# **TỐI ƯU HÓA HIỆU SUẤT KHỞI ĐỘNG NGUỘI CỦA AWS LAMBDA BẰNG CÁCH SỬ DỤNG CÁC CHIẾN LƯỢC MỒI NÂNG CAO VỚI SNAPSTART**

bởi Shan Kandaswamy và Gustavo Martim vào29 THÁNG 4 NĂM 2025 trong [Nâng cao (300)](https://aws.amazon.com/blogs/compute/category/learning-levels/advanced-300/) , [AWS Lambda](https://aws.amazon.com/blogs/compute/category/compute/aws-lambda/) , [Thực hành tốt nhất](https://aws.amazon.com/blogs/compute/category/post-types/best-practices/) , [Không có máy chủ](https://aws.amazon.com/blogs/compute/category/serverless/) , [Hướng dẫn kỹ thuật Liên](https://aws.amazon.com/blogs/compute/category/post-types/technical-how-to/) [kết cố định](https://aws.amazon.com/blogs/compute/optimizing-cold-start-performance-of-aws-lambda-using-advanced-priming-strategies-with-snapstart/) [Chia sẻ](https://aws.amazon.com/blogs/compute/optimizing-cold-start-performance-of-aws-lambda-using-advanced-priming-strategies-with-snapstart/)

Được giới thiệu tại re:Invent 2022, [SnapStart](https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html) là một giải pháp tối ưu hóa hiệu suất giúp xây dựng các ứng dụng có khả năng phản hồi cao và có thể mở rộng dễ dàng hơn bằng [AWS Lambda](https://aws.amazon.com/lambda/) . Yếu tố đóng góp lớn nhất vào độ trễ khởi động ( *thường được gọi là thời gian khởi động nguội)* là thời gian dành cho việc khởi tạo một hàm. Điều này bao gồm việc tải mã của hàm và khởi tạo các phụ thuộc. Đối với các khối lượng công việc nhạy cảm với độ trễ như API và các ứng dụng xử lý dữ liệu thời gian thực, độ trễ khởi động cao có thể dẫn đến trải nghiệm người dùng cuối không tối ưu. Lambda SnapStart có thể giảm thời gian khởi động từ vài giây xuống mức dưới một giây, với ít hoặc không có thay đổi mã. Bài đăng này thảo luận về 'Priming', một kỹ thuật để tối ưu hóa thêm thời gian khởi động cho các hàm AWS Lambda được xây dựng bằng Java và Spring Boot.

Các ứng dụng Spring Boot thường gặp độ trễ khởi động nguội cao trong quá trình khởi tạo JVM và khung, trong đó thời gian đáng kể được dành cho việc tải các lớp và thực hiện biên dịch Just-In-Time (JIT) của mã bytecode Java. Bài đăng trên blog này sử dụng ứng dụng Spring Boot làm ví dụ để truy xuất 10 bản ghi từ bảng 'UnicornEmployee' trong cơ sở dữ liệu [Amazon RDS](https://aws.amazon.com/rds/) cho PostgreSQL, trong đó mỗi bản ghi nhân viên bao gồm tên nhân viên, vị trí và ngày tuyển dụng.

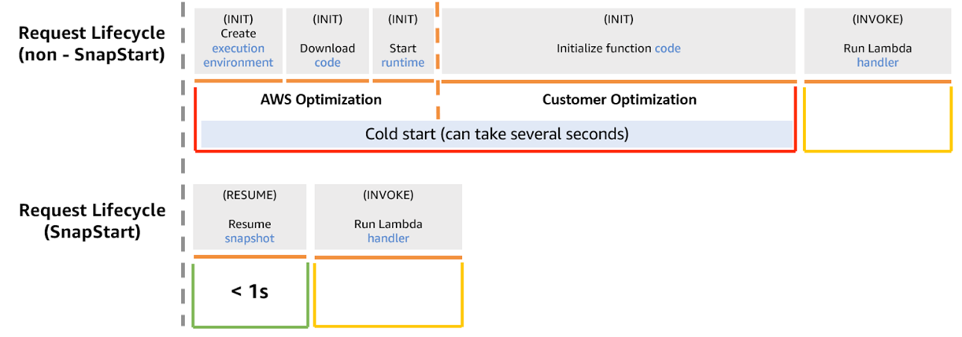
Ứng dụng mẫu sử dụng [Amazon API Gateway](https://aws.amazon.com/api-gateway/) kích hoạt hàm AWS Lambda kết nối với cơ sở dữ liệu thông qua RDS Proxy để trả về dữ liệu nhân viên. Mặc dù ứng dụng mẫu này sử dụng dữ liệu nhân viên giả để trình diễn, nhưng các mẫu và kỹ thuật tối ưu hóa được thảo luận trong bài đăng này có thể áp dụng cho các tình huống thực tế có các mẫu truy cập dữ liệu tương tự. Mã mẫu cho triển khai này có thể được tìm thấy trong kho lưu trữ GitHub của chúng tôi tại [lambda-priming-crac-java-cdk](https://github.com/aws-samples/lambda-priming-crac-java-cdk) .

## **Bối cảnh: SnapStart hoạt động như thế nào**

Bài đăng này giả định bạn đã quen thuộc với SnapStart và cung cấp một số thông tin cơ bản ngắn gọn. Để biết thêm chi tiết, hãy tham khảo [tài liệu SnapStart](https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html) .

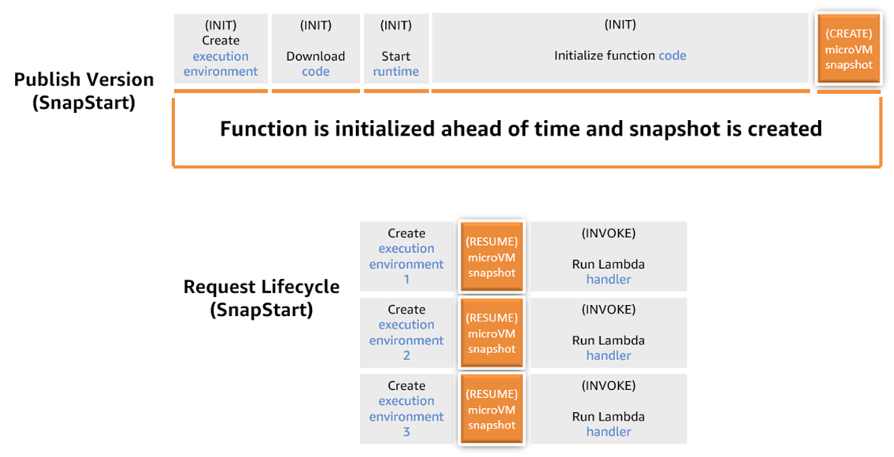
Để tóm tắt nhanh,  giai đoạn [INIT](https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html) cho hàm Lambda bao gồm việc tải xuống mã hàm, khởi động thời gian chạy và bất kỳ phụ thuộc bên ngoài nào, và chạy mã khởi tạo hàm. Đối với các hàm không sử dụng SnapStart, giai đoạn này xảy ra mỗi khi ứng dụng của bạn mở rộng quy mô để tạo môi trường thực thi mới. Khi SnapStart được kích hoạt, giai đoạn INIT xảy ra khi bạn xuất bản phiên bản hàm.

Hình ảnh sau đây cho thấy sự so sánh vòng đời yêu cầu Lambda có và không có SnapStart.



*Hình 1 – so sánh vòng đời yêu cầu Lambda có và không có SnapStart*

Vào cuối giai đoạn INIT, Lambda thực hiện  [các hook thời gian chạy](https://docs.aws.amazon.com/lambda/latest/dg/snapstart-runtime-hooks.html) trước điểm kiểm tra của bạn . Sau đó, Lambda chụp nhanh trạng thái bộ nhớ và đĩa của môi trường thực thi đã khởi tạo, lưu trữ ảnh chụp nhanh đã mã hóa và lưu vào bộ nhớ đệm để truy cập độ trễ thấp. Khi hàm được gọi sau đó, các môi trường thực thi mới sẽ được tiếp tục từ ảnh chụp nhanh đã lưu vào bộ nhớ đệm (trong giai đoạn [RESTORE](https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html" \l "runtimes-lifecycle-restore) ), giúp tăng tốc khởi động hàm.



*Hình 2 – môi trường thực thi mới được tiếp tục từ ảnh chụp nhanh được lưu trong bộ nhớ đệm.*

Bạn có thể xác thực tốc độ tăng tốc này bằng cách so sánh thời lượng RESTORE với thời lượng INIT được ghi lại trước SnapStart trong [Amazon CloudWatch](https://aws.amazon.com/cloudwatch/) Logs của hàm Lambda. Như được chứng minh trong bảng sau, việc bật SnapStart làm giảm độ trễ khởi động của ứng dụng Spring Boot mẫu của chúng tôi xuống 4,3 lần từ 6,1 giây xuống 1,4 giây. Độ trễ khởi động nguội 6,1 giây ON\_DEMANDchủ yếu là do sự kết hợp của (1) khởi tạo JVM và khung Spring Boot, (2) biên dịch JIT của mã ứng dụng được tải lười biếng trong quá trình gọi ban đầu và (3) thời gian cần thiết để thiết lập kết nối cơ sở dữ liệu với RDS thông qua [Amazon RDS Proxy](https://aws.amazon.com/rds/proxy/) . Bằng cách bật SnapStart, Lambda khởi tạo JVM và Spring Boot trước khi gọi hàm - dẫn đến độ trễ giảm đáng kể là 1,4 giây.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Phương pháp | Lời kêu gọi khởi động nguội | trang 50 | P90 | P99 | trang 99.9 |
| PrimingLogGroup-1\_ON\_DEMAND | 128 | 5047,94 giây | 5386,78 giây | 6158,80 giây | 6195,84 giây |
| PrimingLogGroup-2\_SnapStart\_NO\_PRIMING | 111 | 1177,87 giây | 1288,73 giây | 1419,94 giây | 1425,63 giây |

Bạn có thể giảm thiểu hơn nữa việc khởi động nguội cho các ứng dụng Spring Boot nhạy cảm với độ trễ bằng cách sử dụng các kỹ thuật mồi trên các hàm Lambda. Hãy cùng khám phá cách triển khai các kỹ thuật mồi.

## **Giải thích về mồi**

Priming là quá trình tải trước các phụ thuộc và khởi tạo tài nguyên trong giai đoạn INIT, thay vì trong giai đoạn INVOKE để tối ưu hóa hiệu suất khởi động với SnapStart hơn nữa. Điều này là cần thiết vì các khung Java sử dụng tiêm phụ thuộc sẽ tải các lớp vào bộ nhớ khi các lớp này được gọi một cách rõ ràng, điều này thường xảy ra trong giai đoạn INVOKE của Lambda. Bạn có thể chủ động tải các lớp bằng [các hook thời gian chạy Java](https://docs.aws.amazon.com/lambda/latest/dg/snapstart-runtime-hooks-java.html) , là một phần của  dự án [CRaC](https://openjdk.org/projects/crac/)  (Khôi phục phối hợp tại Điểm kiểm tra) nguồn mở. Bài đăng này trình bày cách sử dụng hook này, được gọi là beforeCheckpoint(), để khởi tạo các hàm Java hỗ trợ SnapStart, theo hai cách:

1. **Invoke Priming:** Phương pháp này bao gồm việc gọi trực tiếp các điểm cuối hoặc phương thức ứng dụng trong hook trước khi chụp nhanh của bạn để chúng được biên dịch JIT trong giai đoạn INIT và được đưa vào ảnh chụp nhanh. Điều này có thể bao gồm các hoạt động như gọi các điểm cuối API Gateway hoặc lấy dữ liệu từ thùng S3 hoặc cơ sở dữ liệu RDS để chủ động thực thi các đường dẫn mã, đảm bảo rằng các lớp cơ bản được đưa vào ảnh chụp nhanh.
2. **Class Priming:** Phương pháp này bao gồm việc khởi tạo chủ động các lớp trong giai đoạn INIT, đảm bảo rằng chúng được bao gồm trong ảnh chụp nhanh của hàm mà không có nguy cơ thay đổi trạng thái hoặc dữ liệu ứng dụng không mong muốn. Điều này có thể đạt được bằng cách tận dụng forName()phương thức của Java, phương thức này tải, liên kết và khởi tạo lớp đã chỉ định. Khởi tạo đề cập đến quy trình JVM tải định nghĩa lớp vào bộ nhớ, xác minh mã byte, chuẩn bị các trường tĩnh với các giá trị mặc định và thực thi các trình khởi tạo tĩnh. Điều này khác với việc khởi tạo, tức là tạo các đối tượng của lớp bằng cách sử dụng các hàm tạo. Để tạo danh sách các lớp cần thiết để tải trước, bạn có thể sử dụng tùy chọn VM sau, ghi danh sách vào tệp có tên *classes-loaded.txt* :  
   -Xlog:class+load=info:classes-loaded.txt

Trong khi invoke priming có thể mang lại hiệu suất tốt hơn, nó đòi hỏi nỗ lực bổ sung để đảm bảo rằng các hành động được thực hiện là idempotent và không có tác dụng phụ không mong muốn, ví dụ như xử lý các giao dịch tài chính trong ứng dụng ngân hàng. Vì lý do này, invoke priming chỉ nên được sử dụng khi mã được thực thi trong quá trình priming là idempotent hoặc không sửa đổi trạng thái. Đối với các tình huống không thể thực hiện điều này, class priming cung cấp một giải pháp thay thế an toàn hơn bằng cách chỉ khởi tạo các lớp mà không thực thi các phương thức của chúng. Lưu ý rằng điều này giả định rằng ứng dụng của bạn không thực thi mã sửa đổi trạng thái trong quá trình khởi tạo lớp.

Với bối cảnh này, chúng ta hãy xem cách triển khai Invoke và Class priming cho ứng dụng mẫu Spring Boot.

### **Ví dụ về việc triển khai mồi bằng cách sử dụng các móc thời gian chạy CRaC trước khi chụp ảnh nhanh Lambda**

Bài đăng này trình bày cả Invoke priming và Class priming bằng ứng dụng Spring Boot mẫu. Lựa chọn giữa hai phương pháp phụ thuộc vào các yêu cầu cụ thể và độ phức tạp của ứng dụng của bạn.

Bước 1: Thiết lập Ứng dụng Spring Boot của bạn bằng cách sử dụng aws-serverless-springboot3-archetypenhư được giải thích trong hướng dẫn [Khởi động nhanh Spring Boot3](https://github.com/aws/serverless-java-container/wiki/Quick-start---Spring-Boot3) của chúng tôi , thêm mã kết nối cơ sở dữ liệu hoặc chỉ cần sao chép dự án mẫu từ [kho lưu trữ GitHub](https://github.com/aws-samples/lambda-priming-crac-java-cdk) .

1. Tạo ứng dụng Spring Boot.

// src/main/java/software/amazon/awscdk/examples/unicorn/UnicornApplication.java

package software.amazon.awscdk.examples.unicorn;

…

@Import({ UnicornConfig.class })

@SpringBootApplication

public class UnicornApplication {

    private static final Logger log = LoggerFactory.getLogger(UnicornApplication.class);

    public static void main(String... arguments) {

        SpringApplication.run(UnicornApplication.class, arguments);

    }

}

1. Thêm tất cả các phụ thuộc Maven cần thiết cho Spring Boot, AWS Lambda và Database Connection vào  tệp [pom.xml](https://github.com/aws-samples/lambda-priming-crac-java-cdk/blob/main/software/priming/pom.xml) của bạn . Phụ thuộc được tô sáng sau đây chứa các lớp cần thiết để sử dụng các móc thời gian chạy CRaC.

...

        <dependency>

            <groupId>org.crac</groupId>

            <artifactId>crac</artifactId>

        </dependency>

...

1. Cấu hình kết nối cơ sở dữ liệu – Thiết lập thông tin chi tiết về kết nối cơ sở dữ liệu trong application.properties.

spring.datasource.password=${SPRING\_DATASOURCE\_PASSWORD}

spring.datasource.url=${SPRING\_DATASOURCE\_URL}

spring.datasource.username=postgres

spring.datasource.hikari.maximumPoolSize=1

Bước 2: Triển khai Trình xử lý hàm Lambda với các móc thời gian chạy CRaC và Gọi phương pháp tiếp cận Priming:

Tạo Trình xử lý hàm Lambda và tích hợp các móc thời gian chạy CRaC để thực thi beforeCheckpoint()và afterRestore()các phương thức trong ứng dụng của bạn trước khi chụp và sau khi khôi phục ảnh chụp nhanh.

1. Triển khai RequestHandler<UnicornRequest, UnicornResponse>giao diện trong lớp xử lý hàm Lambda.
2. Triển khai giao diện tài nguyên CRaC bằng hai phương thức: beforeCheckpoint()và afterRestore(), phương thức này xác định các hành động được thực hiện trước khi Lambda tạo ảnh chụp nhanh và sau khi ảnh chụp nhanh được khôi phục.
3. Thêm lệnh invoke priming bằng cách tạo một UnicornRequestđối tượng có GETyêu cầu tới điểm cuối cụ thể (chẳng hạn như /unicorn) và gọi handleRequest(unicornRequest, null)phương thức.

Điều này đảm bảo rằng các đường dẫn mã liên quan đến điểm cuối được chỉ định được biên dịch JIT và tối ưu hóa để thực thi nhanh hơn trong lần gọi đầu tiên sau khi ảnh chụp nhanh được khôi phục.

/src/main/java/software/amazon/awscdk/examples/unicorn/handler/InvokePriming.java

package software.amazon.awscdk.examples.unicorn.handler;

import org.crac.Core;

import org.crac.Resource;

...

public class InvokePriming implements RequestHandler<APIGatewayV2HTTPEvent, APIGatewayV2HTTPResponse>, Resource {

...

@Override

public APIGatewayV2HTTPResponse handleRequest(APIGatewayV2HTTPEvent event, Context context) {

    var awsLambdaInitializationType = System.getenv("AWS\_LAMBDA\_INITIALIZATION\_TYPE");

    var unicorns = getUnicorns();

    var body = gson.toJson(unicorns);

    return APIGatewayV2HTTPResponse.builder().withStatusCode(200).withBody(body).build();

}

@Override

public void beforeCheckpoint(org.crac.Context<? extends Resource> context)

        throws Exception {

    var event = APIGatewayV2HTTPEvent.builder().build();

    handleRequest(event, null);

}

...

}

Bước 3: Triển khai phương pháp tiếp cận chuẩn bị lớp:

Phương pháp class priming tập trung vào việc tải trước các lớp cần thiết để đạt được hiệu suất tối ưu. Để triển khai class priming, hãy tạo danh sách các lớp được tải trong quá trình khởi động ứng dụng và thực thi hàm bằng cách chạy ứng dụng cục bộ bằng đối số JVM sau:-Xlog:class+load=info:classes-loaded.txt

1. Đảm bảo rằng các lớp ứng dụng của bạn có trong classes-loaded.txttệp được tạo không thay đổi trạng thái trong quá trình khởi tạo tĩnh.  
   Lưu ý: tệp được tạo classes-loaded.txtcó chứa các mục nhập lớp theo định dạng sau:

[0.068s][info][class,load] software.amazon.awscdk.examples.unicorn.handler.ClassPriming

source: file:/var/task/

1. Chỉ trích xuất các tên lớp đủ điều kiện từ mỗi dòng và xóa thông tin ghi nhật ký bổ sung. Ví dụ:

software.amazon.awscdk.examples.unicorn.handler.ClassPriming

1. Sử dụng ClassLoaderUtil.loadClassesFromFile()phương thức tiện ích để trích xuất các mục lớp đã tạo.

    //src/main/java/software/amazon/awscdk/examples/unicorn/service/ClassLoaderUtil.java

package software.amazon.awscdk.examples.unicorn;

...

public class ClassLoaderUtil {

...

    public static void loadClassesFromFile() {

        log.info("loadClassesFromFile->started");

        Path path = Paths.get("classes-loaded.txt");

        try (BufferedReader bufferedReader = Files.newBufferedReader(path)) {

            Stream<String> lines = bufferedReader.lines();

            lines.forEach(line -> {

                var index1 = line.indexOf("[class,load] ");

                var index2 = line.indexOf(" source: ");

                if (index1 < 0 || index2 < 0) {

                    return;

                }

                var className = line.substring(index1 + 13, index2);

                try {

                    Class.forName(className, true,

                            ClassPriming.class.getClassLoader());

                } catch (Throwable ignored) {

                }

            });

            log.info("loadClassesFromFile->finished");

        } catch (IOException exception) {

            log.error("Error on newBufferedReader", exception);

        }

    }

...

}

1. Đọc một tệp (chẳng hạn như /classes-loaded.txt) chứa danh sách các lớp đã được tải trong quá trình thực thi ứng dụng trong beforeCheckpoint()phương thức.
2. Sử dụng Class.forName()phương pháp để tải và khởi tạo lớp, đảm bảo rằng lớp đã sẵn sàng trong quá trình chụp nhanh.  
   Lưu ý: bằng cách tải trước các lớp này một cách có hệ thống, phương pháp Class priming đơn giản hóa quy trình tối ưu hóa và giảm bớt sự phức tạp liên quan đến Invoke priming.

//src/main/java/software/amazon/awscdk/examples/unicorn/handler/ClassPriming.java

package software.amazon.awscdk.examples.unicorn.handler;

...

import org.crac.Core;

import org.crac.Resource;

public class ClassPriming implements RequestHandler<APIGatewayV2HTTPEvent, APIGatewayV2HTTPResponse>, Resource {

...

        ConfigurableApplicationContext configurableApplicationContext =

SpringApplication.run(UnicornApplication.class);

        this.unicornService = configurableApplicationContext.getBean(UnicornService.class);

        this.gson = configurableApplicationContext.getBean(Gson.class);

        Core.getGlobalContext().register(this);

    }

    @Override

    public APIGatewayV2HTTPResponse handleRequest(APIGatewayV2HTTPEvent event, Context context) {

        var unicorns = getUnicorns();

        var body = gson.toJson(unicorns);

        return APIGatewayV2HTTPResponse.builder().withStatusCode(200).withBody(body).build();

    }

    @Override

    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)

            throws Exception {

        ClassLoaderUtil.loadClassesFromFile();

    }

...

}

Bước 4: Thiết lập cơ sở hạ tầng AWS CDK

Trước khi tiếp tục, hãy xem lại các điều kiện tiên quyết trong tệp [README](https://github.com/aws-samples/lambda-priming-crac-java-cdk/tree/main" \l "requirements) của dự án .

Ngăn xếp CDK triển khai một ứng dụng không có máy chủ và cơ sở hạ tầng cần thiết để thử nghiệm các chiến lược tối ưu hóa Lambda khác nhau. Nó tạo ra một VPC với các mạng con riêng, một RDS cho phiên bản PostgreSQL với proxy cơ sở dữ liệu và năm hàm Lambda triển khai các phương pháp tối ưu hóa khác nhau (ON\_DEMAND không có SnapStart, SnapStart không có priming, SnapStart với invoke priming và SnapStart với class priming). Mỗi hàm Lambda được tích hợp với API Gateway để truy cập HTTP, được định cấu hình với thời gian chạy Java 21 trên kiến ​​trúc ARM64 và bao gồm các nhóm nhật ký CloudWatch để giám sát.

Thực hiện theo các bước sau để triển khai cơ sở hạ tầng:

1. Sao chép kho lưu trữ mẫu:

git clone https://github.com/aws-samples/lambda-priming-crac-java-cdk.git

1. Triển khai ngăn xếp CDK:

cd lambda-priming-crac-java-cdk/infrastructure

cdk synth

cdk deploy --require-approval never --all 2>&1 | tee cdk\_output.txt

1. Lưu URL API Gateway:  
   Việc triển khai sẽ xuất ra năm URL theo định dạng này:

# ON\_DEMAND endpoint (without SnapStart)

LambdaPrimingCracJavaCdkStack.PrimingJavaRestApi1ONDEMANDEndpoint = https://[id].execute-api.us-east-1.amazonaws.com/prod/

# SnapStart without priming endpoint

LambdaPrimingCracJavaCdkStack.PrimingJavaRestApi2SnapStartNOPRIMINGEndpoint = https://[id].execute-api.us-east-1.amazonaws.com/prod/

# SnapStart with invoke priming endpoint

LambdaPrimingCracJavaCdkStack.PrimingJavaRestApi3SnapStartINVOKEPRIMINGEndpoint = https://[id].execute-api.us-east-1.amazonaws.com/prod/

# SnapStart with class priming endpoint

LambdaPrimingCracJavaCdkStack.PrimingJavaRestApi4SnapStartCLASSPRIMINGEndpoint = https://[id].execute-api.us-east-1.amazonaws.com/prod/

# Database setup endpoint

LambdaPrimingCracJavaCdkStack.PrimingJavaRestApi5DBLOADEREndpoint=https://[id].execute-api.us-east-1.amazonaws.com/prod/

1. Trích xuất các URL thành các biến để thử nghiệm:

ONDEMAND\_URL=$(grep -oE 'https://[a-zA-Z0-9.-]+\.execute-api\.[a-zA-Z0-9-]+\.amazonaws\.com/prod/' "cdk\_output.txt" | head -n 1) \

NOPRIMING\_URL=$(grep -oE 'https://[a-zA-Z0-9.-]+\.execute-api\.[a-zA-Z0-9-]+\.amazonaws\.com/prod/' "cdk\_output.txt" | head -n 2 | tail -n 1) \

INVOKEPRIMING\_URL=$(grep -oE 'https://[a-zA-Z0-9.-]+\.execute-api\.[a-zA-Z0-9-]+\.amazonaws\.com/prod/' "cdk\_output.txt" | head -n 3 | tail -n 1) \

CLASSPRIMING\_URL=$(grep -oE 'https://[a-zA-Z0-9.-]+\.execute-api\.[a-zA-Z0-9-]+\.amazonaws\.com/prod/' "cdk\_output.txt" | head -n 4 | tail -n 1) \

SETUP\_URL=$(grep -oE 'https://[a-zA-Z0-9.-]+\.execute-api\.[a-zA-Z0-9-]+\.amazonaws\.com/prod/' "cdk\_output.txt" | head -n 5 | tail -n 1)

Bước 5: Tải cơ sở dữ liệu và chạy thử nghiệm hiệu suất bằng [fortress](https://www.artillery.io/) :

1. Khởi tạo cơ sở dữ liệu bằng dữ liệu mẫu.

curl -X GET "$SETUP\_URL"

#Expected output: {"message":"Database schema initialized and data loaded"}

1. Chạy thử nghiệm hiệu suất cho tất cả các điểm cuối

artillery run -t "$ONDEMAND\_URL" -v '{ "url": "/unicorn" }' ./loadtest.yaml && \

artillery run -t "$NOPRIMING\_URL" -v '{ "url": "/unicorn" }' ./loadtest.yaml && \

artillery run -t "$INVOKEPRIMING\_URL" -v '{ "url": "/unicorn" }' ./loadtest.yaml && \

artillery run -t "$CLASSPRIMING\_URL" -v '{ "url": "/unicorn" }' ./loadtest.yaml

Bước 6: So sánh kết quả kiểm tra tải cho On-demand (không phải SnapStart), SnapStart, Invoke priming và Class priming

Kết quả kiểm tra hiệu suất trong bảng bên dưới được sắp xếp từ độ trễ khởi động chậm nhất đến nhanh nhất. Hàm không có SnapStart thực hiện chậm nhất do khởi tạo JVM, tải lớp và biên dịch JIT xảy ra khi hàm được gọi. Lưu ý cải tiến 4,3 lần với SnapStart, tiếp tục các lệnh gọi từ ảnh chụp nhanh được khởi tạo trước, do đó tránh được khởi tạo JVM và biên dịch JIT ban đầu. SnapStart với lớp mồi đạt tốc độ nhanh hơn 1,4 lần so với SnapStart, bằng cách chủ động tải/khởi tạo các lớp trong INIT để chúng được đưa vào ảnh chụp nhanh của hàm. Cuối cùng, SnapStart với lớp mồi gọi đạt hiệu suất nhanh nhất – với độ trễ khởi động nguội 781,68ms p99.9 nhanh hơn 1,8 lần so với SnapStart. Điều này là do ngoài việc khởi tạo các lớp, nó còn thực thi các phương thức trên các thể hiện của các lớp đó, dẫn đến việc đưa nhiều thành phần hơn vào ảnh chụp nhanh của hàm.

Lưu ý rằng với invoke priming, bất kỳ mã ứng dụng nào bạn thực thi phải là idempotent hoặc chỉ sửa đổi dữ liệu stub. Ví dụ, hãy xem xét mã ứng dụng kích hoạt giao dịch tài chính. Nếu mã này được thực thi trong quá trình invoke priming với dữ liệu người dùng thực, nó có thể gây ra những tác động không mong muốn với hậu quả nghiêm trọng tiềm tàng. Class priming tránh được điều này, vì các lớp ứng dụng được khởi tạo thay vì được khởi tạo và các phương thức của chúng được thực thi. Điều này giả định rằng mã ứng dụng không thực thi logic sửa đổi trạng thái trong quá trình khởi tạo lớp. Chúng tôi khuyên bạn nên ghi nhớ những cân nhắc này khi sử dụng invoke và/hoặc class priming và chọn phương pháp phù hợp cho trường hợp sử dụng của bạn.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Phương pháp | Lời kêu gọi khởi động nguội | trang 50 | P90 | P99 | trang 99.9 |
| PrimingLogGroup-1\_ON\_DEMAND | 128 | 5047,94 giây | 5386,78 giây | 6158,80 giây | 6195,84 giây |
| PrimingLogGroup-2\_SnapStart\_NO\_PRIMING | 111 | 1177,87 giây | 1288,73 giây | 1419,94 giây | 1425,63 giây |
| PrimingLogGroup-4\_SnapStart\_CLASS\_PRIMING | 82 | 857,81 giây | 997,49 giây | 1085,94 giây | 1085,94 giây |
| PrimingLogGroup-3\_SnapStart\_INVOKE\_PRIMING | 66 | 608,42 giây | 688,88 giây | 781,68 giây | 781,68 giây |

## **Phần kết luận**

Bài đăng này cho thấy cách AWS Lambda SnapStart, được tăng cường bởi các móc thời gian chạy CRaC, mở khóa khả năng kiểm soát chi tiết đối với quá trình tối ưu hóa khởi động nguội cho các ứng dụng Java thông qua hai chiến lược mồi riêng biệt:

* **Invoke Priming** : cải thiện hiệu suất bằng cách thực hiện các điểm cuối quan trọng trong quá trình tạo ảnh chụp nhanh, lý tưởng cho các quy trình làm việc có tính bất biến.
* **Class Priming** : tải trước các lớp mà không kích hoạt logic kinh doanh, giảm thiểu rủi ro tác dụng phụ.

Để triển khai các kỹ thuật tối ưu hóa này trong ứng dụng của bạn, hãy đánh giá trường hợp sử dụng của bạn và lựa chọn phương pháp tiếp cận mồi tối ưu. Theo dõi việc giảm độ trễ và sử dụng tài nguyên của ứng dụng của bạn thông qua số liệu của Amazon CloudWatch để định lượng các cải tiến về hiệu suất. Bằng cách tích hợp các chiến lược này, các nhà phát triển có thể đạt được khởi động lạnh dưới một giây trong khi vẫn duy trì khả năng mở rộng và hiệu quả về chi phí của kiến trúc không máy chủ bằng Java.

Để tìm hiểu sâu hơn, hãy xem [kho lưu trữ GitHub](https://github.com/aws-samples/lambda-priming-crac-java-cdk) với mã ví dụ đầy đủ, bao gồm hướng dẫn thiết lập và các mẫu có thể tái sử dụng mà bạn có thể áp dụng cho các dự án của riêng mình. Để biết thêm [ví dụ về các ứng dụng Java chạy trên AWS Lambda](https://serverlessland.com/repos?services=lambda&runtime=Java) , hãy truy cập [serverlessland.com](https://serverlessland.com/) và khám phá nhiều tài nguyên, hướng dẫn và trường hợp sử dụng thực tế.