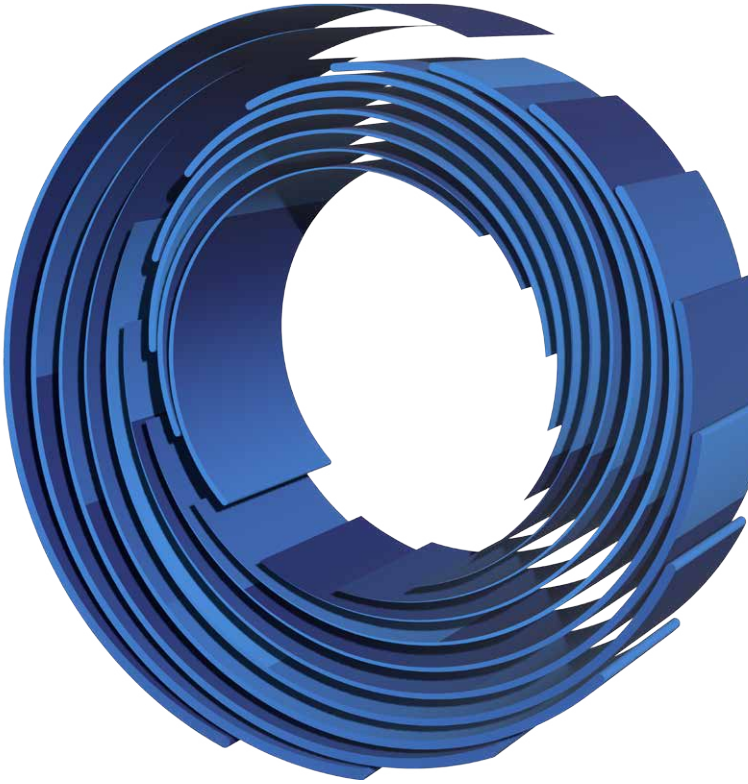
WHITEPAPER

Mit CRaC die Java-Startzeiten in Amazon EKS verkürzen

Aufwärmen statt kalt starten

von Islam Mahgoub, Raglin Anthony, Owen Hawkins und Sascha Möllering



Coordinated Restore at Checkpoint (CRaC) ist ein OpenJDK-Projekt, das einen schnellen Start und sofortige Leistung für Java-Anwendungen ermöglicht. Der Start eines neuen Java-basierten Containers leidet manchmal unter einer verlängerten Bootstrapping-Zeit, verursacht durch das Hochfahren des zugrunde liegenden Anwendungsservers und verschiedene Initialisierungsaktivitäten. Wir zeigen, wie CRaC in einer CI-Pipeline genutzt werden kann, um ein aufgewärmtes Container-Image der Anwendung zu erstellen und es in einem Kubernetes-Cluster zu deployen.

Die Anwendungsmodernisierung ist ein Schwerpunktbereich für Unternehmen verschiedener Größen und Branchen, um Geschäftsziele wie kürzere Markteinführungszeiten, höhere Kosteneffizienz und bessere Kundenzufriedenheit zu erreichen. Container und Containerorchestrierungsplattformen sind wichtige Wegbereiter dafür. Viele Kunden standardisieren Kubernetes als Containerorchestrierungsplattform und verwenden Amazon Elastic Kubernetes Service (Amazon EKS), um Kubernetes-Cluster in der AWS-Cloud und in lokalen Rechenzentren einfach zu deployen und zu verwalten.

Viele Legacy-Anwendungen, die modernisiert werden, sind in Java geschrieben. Außerdem ist Java eine der beliebtesten Programmiersprachen für die Erstellung neuer Microservices, die Frameworks wie Spring Boot nutzen. Das Hochfahren eines neuen Java-basierten Containers leidet manchmal unter einer verlängerten Startzeit. Das bedeutet eine geringere Reaktionsfähigkeit bei Scale-out-Ereignissen. Die längere Startzeit wirkt sich auch negativ auf betriebliche Aktivitäten aus, wie beispielsweise das Recycling von Worker Nodes, auf denen mehrere Container gleichzeitig beendet und auf neuen Worker Nodes eingeplant werden und sich um Ressourcen „streiten“. Azul hat ursprünglich das OpenJDK-Projekt CRaC gestartet und ein JDK mit CRaC-Unterstützung veröffentlicht [1]. Eine alternative OpenJDK-Distribution mit CRaC-Unterstützung wird von Bellsoft angeboten. Im Folgenden demonstrieren wir, wie CRaC in einer Continuous Integration (CI) Pipeline mit AWS CodePipeline und AWS CodeBuild genutzt werden kann, um ein aufgewärmtes Container-Image der Anwendung zu erstellen. Anschließend stellen wir es in Amazon EKS bereit. Wir führen einige Vergleiche durch, um die mit CRaC erzielte Verbesserung der Startzeit zu zeigen.

Die Lösung im Überblick

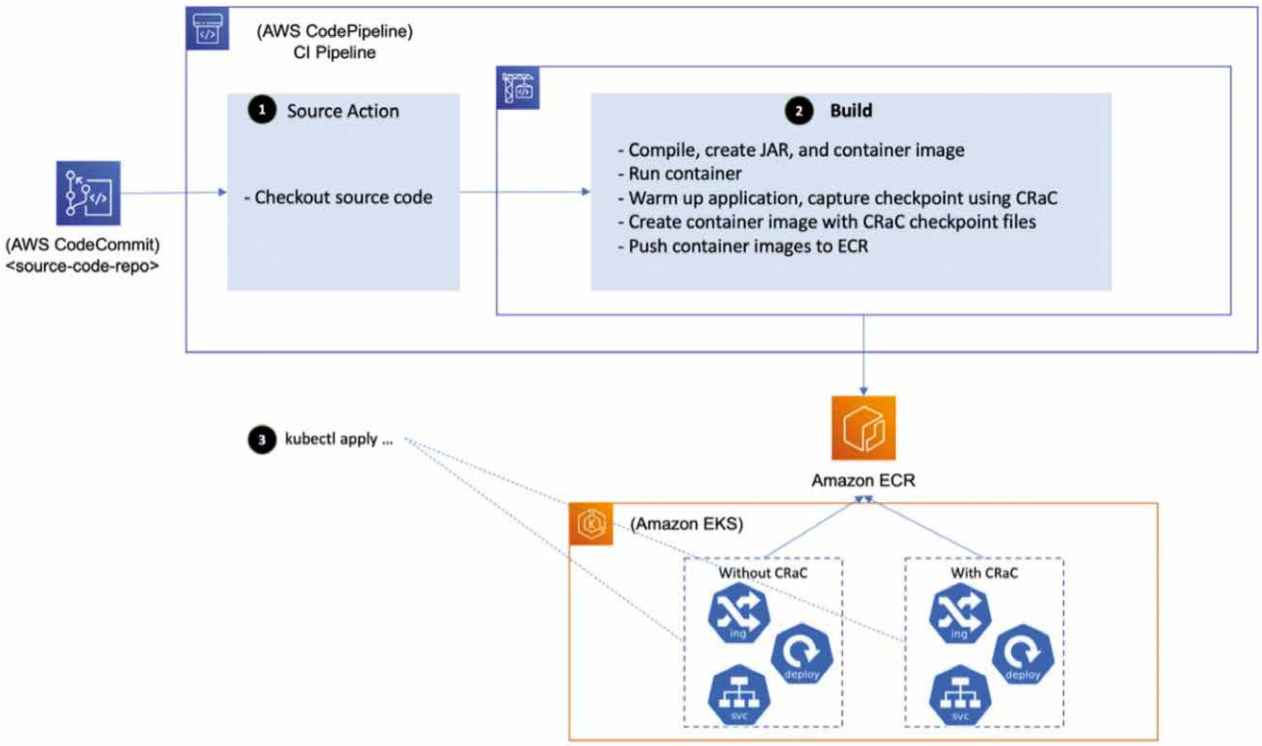
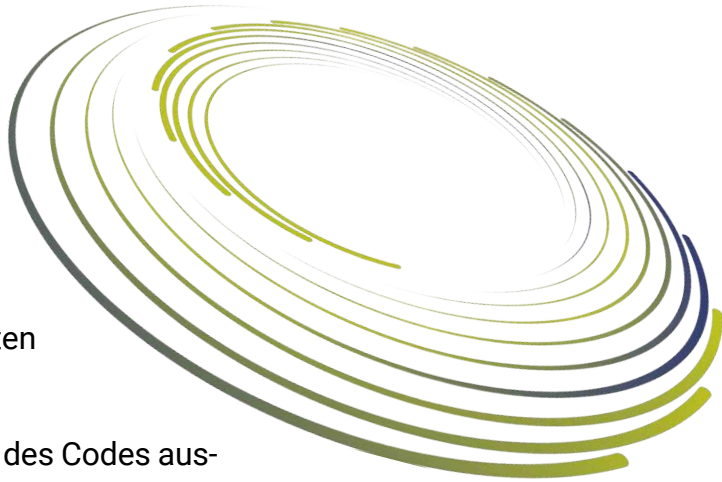


ABB. 1 Architektur der Beispielimplementierung

Das Diagramm in **Abbildung 1** zeigt die Architektur einer Beispielimplementierung für CRaC. Ihre Bestandteile sind:

- Ein Git-Repository, das den Quellcode der Beispielanwendung enthält. AWS CodeCommit, ein sicherer, hoch skalierbarer und vollständig verwalteter Versionskontrolldienst, wird für das Hosting des Repositorys verwendet.
- Eine CI Pipeline, die die verschiedenen Aktivitäten des Build-Prozesses orchestriert. CodePipeline, ein vollständig verwalteter Continuous-Delivery-Service, der Release Pipelines automatisiert, wird für den Aufbau der CI Pipeline verwendet. CodeBuild, ein vollständig verwalteter CI-Dienst, der Quellcode kompiliert, Tests durchführt und einsatzbereite Softwarepakete erstellt, führt die Build-Aufgaben aus und erstellt das endgültige Container-Image, das im EKS-Cluster deployt wird.
- Eine Container-Image-Registry, in der das Container-Image gespeichert und von der Laufzeitumgebung abgerufen wird. Amazon Elastic Container Registry (Amazon ECR) – eine vollständig verwaltete Container-Registry, die Hochleistungshosting bietet, wird für die Speicherung von Container-Images verwendet.



- Ein Kubernetes-Cluster, in dem das Container-Image deployt wird; Amazon EKS, ein verwalteter Kubernetes-Service zur Ausführung von Kubernetes in der AWS-Cloud und in lokalen Rechenzentren, wird zum Deployment des benötigten Clusters verwendet.

Die CI Pipeline wird erweitert, um die neue Version des Codes auszuführen, sie aufzuwärmen, einen Checkpoint mit CRaC zu erfassen und ein Container-Image mit CRaC-Snapshotdateien in der Container-Registry (Amazon ECR) zu veröffentlichen. Die Anwendung wird in der Zielumgebung gestartet, indem sie aus den Snapshot-Dateien wiederhergestellt wird, anstatt sie von Grund auf neu zu starten. Das führt zu einer erheblichen Verkürzung der Startzeit und vermeidet die Spitze im Verbrauch von Rechenressourcen, die normalerweise beim Start von Java-Anwendungen zu beobachten ist.

Der High-Level-Flow für den Snapshotting-Ansatz, der für Java-Anwendungen im Allgemeinen gilt, wird in diesem Beitrag skizziert. Im weiteren Verlauf dokumentieren wir einen für Spring Boot 3.2 spezifischen Ansatz und diskutieren dessen Vor- und Nachteile.

Zunächst checkt die CI Pipeline nach dem Commit einer neuen Version des Codes auf CodePipeline den Quellcode aus. Dann wird ein Build-Lauf in CodeBuild initiiert und ausgeführt, der:

1. die neue Version des Codes kompiliert, eine JAR-Datei erzeugt und ein Image erstellt, das die JAR-Datei enthält;
2. einen Container aus dem Image startet, in dem die Anwendung läuft;
3. die Anwendung durch Senden von Datenverkehr aufwärmt, der den in der Zielumgebung erwarteten Datenverkehr simuliert und anschließend einen Prüfpunkt erfasst;
4. ein Image erzeugt, das die JAR-Datei und CRaC-Checkpoint-Dateien enthält;
5. die Images zu Amazon ECR schiebt.

Schließlich werden Kubernetes-Manifeste, die die Anwendung deployen, auf EKS-Cluster angewendet, die die Zielumgebung bilden.

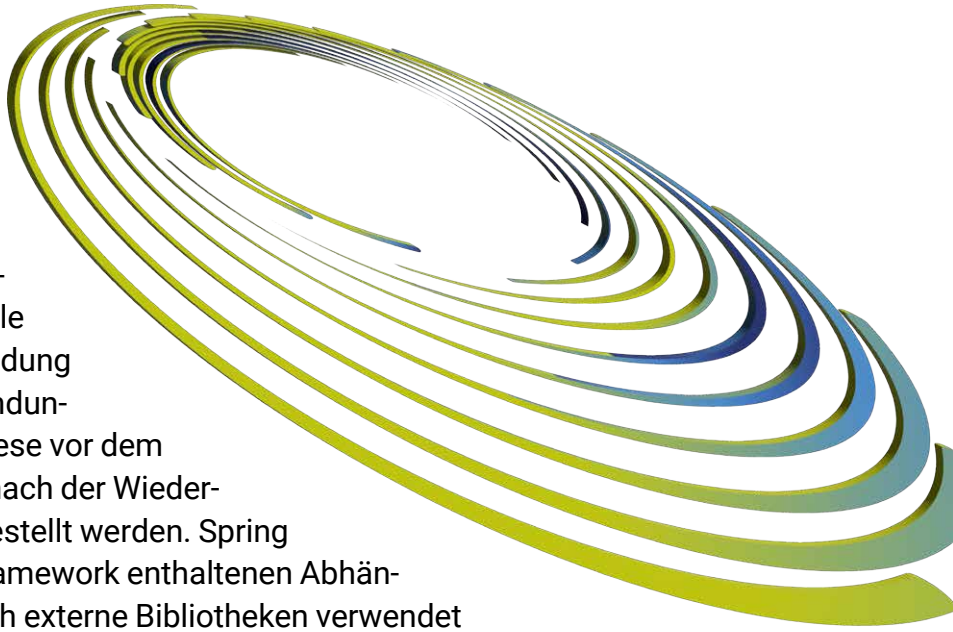
Zum Vergleich werden zwei Deployments erstellt: eins, das auf das Image ohne CRaC-Checkpoint-Dateien verweist, bei dem die Anwendung von Grund auf neu gestartet wird, und eins, das auf das Image mit CRaC-Checkpoint-Dateien verweist, bei dem die Anwendung durch Wiederherstellung aus dem erfassten Checkpoint gestartet wird.

Bei unserer Beispielanwendung handelt es sich um einen einfachen REST-basierten *Create-Read-Update-Delete*-(CRUD-)Service, der grundlegende Kundenverwaltungsfunktionen implementiert. Alle Daten werden in einer Amazon-DynamoDB-Tabelle persistiert, auf die mit dem AWS SDK für Java V2 zugegriffen wird.

Die REST-Funktionalität befindet sich in der Klasse *CustomerController*, die die *RestController*-Annotation von Spring Boot verwendet. Diese Klasse ruft den *CustomerService* auf, der die Spring-Datenrepositoryimplementierung *CustomerRepository* verwendet. Dieses Repository implementiert die Funktionalitäten für den Zugriff auf eine DynamoDB-Tabelle mit dem SDK für Java V2. Alle benutzerbezogenen Informationen werden in einem Plain Old Java Object (POJO) namens *Customer* gespeichert.

Details zur Implementierung

Wie beschrieben, verwenden wir CRaC zur Erstellung von Checkpoint-Dateien, um den Status einer „warmen“ JVM in Form von Dateien zu speichern. Dieser Zustand kann durch Einlesen der Checkpoint-Dateien wiederhergestellt werden, was zu einer deutlichen Verbesserung der Startleistung führt. Die Checkpoint-Dateien können als zusätzliche Schicht in einem Container-Image gespeichert werden. Wir haben auch andere Optionen getestet, wie das Speichern in einem Amazon Elastic Filesystem (Amazon EFS) oder in einem Amazon Simple Storage Service (Amazon S3) Bucket.



Die Persistierung des aktuellen Zustands der JVM in einer Datei hat natürlich bestimmte Auswirkungen: Sie kann beispielsweise Secrets oder andere sensible Daten enthalten. Wenn die Anwendung Dateihandler oder Netzwerkverbindungen erstellt und pflegt, müssen diese vor dem Checkpointing geschlossen und nach der Wiederherstellung ebenfalls wiederhergestellt werden. Spring Boot unterstützt das für die im Framework enthaltenen Abhängigkeiten vollständig. Wenn jedoch externe Bibliotheken verwendet werden, muss geprüft werden, ob CRaC-Unterstützung implementiert ist; wenn nicht, muss zusätzliche Programmlogik geschrieben werden.

In unserem Beispiel verwenden wir das AWS SDK für Java V2, das zum Zeitpunkt der Veröffentlichung des Beitrags noch keine CRaC-Unterstützung hatte, sodass die Verbindung zu DynamoDB nach dem Laden des Snapshots neu aufgebaut werden muss. Einen Sonderfall stellen Java-Agents dar, die in Form von JVM-Parametern beim Start der Anwendung angegeben werden. Agents werden häufig für APM-Tools verwendet, die die Anwendung automatisch instrumentieren und Metriken generieren. Sie müssen ebenfalls den CRaC-Lebenszyklus unterstützen und die entsprechenden Laufzeit-Hooks implementieren.

High-Level-Flow im Einzelnen

Der zuvor skizzierte High-Level-Flow wird in den folgenden Unterkapiteln detailliert dargestellt.

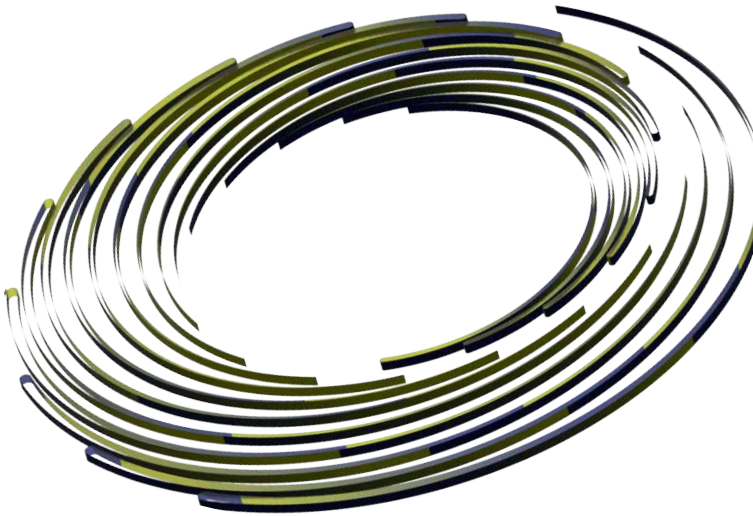
Schritt 1: Quellcode auschecken

Die CI-Umgebung (in diesem Fall CodePipeline) wird durch Commits im konfigurierten Git-Repository (CodeCommit Repo in dieser Beispielimplementierung) ausgelöst. Nach der Auslösung wird der Quellcode ausgecheckt und der nächste Schritt, die Ausführung eines CodeBuild-Projekts, wird eingeleitet.

Das CodeBuild-Projekt führt dann als „Schritt 2: Bauen“ die nun folgenden Einzelschritte aus.

Schritt 2a: Quellcode kompilieren und Container-Image erstellen

Ein mehrstufiger Container-Image-Build-Prozess wird für die Erstellung eines Container-Images mit Anwendungs-JAR verwendet. Schauen wir uns das Dockerfile, das für die Erstellung des Container-Images verwendet wird, genauer an (Listing 1).



Listing 1

```
FROM azul/zulu-openjdk:17-jdk-crac as builder

COPY ./pom.xml ./pom.xml
COPY src ./src/

ENV MAVEN_OPTS='-Xmx6g'

RUN apt-get update --fix-missing \
&& apt-get install zip curl -y \
&& curl -s "https://get.sdkman.io" | bash \
&& bash -c "source $HOME/.sdkman/bin/sdkman-init.sh; \
sdk install maven; mvn -Dmaven.test.skip=true clean package"

FROM azul/zulu-openjdk:17-jdk-crac

RUN apt-get update --fix-missing \
&& apt-get install zip curl -y \
&& curl -s "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip"
                                                    -o "awscliv2.zip" \
&& unzip awscliv2.zip \
&& ./aws/install \
&& rm -f awscliv2.zip \
&& rm -rf /var/lib/apt/lists/*
COPY scripts/* /opt/scripts/
COPY --from=builder target/${SRVC_JAR_FILE_NAME} ${SRVC_JAR_FILE_NAME}
```

Wir verwenden das OpenJDK 17 mit CRaC von Azul als übergeordnetes Image für den Build. Im ersten Schritt kopieren wir die *Pom*-Datei und den Quellcode in das Image, installieren Maven mit SDKMAN und starten den Build. Wir starten die zweite Phase des Build mit demselben übergeordneten Image, installieren einige fehlende Pakete und kopieren die notwendigen Skripte und die JAR-Datei, die wir in der ersten Phase des Build erstellt haben.

Schritt 2b: die Anwendung innerhalb der CI-Umgebung ausführen

docker run wird verwendet, um die Anwendung innerhalb der CI-Umgebung auszuführen. Die Konfigurationsparameter werden dem Container als Umgebungsvariablen des Betriebssystems über die Optionen *-env/-e* übergeben.

Ein in der CI-Umgebung verfügbares Dateisystem wird über die Optionen *-volume/-v* in den Container eingehängt und dem im Container laufenden Java-Prozess zur Verfügung gestellt, um die Checkpoint-Dateien zu speichern und sie über die Lebensdauer des Containers hinaus aufzubewahren.

Diese Beispielimplementierung basiert auf OpenJDK 17 mit CRaC von Azul, das CRIU zum Checkpointing/Wiederherstellen von Java-Prozessen verwendet. CRIU benötigt die Kontrolle über die PID für Checkpointing und Wiederherstellung. Zuvor war dafür die *CAP_SYS_ADMIN*-Fähigkeit erforderlich, aus Gründen der Sicherheit ist es aber nicht empfehlenswert, Java-Anwendungen mit den erhöhten Rechten auszuführen. Die *CAP_CHEKPOINT_RESTORE*-Fähigkeit wurde in Linux 5.9 eingeführt, um dieses Problem zu lösen (Kasten: „Hinweis Linux-Version"). Die Linux-Fähigkeiten *CHECKPOINT_RESTORE* und die *SYS_PTRACE*-Fähigkeit, die ebenfalls für Checkpointing/Restore benötigt wird, werden durch die Option *--cap-add* gewährt. Ein Beispiel für einen *docker run*-Befehl wird in Listing 2 gezeigt.

Hinweis Linux-Version

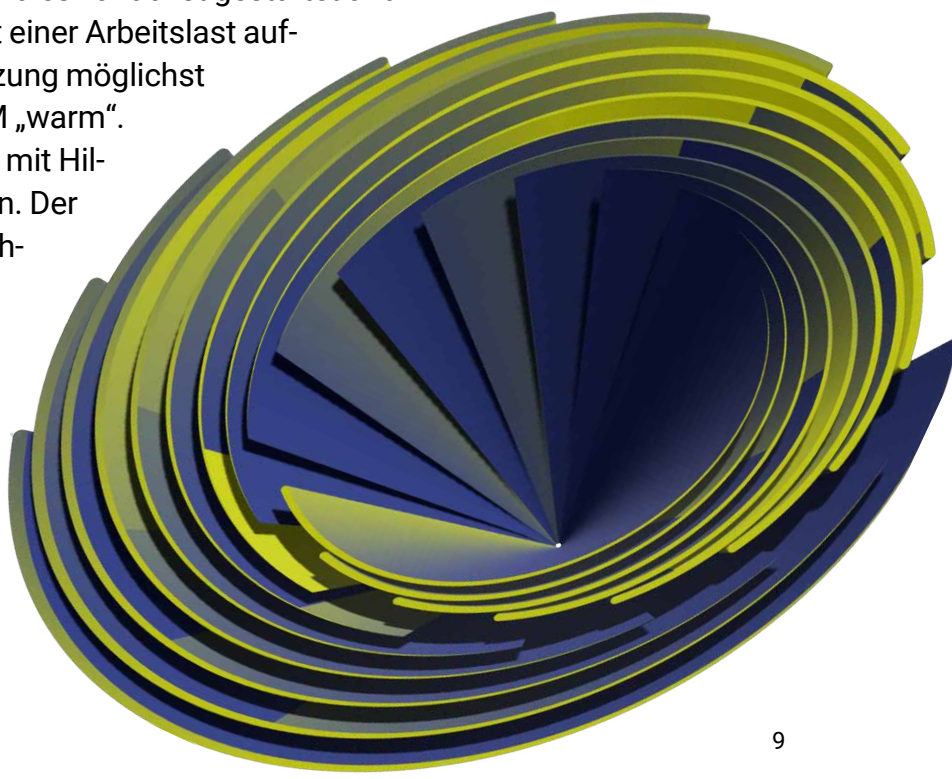
Die Systemfunktion *CAP_CHECKPOINT_RESTORE* wurde mit Linux-Kernel 5.9 eingeführt, während die CodeBuild zugrunde liegenden Instanzen mit Linux-Kernel 4.14 arbeiten. Daher mussten wir Docker innerhalb von CodeBuild im privilegierten Modus ausführen, um den Checkpoint zu erfassen.

Listing 2

```
docker run --cap-add CHECKPOINT_RESTORE \
--cap-add SYS_PTRACE \
--env TABLE_NAME=Customers \
--env ... \
--volume $PWD/crac-files:/opt/crac-files \
--rm \
--name <service-name> <service-name>:<tag> \
/opt/scripts/checkpoint.sh
```

Schritt 2c: Anwendung aufwärmen und Checkpoint erfassen

Wie schreiben wir die Snapshot-Datei in unserem Beispiel? Nachdem das Container-Image erstellt wurde, wird es zunächst gestartet und (idealerweise) die Anwendung mit einer Arbeitslast aufgewärmt, die der produktiven Nutzung möglichst nahekommt. Erst dann ist die JVM „warm". Danach wird die Checkpoint-Datei mit Hilfe von CRaC und CRIU geschrieben. Der entsprechende Abschnitt des Bash-Skripts, das diese Funktionalität implementiert, ist in Listing 3 zu sehen. Dies ist ein allgemeiner Ansatz, der für alle JVM-basierten Workloads unabhängig vom verwendeten Framework funktioniert.



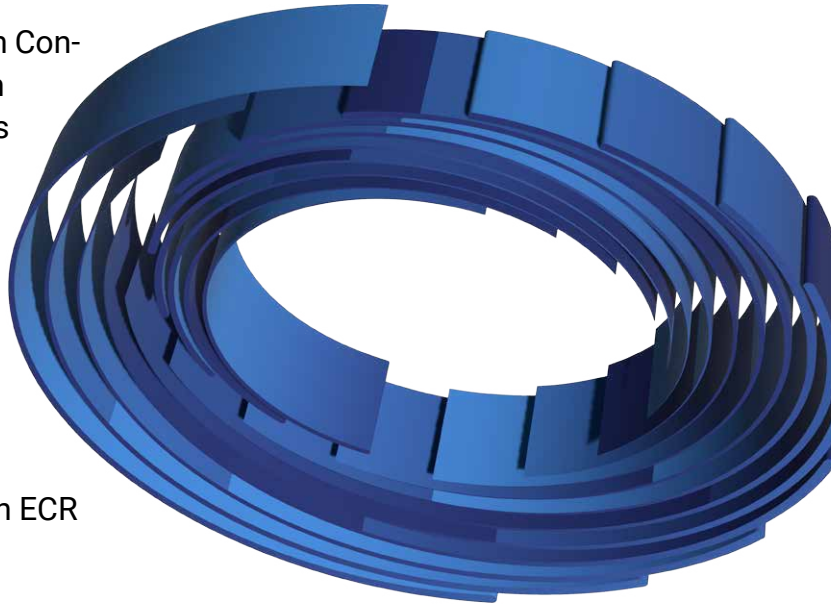
Listing 3

```
echo Starting the application...
( echo 128 > /proc/sys/kernel/ns_last_pid ) 2>/dev/null ||
          while [ $(cat /proc/sys/kernel/ns_last_pid) -lt 128 ]; do :; done;
nohup java -Dspring.profiles.active=prod -Dmode=${MODE} -Damazon.dynamodb.
             endpoint=${AMAZON_DYNAMO_DB_ENDPOINT} -XX:CRaCCheckpointTo=/opt/
                                      crac-files -jar /${SRVC_JAR_FILE_NAME} &

# ensure the application started successfully
echo Confirming the application started successfully...
sleep 30
echo nohup.out

# warm up the application
echo Warming up the application...
siege -c 1 -r 10 -b http://localhost:8080/api/customers
sleep 10

# request a checkpoint
echo Taking a snapshot of the application using CRaC...
mkdir /opt/logs/
# Waiting till the checkpoint is captured correctly
i=0

while [[ $i -lt 10 ]]
do
  echo Waiting till the checkpoint is captured correctly...
  jcmd ${SRVC_JAR_FILE_NAME} JDK.checkpoint >> /opt/logs/snapshot.log
  if ([ -f /opt/crac-files/dump4.log ]) && (grep -Fq "Dumping finished
                                    successfully" "/opt/crac-files/dump4.log")
  then
    echo Checkpoint captured!
    exit 0
    break
  fi
  sleep 10
  ((i++))
done

exit 1;
```

Dieses Bash-Skript startet die Anwendung, verwendet den Last-generator *siege*, um sie aufzuwärmen, und *jcmd*, um die Snapshot-Datei zu erstellen.

Schritt 2d: ein Container-Image erstellen, das die JAR-Datei und CRaC-Checkpoint-Dateien enthält

Jetzt befinden sich die Checkpoint-Dateien in einem Dateisystem in der CI-Umgebung. Aus dem Container-Image, das die Anwendungs-JAR-Datei enthält, wird ein neues Container-Image erstellt, dem die Checkpoint-Datei als zusätzliche Schicht hinzugefügt wurde.



Dieses Dockerfile zur Erstellung des neuen Container-Images, das die Checkpoint-Dateien enthält, wird im Folgenden dargestellt. Das erzeugte Container-Image wird zum Starten der Anwendung in der Zielumgebung verwendet, indem die Prüfpunktdateien wiederhergestellt werden:

```
FROM <container-image>
COPY crac-files /opt/crac-files
```

Zuletzt wird das AWS CLI zum Hochladen der erstellten Container-Images in Amazon ECR verwendet.

Schritt 3: Kubernetes-Manifeste für das Deployment der Anwendung in Amazon EKS verwenden

Ein wichtiger Punkt bei dem Deployment in Amazon EKS sind die Linux-Fähigkeiten, die dem Pod, in dem die Java-Anwendung wiederhergestellt wird, gewährt werden müssen. Dafür sind die Fähigkeiten *CAP_CHECKPOINT_RESTORE* und *SYS_PTRACE* erforderlich. Das YAML-Snippet in Listing 4 zeigt, wie diese Fähigkeiten dem Pod gewährt werden (die vollständigen YAML-Manifeste für das Deployment in Kubernetes können im GitHub-Repository unter [2] gefunden werden).

Listing 4

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-ddb-crac
spec:
  ...
  template:
    ...
    spec:
      ...
      containers:
        - name: spring-boot-ddb-crac
          ...
          securityContext:
            capabilities:
              add:
                - CHECKPOINT_RESTORE
                - SYS_PTRACE
            privileged: false
            runAsUser: 0
            allowPrivilegeEscalation: false
            ...
```

Änderungen in der Anwendung zur Unterstützung von CRaC

CRaC erfordert, dass die Anwendung alle offenen Dateien und Netzwerkverbindungen schließt, bevor der Checkpoint erfasst wird. Außerdem müssen möglicherweise Konfigurationen bei der Wiederherstellung aktualisiert werden, um Unterschiede zwischen der Umgebung, in der der Checkpoint erfasst wird, und der Umgebung, in der er wiederhergestellt wird, zu berücksichtigen (beispielsweise den URL der Datenbankverbindung). Um das zu erleichtern, stellt CRaC ein API zur Verfügung, das es ermöglicht, Klassen zu benachrichtigen, wenn ein Checkpoint erstellt werden soll und wenn eine Wiederherstellung stattgefunden hat. Das API bietet eine Schnittstelle, org.crac.*Resource*, die von den Klassen, die benachrichtigt werden sollen, implementiert werden muss. Es gibt nur zwei Methoden, *beforeCheckpoint()* und *afterRestore()*, die von der JVM als Callbacks verwendet werden. Alle Ressourcen in der Anwendung müssen bei der JVM registriert werden, was durch die Beschaffung eines CRaC-Kontexts und die Verwendung der Methode *register()* erreicht werden kann. Weitere Einzelheiten können in der Azul-Dokumentation [3] nachgelesen werden.

Die in dieser Implementierung verwendete Beispielanwendung interagiert mit DynamoDB über das AWS SDK. Zunächst wird ein Client erstellt, der dann für die Durchführung von Vorgängen in einer DynamoDB-Tabelle verwendet wird. Jeder Client verwaltet seinen eigenen HTTP-Verbindungspool. Um den Prüfpunkt zu erfassen, müssen die Verbindungen im Pool (Netzwerkverbindungen) geschlossen werden. Das wird erreicht, indem der Client in der Methode *beforeCheckpoint()* geschlossen und in *afterRestore()* neu erstellt wird.

Der Codeausschnitt in Listing 5 zeigt, wie die Klasse *CustomerRepository* – die das Spring-Datenrepository implementiert und für das Erstellen, Lesen, Aktualisieren und Löschen von Kundendaten in einer DynamoDB-Tabelle zuständig ist – geändert wird, um die CRaC-Anforderungen für Netzwerkverbindungen über die Schnittstelle *org.crac.Resource* zu erfüllen.

Listing 5

```
public class CustomerRepository implements Resource {
  ...
  @PostConstruct
  public void init() {
    loadConfig();
    this.client = createDynamoDbClient();
    Core.getGlobalContext().register(this);
  }

  @Override
  public void beforeCheckpoint(Context<? extends Resource> context) {
    log.info("Executing beforeCheckpoint...");
    this.client.close();
  }

  @Override
  public void afterRestore(Context<? extends Resource> context) {
    log.info("Executing afterRestore ...");
    loadConfig();
    this.client = createDynamoDbClient();
  }
  ...
}
```

Wir können hier sehen, wie wir die beiden Methoden *beforeCheckpoint()* und *afterRestore()* der Ressourcenschnittstelle implementieren. Entwickler, die bereits Erfahrung mit AWS Lambda SnapStart haben, sollten feststellen, dass diese Runtime Hooks auch zum Speichern und erneuten Laden des Zustands verwendet werden [4]. In unserem Fall schließen wir die Verbindung zu DynamoDB und stellen sie wieder her.

Konfiguration verwalten

Wird der Checkpoint in einer anderen Umgebung erfasst (beispielsweise einer CI-Umgebung) als derjenigen, in der er wiederhergestellt wird (beispielsweise einer Produktionsumgebung), und werden die Konfigurationen geladen, bevor der Checkpoint erfasst wurde, müssen die Konfigurationen als Teil der Checkpoint-Wiederherstellung aktualisiert werden, damit sie mit der Zielumgebung übereinstimmen.

Es existieren mehrere Mechanismen, die für die Konfigurationsverwaltung in Java verwendet werden können, darunter Betriebssystemumgebungsvariablen, Befehlszeilenparameter, Java-Systemeigenschaften und Konfigurationsdateien (wie *application.properties*-Dateien für Spring-Anwendungen). Die Werte der Umgebungsvariablen des Betriebssystems in einer Anwendung, die von einem Checkpoint aus wiederhergestellt wird, sind die Werte der Umgebung, in der der Checkpoint aufgezeichnet wurde. Daher wurden in dieser Implementierung Java-Systemeigenschaften und nicht Umgebungsvariablen für die Konfigurationsverwaltung verwendet.

Spring Framework bietet eine Environment Abstraction [5] zur Erleichterung der Konfigurationsverwaltung und unterstützt verschiedene Mechanismen der Konfigurationsverwaltung. Dazu gehören Betriebssystemumgebungsvariablen, Java-Systemeigenschaften und andere (Kasten: „Hinweis Umgebungsvariablen“).

Hinweis Umgebungsvariablen

Wenn die Umgebungsvariablen des Betriebssystems für die Konfigurationsverwaltung verwendet werden und über die Environment Abstraction darauf zugreifen, muss der Code nicht geändert werden, um zu den Java-System-Properties zu wechseln. Die Environment Abstraction unterstützt beide Mechanismen und gibt den Java-System-Properties Vorrang vor den Umgebungsvariablen des Betriebssystems.

Wie in Listing 5 dargestellt wird, ruft die Methode *afterRestore()* die Methode *loadConfig()* auf, die die Konfigurationen aus den Java-Systemeigenschaften über die Environment Abstraction lädt (Listing 6).

Listing 6

```
public class CustomerRepository implements Resource {
   ...
   @Autowired
   private Environment environment;
   ...
   public void createClient() {
     this.mode = environment.getProperty("mode");
     this.tableName = environment.getProperty("table.name");
     ...
   }
```

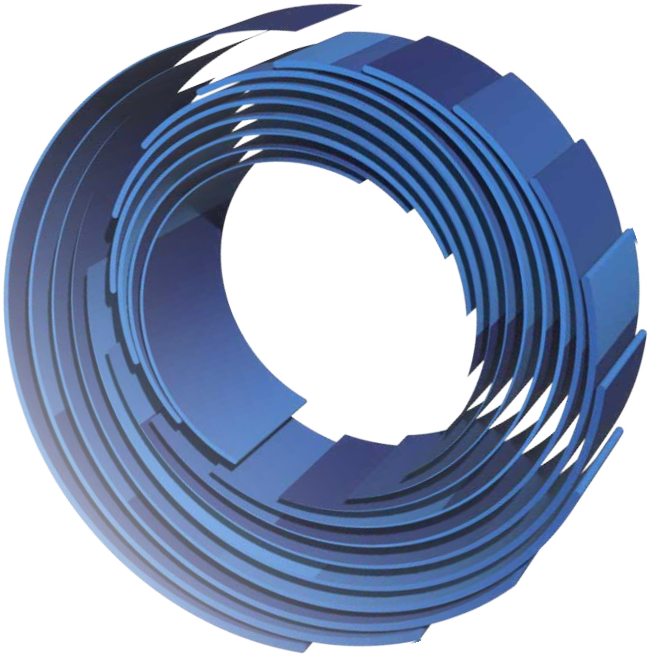
Die Java-System-Property wird in dem Befehl, der die Anwendung ausführt, mit dem Wert festgelegt, der in einer Betriebssystemumgebungsvariablen vorhanden ist:

```
java -Dspring.profiles.active=prod -Dtable.name=${TABLE_NAME}
```

Die Betriebssystemumgebung wird über das Kubernetes Deployment festgelegt, und der Wert wird aus einer *ConfigMap* abgerufen (Listing 7).

Listing 7

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-ddb
spec:
  ...
  template:
    ...
    spec:
      ...
      containers:
      - name: spring-boot-ddb
        ...
        - ...
        - name: TABLE_NAME
          valueFrom:
            configMapKeyRef:
              name: spring-demo-config
              key: table.name
```



AWS-Anmeldeinformationen verwalten

Damit die Anwendung mit DynamoDB interagieren kann, benötigt sie AWS-Anmeldeinformationen. In der CI-Umgebung (in der der Checkpoint erfasst wird) wird eine AWS-Identity-and-Access-Management-(IAM-)Rolle angenommen und die temporären Anmeldeinformationen werden der Anwendung als Java-System-Properties zur Verfügung gestellt. Daher wird *SystemPropertyCredentialsProvider* als *CredentialsProvider* verwendet. Die Zielumgebung basiert auf Amazon EKS. Daher werden IAM-Rollen für Servicekonten (IRSA) für die Interaktion mit dem AWS-API verwendet. Das erfordert die Verwendung von *WebIdentityTokenFileCredentialsProvider* als *CredentialsProvider*.

Mit dem Konfigurationsparameter *mode* wird dem Code mitgeteilt, ob er *SystemPropertyCredentialsProvider* oder *WebIdentityTokenFileCredentialsProvider* verwenden soll (Listing 8).

Listing 8

```
public class CustomerRepository implements Resource {
   ...
   @Autowired
   private DynamoDbClient createDynamoDbClient(){
     ...
     String mode = environment.getProperty("mode");
     if ("ci".equals(this.mode)) {
       return DynamoDbClient.builder()
         .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
         .build();
     }

     return DynamoDbClient.builder()
       .credentialsProvider(WebIdentityTokenFileCredentialsProvider.create())
       .build();
   }
```

Eine weitere Möglichkeit, dieses Problem zu lösen, ist die Verwendung von *DefaultCredentialsProvider*. In der CI-Umgebung (in der der Prüfpunkt erfasst wird) werden die Anmeldeinformationen als Java-Systemparameter übergeben, in diesem Fall verwendet die Standardanbieterkette für Anmeldeinformationen *SystemPropertyCredentialsProvider*. In der Zielumgebung (Amazon EKS) wird das Webidentitätstoken von Amazon EKS injiziert, daher verwendet die Standard-Credentials-Provider-Kette *WebIdentityTokenFileCredentialsProvider*. *DefaultCredentialsProvider* prüft die Konfigurationen und entscheidet, welcher Provider zum Zeitpunkt des Ladens der Klasse zu verwenden ist. Daher muss der Checkpoint vor diesem Zeitpunkt erfasst werden, damit diese Option funktioniert. Einige Kunden ziehen es vor, den Checkpoint in derselben Umgebung zu erfassen, in der er wiederhergestellt wird.

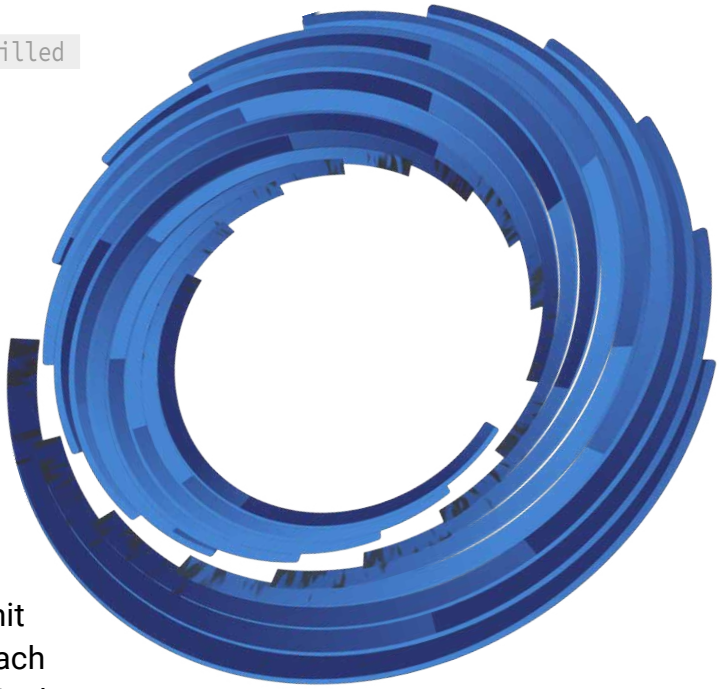
Eingebaute Spring-Unterstützung für CRaC

Spring verfügt seit Version 6.1 (und Spring Boot seit Version 3.2) über eine eingebaute CRaC-Unterstützung, was unter anderem bedeutet, dass CRaC in den Spring Lifecycle integriert ist. In der Dokumentation [6] findet sich der folgende Abschnitt: „When the *-Dspring.context.checkpoint=onRefresh* JVM system property is set, a checkpoint is created automatically at startup during the *LifecycleProcessor.onRefresh* phase. After this phase has completed, all non-lazy initialized singletons have been instantiated, and *Initializing Bean#afterPropertiesSet* callbacks have been invoked; but the lifecycle has not started, and the *ContextRefreshedEvent* has not yet been published."

Mit diesem Ansatz ist es möglich, einen Snapshot des Frameworkcodes, aber nicht des Anwendungscodes zu erstellen. Außerdem haben wir keine vollständig aufgewärmte JVM, die per Snapshot erfasst werden soll. Das bedeutet, dass die Startzeit im Vergleich zum allgemeinen Ansatz höher ist (Listing 9).

Listing 9

```
echo Starting the application...
( echo 128 > /proc/sys/kernel/ns_last_pid ) 2>/dev/null || while
                [ $(cat /proc/sys/kernel/ns_last_pid) -lt 128 ]; do :; done;
java -Dspring.context.checkpoint=onRefresh -Dspring.profiles.active=
  prod -Dmode=${MODE} -Damazon.dynamodb.endpoint=${AMAZON_DYNAMO_DB_ENDPOINT}
                -Djdk.crac.collect-fd-stacktraces=true -XX:CRaCCheckpointTo=
                                    /opt/crac-files/ -jar /${SRVC_JAR_FILE_NAME}

EXIT_CODE=$?

# Error code 137 is expected, because process is killed
if [ $EXIT_CODE -eq 137 ]
then
# let's check if there are snapshot files
   if [ -z "$(ls -A /opt/crac-files/)" ]
   then
     echo "Directory is empty, exiting with -1"
     exit -1
   fi
fi

exit 0
```



Der automatische Snapshotting-Prozess ist einfacher als der manuelle. Der Java-Prozess wird mit den entsprechenden Parametern gestartet und nach seiner Beendigung wird geprüft, ob die Dateien, die den aktuellen Zustand speichern, im Dateisystem vorhanden sind.

Um den Zustand wiederherzustellen und unsere Anwendung zu starten, können wir einfach Folgendes verwenden:

```
java -XX:CRaCRestoreFrom=/opt/crac-files
```

Checkpoint-Dateien auslagern

Einige Unternehmen möchten vielleicht darauf verzichten, Checkpoint-Dateien im Container-Image aufzubewahren, unter anderem aus folgenden Gründen:

- Verringerung der Größe des Container-Images
- um den in Form von Daten gespeicherten Zustand in Amazon ECR nicht ändern zu müssen (In-Memory-Daten für Microservices)

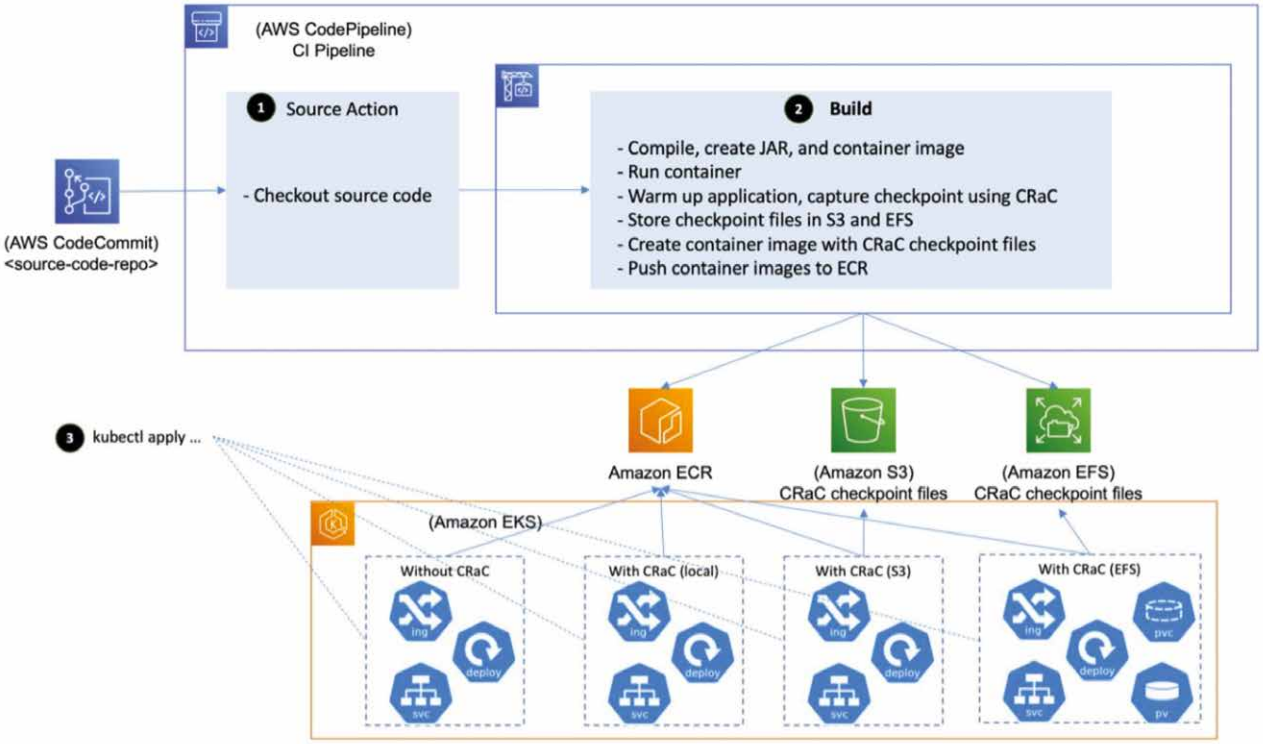
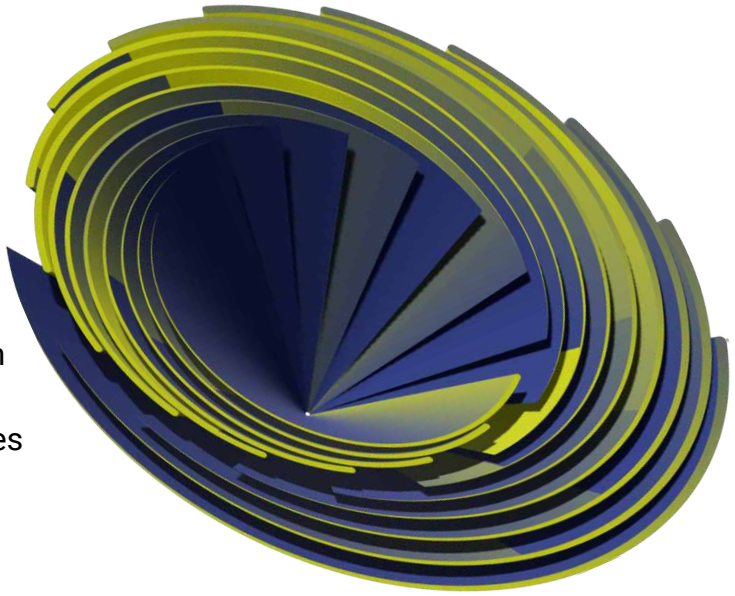


ABB. 2 Beispielimplementierung mit verschiedenen Speicher-Backends

Dazu können die folgenden Abhilfemaßnahmen in Betracht gezogen werden (**Abb. 2**):

- Speichern der Prüfpunktdateien in Amazon EFS und Einhängen in den Pod. Wir haben festgestellt, dass die Wiederherstellung des Java-Beispielprozesses zwei Sekunden dauert, wenn die Prüfpunktdateien in Amazon EFS gespeichert sind, gegenüber 0,3 Sekunden, wenn die Prüfpunktdateien Teil des Container-Image sind.
- Speicherung der Prüfpunktdateien in Amazon S3 und Synchronisierung mit dem Pod-Dateisystem zur Startzeit. Wir haben beobachtet, dass die Synchronisierungsvorgänge sechs Sekunden für Checkpoint-Dateien mit einer Größe von 170 MB auf Worker-Nodes von *m5.large* benötigen (für diese Tests wurde ein öffentlicher S3-Endpunkt verwendet).



Performanceüberlegungen

Für unsere Leistungstests haben wir verschiedene Konfigurationen getestet. Zunächst haben wir die folgenden Szenarien für den allgemeinen Ansatz getestet (der Prüfpunkt wird nach dem Aufwärmen der Anwendung durch Ausführen des Befehls *jcmd* erfasst):

- Container-Image ohne CRaC
- Container-Image mit CRaC, Checkpoint-Datei als zusätzliche Schicht im Container-Image gespeichert
- Container-Image mit CRaC, Checkpoint-Datei in Amazon EFS gespeichert
- Container-Image mit CRaC, Checkpoint-Datei in Amazon S3 gespeichert

Die Ergebnisse zeigt Tabelle 1.

Deployment	Checkpoint Files Size (MB, uncompressed)	Image Size in Amazon ECR (MB)	Time to download Checkpoint Files (Seconds)	Start-up Time (Seconds)	Total Start-up Time (Seconds)
No CRaC	–	349,97	–	12	12
CRaC – Container-Image	232	397,39 (contains CRaC files)	–	0,3	0,3
CRaC – EFS	232	349,97	–	2	2
CRaC – S3 CLI	232	463,38 (contains AWS CLI)	6	0,3	6,3

TABELLE 1 Ergebnisse der Leistungstests

Zweitens haben wir dieselben Szenarien getestet, jedoch mit dem Checkpoint, der automatisch durch die in Spring integrierte CRaC-Integration erfasst wird (der Checkpoint wird nur für das Framework erfasst, nicht für die Anwendung). Die Ergebnisse sind in Tabelle 2 zu sehen.

Deployment	Checkpoint Files Size (MB, uncompressed)	Image Size in Amazon ECR (MB)	Time to download Checkpoint Files (Seconds)	Start-up Time (Seconds)	Total Start-up Time (Seconds)
No CRaC	–	354,3	–	19	19
CRaC – Container-Image	184	389,91 (contains CRaC files)	–	2,5	2,5
CRaC – EFS	184	354,3	–	4,2	4,2
CRaC – S3 CLI	184	467,71 (contains AWS CLI)	7,5	2,5	10

TABELLE 2 Ergebnisse der Leistungstests mit Checkpoint

Kompromisse

Wie bereits angedeutet, erhöht CRaC die Komplexität des Build-Prozesses zur Erfassung von Checkpoints erheblich. Darüber hinaus sollte beim Vorwärmen der JVM darauf geachtet werden, dass ein großer Teil der Pfade im Code abgedeckt wird, um das bestmögliche Ergebnis zu erzielen. CRaC ist noch eine recht neue Technologie: Zum Zeitpunkt der Erstellung dieses Beitrags wurde sie nur von der OpenJDK-Distribution von Azul und Bellsoft unterstützt. Darüber hinaus gibt es viele Bibliotheken von Drittanbietern, die CRaC noch nicht unterstützen. Das Gleiche gilt für Java-Agenten.

Das Konfigurationsmanagement und die Verwaltung der AWS-Anmeldeinformationen sind weitere potenzielle Herausforderungen. Wie wir gesehen haben, kann eine Anpassung erforderlich sein, wenn der Prüfpunkt in einer anderen Umgebung aufgezeichnet wird als derjenigen, in der er wiederhergestellt wird.

Eine weitere Herausforderung besteht darin, dass ein Checkpoint garantiert nur auf einer CPU ausgeführt werden kann, die über die gleichen Funktionen verfügt wie die CPU, die für die Erfassung des Prüfpunkts verwendet wurde. Das liegt daran, dass CRaC eine bereits laufende JVM nicht so rekonfigurieren kann, dass sie einige der CPU-Funktionen nicht mehr verwendet, wenn sie auf einer CPU wiederhergestellt wird, der diese Funktionen fehlen. Es ist möglich, ein generisches CPU-Ziel mit *-XX:CPUFeatures=generic* für das Checkpointing anzugeben. Das bedeutet, dass die JVM nur CPU-Funktionen verwendet, die auf jeder x86-64-CPU verfügbar sind. Das kann jedoch negative Auswirkungen auf die Leistung haben. Weitere Informationen hierzu stehen in der Azul-Dokumentation [7].

Fazit

In diesem Artikel haben wir die Auswirkungen von CRaC auf die Startzeit einer Spring-Boot-Anwendung auf Amazon EKS gezeigt. Genau das gleiche Muster kann auch für Amazon ECS angewendet werden. Wir haben mit einer typischen Implementierung einer Spring-Boot-Anwendung ohne CRaC begonnen, verschiedene Konfigurationen für den Checkpoint-Speicherort hinzugefügt und die Auswirkungen gemessen. In unseren Leistungstests haben wir herausgefunden, dass die Konfiguration mit der Checkpoint-Datei, die als zusätzliche Schicht im Container-Image gespeichert wird, den größten Einfluss auf die Startleistung der Anwendung hat.

Islam Mahgoub ist Senior Solutions Architect bei AWS und verfügt über mehr als 15 Jahre Erfahrung in der Anwendungs-, Integrations- und Technologiearchitektur. Bei AWS unterstützt er Kunden beim Aufbau neuer nativer Cloud-Lösungen und bei der Modernisierung ihrer Legacy-Anwendungen unter Nutzung von AWS-Services. Außerhalb der Arbeit geht Islam gern spazieren, schaut Filme und hört Musik.

Anthony Raglin ist Solutions Architect bei AWS in London. Er arbeitet mit globalen Kunden in der Medien- und Unterhaltungsbranche zusammen und beschäftigt sich leidenschaftlich mit der Schaffung maßgeschneiderter Kundenerlebnisse und dem Vorantreiben digitaler Transformationen. Raglins Hintergrund liegt in Java-Frameworks und Microservices.

Owen Hawkins verfügt über mehr als 20 Jahre Erfahrung im Bereich Informationssicherheit und bringt in seiner Rolle als Senior Solutions Architect bei AWS umfassendes Fachwissen mit. Er arbeitet eng mit ISV-Kunden zusammen und stützt sich dabei mit seinem umfassenden Hintergrund im Bereich der digitalen Bankensicherheit. Owen ist auf SaaS und mandantenfähige Architekturen spezialisiert. Seine Leidenschaft ist es, Unternehmen zu ermöglichen, die Cloud auf sichere Weise zu nutzen. Das Lösen komplexer Herausforderungen begeistert Owen, der gerne innovative Wege findet, um Anwendungen auf AWS zu schützen und auszuführen.

Sascha Möllering arbeitet seit mehr als acht Jahren als Solutions Architect und Solutions Architect Manager bei Amazon Web Services EMEA in der deutschen Niederlassung. Seine Expertise mit den Schwerpunkten Automatisierung, Infrastructure as Code, Distributed Computing, Container und JVM teilt er in regelmäßigen Beiträgen in verschiedenen IT-Magazinen und Posts.

smoell@amazon.de

Links & Literatur

[1] https://www.azul.com/downloads/#zulu

[2] https://github.com/aws-samples/aws-eks-crac

[3] https://docs.azul.com/core/crac/crac-guidelines#implementing-crac-resource

[4] https://docs.aws.amazon.com/lambda/latest/dg/snapstart-runtime-hooks.html

[5] https://docs.spring.io/spring-framework/reference/core/beans/environment.html

[6] https://docs.spring.io/spring-framework/reference/integration/checkpoint-restore.html

[7] https://docs.azul.com/core/crac/cpu-features

MÜNCHEN

jax

Die Konferenz für Java, Architektur und Software-Innovation

MAINZ