfen geworden, die im „Normalbetrieb" nicht auftreten.

Threads
Daten über das Threading-Verhalten der Applikation liefern regelmäßig oder adhoc angeforderte ThreadDumps. Normalerweise reichen freie Tools wie der TDA („Thread Dump Analyzer“) oder das darauf aufbauende ThreadLogic für die Analyse aus. Eine weitere Informationsquelle ist die Administrationsoberfläche des Application Servers (Statistiken über Threads, Request Dispatching etc.). Weblogic bietet z.B. WorkManager an, um Prioritäten für bestimmte Arten von Requests vergeben zu können. Dadurch können hochkritische „Bereiche“ (i.d.R. einzelne Web-Applikationen) vor erhöhter Aktivität oder Problemen in weniger kritischen geschützt werden.

CPU-Nutzung und Zeit
Spezielle Systemkennzahlen (z.B. über Testtransaktionen, s.o.) sollten bereits Anhaltspunkte geben, welche Bereiche besonders langsam sind. Ein ATG-spezifisches Tool, das detaillierte Statistiken zum Zeitverbrauch im Java Code liefern kann, ist der PerformanceMonitor. Die Erfassung muss allerdings in den Applikationscode integriert werden. Außerdem gibt es Probleme unter hoher Last (bis zum JVM-Absturz durch Deadlocks), so dass er hierfür nicht aktiviert werden sollte.
Weitere Details können über ein dediziertes Profiling ermittelt werden. JProfiler, JProbe, JVisualVM und Co. bieten umfassende Analyse-Möglichkeiten für eine JVM-Instanz und über Snapshots auch den Vergleich zwischen verschiedenen Zeitpunkten oder Instanzen. Mit einer sorgfältigen Auswahl der Profiling-Konfiguration lässt sich die Belastung für das zu testende System in Grenzen halten: Reines CPU-Profiling über Sampling, d.h. ohne Instrumentierung von Klassen, und mit einem nicht allzu feinen Zeitraster (z.B. 10 ms), sollte einen Lasttest nicht „gefährden“ - insbesondere wenn nur eine von mehreren Instanzen analysiert wird. Für tiefergehende Analysen kann dann immer noch die Instrumentierung eingesetzt werden, eingeschränkt auf wenige im Sampling „auffällige“ Klassen.
U.a. der JProfiler bietet auch „offline profiling“ an. Vorteil: Bei Tests kann immer ein Profiling mitlaufen, und bei Bedarf stehen die Daten direkt ohne Wiederholung des Tests zur Verfügung.

Speicher und Garbage Collection
Die o.g. Profiling-Tools bieten Unterstützung bei der Analyse von Speicherverschwendung und tatsächlichen Memory Leaks. Ein gutes Tool und auf diesen Zweck spezialisiertes Tool ist der auf Eclipse basierende MemoryAnalyzer (MAT).
Die Auswahl bzw. das Tuning der GC-Strategie kann signifikanten Einfluss auf die Systemperformance und -stabilität haben. Das Weglassen bestehender Tuningparameter bringt durch das Selbsttuning „moderner“ JVMs oft mehr als das Hinzufügen neuer. Es lohnt sich, die Strategie und bestimmte Tuningoptionen mit dedizierten Lasttestserien zu prüfen. Die von ATG (noch) geforderte Hotspot JVM arbeitet mit verschiedenen Heapbereichen für unterschiedlich alte Objekte (generational collectors). Das Tunen fokussiert sich normalerweise darauf, die aufwändigeren Aufräumarbeiten in der „old generation“ zu optimieren oder am besten ganz zu vermeiden. Für Webapplikationen wird i.d.R. der "low pause" CMS Collector für die „old generation“ zu guten Ergebnissen führen. Allerdings haben wir schon Probleme in der Langzeitstabilität mit diesem Collector erfahren. Eine Alternative ist der durchsatz-optimierende Parallel Collector, der aber die gesamte JVM für die Dauer der Collection anhält. Ein sinnvolles Tool, um das GC-Verhalten online zu beobachten, ist visualgc, das auch als Plug-In für JVisualVM existiert. Es stellt die einzelnen Speicherbereiche für „generational GCs“ im Detail und im Zeitverlauf dar, und eignet sich perfekt zum Beobachten und Tunen des GC-Verhaltens.

Caches
Caching ist natürlich eines der wichtigsten Performance-Themen. Bei ATG sind dies zunächst einmal die Repository Caches. Alle häufig gelesenen Objekte sollten gecached sein. Einen Überblick liefern die Cache-Statistiken auf den jeweiligen Repository-Seiten. Auch weitere Caches sollten zumindest einmal pro Release zu Beginn der Lasttests geprüft werden – hier haben wir schon so manche Überraschung durch ATGs „Config Layers“ erlebt, wo ein Cache unabsichtlich deaktiviert wurde, weil z.B. falscherweise die Entwicklungs-Konfiguration benutzt wurde.

Da ein nicht genutzter Cache-Platz keinen relevanten Speicherbedarf hat, sollte bei den Caches nicht gespart werden – solange es um häufig genutzte Daten geht! Lieber einen mehr als großzügigen Puffer einplanen, als durch eine größere Änderung an Daten plötzlich an das Limit zu geraten.

Load Balancer
Gute Load Balancer erledigen alles von der einfachen Round-Robin-Verteilung von Requests über komplexe Lastverteilungsmethoden hin zum automatischen Weiterleiten für Besucher ohne aktive Session auf statische „Sorry"-Seiten zum Vermeiden von Überlast. Manchmal funktionieren aber auch die teuersten Hardware-Load Balancer nicht korrekt, und dann ist es meist schwer, die Ursache zu erkennen – und sei es nur weil niemand an diese Möglichkeit glaubt. Wann immer es unerklärliche Probleme in Performance-Tests gibt, in denen einzelne Instanzen plötzlich in Überlast laufen (idealtypisch ein oszillierender Verlauf über alle Instanzen), sollte man einfach für einen Test die Load Balancer außen vor lassen, wenn das möglich ist (z.B. durch mod_wl in den Web Servern ersetzen o.ä.).

Wie kann das Erreichen und Halten der Performance gesteuert werden?

Der wesentliche Teil wurde bereits beschrieben: Das Sammeln und Archivieren von Kennzahlen und Umgebungsparametern. Sie sind mindestens einmal pro Release zu prüfen und zu vergleichen. Wenn die wichtigsten Kennzahlen aus der „black box"-Sicht stimmen (Durchsatz, Antwortzeiten, CPU-Nutzung), besteht i.d.R. kein Grund, auf die Details und in die Systeme hineinzuschauen. Performance-Tests sollten natürlich möglichst frühzeitig starten, nicht erst nach Abschluss der Entwicklungstätigkeiten. Für erste Sanity Checks reicht es schon, wenn während der Entwicklung regelmäßig eine Untermenge der gesamten Lasttests ausgeführt wird und die wichtigsten Kennzahlen (insbesondere Erfolgsraten, Antwortzeiten, CPU-Auslastung) ausgewertet werden. Dazu kann ein eigener Build aufgesetzt werden, der z.B. einmal täglich läuft. Wird dies mit Offline Profiling kombiniert, hat man bei Auffälligkeiten natürlich direkt detaillierte Vergleichsdaten. Die Test-Suite sollte dazu automatisch gestartet werden können. Das ist z.B. mit LoadRunner nicht ohne weiteres möglich. Ggf. muss man sich mit einem halb-automatischen Prozess begnügen, was dann aber schnell dazu führen kann, dass die Tests nicht häufig genug ausgeführt werden. Alternativ kann man mit einem anderen Tool wie ATGs URLHammer zumindest sehr rudimentäre Tests durchführen und die Zeiten messen. Manchmal können „Unit Performance Tests" helfen, d.h. Unit Tests, die keine Funktionalität testen, sondern die Zeit zur Ausführung einer Methode messen. Allerdings geht ihre Granularität normalerweise weit über das hinaus, was die reine Lehre für Unit Tests vorsieht, und das Datensetup spielt eine noch weit größere Rolle. Aber sparsam in zentralen Komponenten eingesetzt können sie Sinn machen. Auch wenn diese Tests nur für das Entwicklungsteam gedacht sind, machen sie nur Sinn, wenn die Ergebnisse ebenfalls für spätere Vergleiche archiviert werden können.

Entscheidend ist, möglichst frühzeitig und regelmäßig zu testen und auf Einhaltung der wichtigsten KPIs zu kontrollieren, um bei jeder Abweichung schnell reagieren zu können. Das richtige Tooling, und welche Tests im Detail für ein System sinnvoll sind, wird sich im Laufe der Zeit herausstellen.

Kontaktadresse:
Andreas Badelt
Infosys Limited
OpernTurm, Bockenheimer Landstr. 2-4
D-60306 Frankfurt a.M.

Telefon:	+49 (0) 69-269 566 100
Fax:	+49 (0) 69-269 566 200
E-Mail	andreas_badelt@ infosys.com
Internet:	www.infosys.com