İZMİR KÂTİP ÇELEBİ ÜNİVERSİTESİ
2010
FEN BİLİMLERİ ENSTİTÜSÜ

Node js ve Java Programlama Dillerininin Kullanımının AWS Lambda Performansına Etkilerinin İncelenmesi

Yazılım Mühendisliği Ana Bilim Dalı
Dönem Projesi

Özüm Kalkan
Y230240089

Proje Danışmanı: Doç. Dr. Vahide BULUT

Haziran 2024

Node js ve Java Programlama Dillerininin Kullanımının AWS Lambda Performansına Etkilerinin İncelenmesi

Öz

Bulut bilişim, modern yazılım geliştirme süreçlerinde büyük bir öneme sahiptir. Sunucusuz (Serverless) mimari, geliştiricilerin altyapıyı yönetmekle uğraşmaksızın uygulama geliştirmesine olanak tanımaktadır. Bu, geliştiricilerin kodlarını yüklemek, çalıştırmak ve ölçeklendirmek için bulut sağlayıcısının yönetimine güvenmeleri anlamına gelmektedir.

AWS Lambda, sunucusuz uygulama geliştirmenin önde gelen hizmetlerinden bir tanesidir. Programlama dili seçimi, AWS Lambda işlevlerinin performansı üzerinde doğrudan bir etkiye sahiptir. Node.js ve Java gibi farklı programlama dilleri, Lambda fonksiyonunun performansını önemli ölçüde etkileyebilmektedir.

Bu çalışmada, Node.js ve Java programlama dillerinin AWS Lambda performansına etkisi incelenmiştir. İki farklı programlama dili arasındaki performans farklarını anlamak, geliştiricilere doğru programlama dilini seçme konusunda önemli bir rehberlik sağlayabilir.

Anahtar Sözcükler: AWS Lambda, Sunucusuz Mimari, Node.js, Java, Performans Analizi, Bulut Bilişim, Programlama Dilleri, DynamoDB, CloudWatch Metrics.

Analyzing the Impact of Node.js and Java Programming Languages on AWS Lambda Performance

Abstract

Cloud computing holds significant importance in modern software development processes. Serverless architecture allows developers to develop applications without managing the infrastructure. This means developers can rely on the cloud provider's management to upload, run, and scale their code.

AWS Lambda is one of the leading services for serverless application development. The choice of programming language directly affects the performance of AWS Lambda functions. Different programming languages such as Node.js and Java can significantly influence the performance of the Lambda function.

This study examines the impact of Node.js and Java programming languages on the performance of AWS Lambda. Understanding the performance differences between these two programming languages can provide developers with essential guidance in choosing the right programming language.

Keywords: AWS Lambda, Serverless Architecture, Node.js, Java, Performance Analysis, Cloud Computing, Programming Languages, DynamoDB, CloudWatch Metrics.

İÇİNDEKİLER

Şekiller Listesi

Kısaltmalar Listesi

AWS	Amazon Web Services
BT	Bilgi Teknolojileri
JIT	Just-In-Time
JVM	Java Virtual Machine
I/O	Input/Output
API	Application Programming Interface
ms	Milisaniye

Semboller Listesi

%	Yüzde

BÖLÜM I
GİRİŞ

Bulut Bilişim, veri ve programların bilgisayarın sabit diski veya yerel sunucusu yerine internet üzerinde barındırılan uzak sunucularda saklanması ve bu sunuculara erişilmesi anlamına gelmektedir. Bulut bilişim aynı zamanda internet tabanlı bilişim olarak da anılmaktadır, kaynağın internet üzerinden kullanıcıya hizmet olarak sunulduğu bir teknolojidir. Saklanan veriler dosyalar, resimler, belgeler veya başka herhangi bir saklanabilir belge olabilir.

Bulut bilişim işletmeler ve geliştiriciler için hız, ölçeklenebilirlik ve esneklik sağlayan bir dönüşümü tetiklemiştir. Bu dönüşüm, geleneksel altyapı maliyetlerini azaltırken, uygulamaların daha hızlı dağıtılmasını ve güncellenmesini sağlamak için yeni imkanlar sunmaktadır. Amazon Web Services (AWS), bulut bilişim alanında geniş bir hizmet yelpazesi sunmaktadır. AWS Lambda ise bu hizmetlerden biri olarak sunulan kaynakların yönetimini en aza indirerek, fonksiyonların hızlı bir şekilde yürütülmesini sağlayan bir serverless (sunucusuz) hesaplama hizmetidir.

Bu çalışmada, AWS Lambda hizmetinin performansı üzerindeki etkileri açısından iki farklı programlama dili olan Node.js ve Java'nın incelenmesi amaçlanmıştır. Node.js hızlı bir şekilde popülerlik kazanan ve özellikle web uygulamaları için kullanılan bir ortamken, Java ise uzun yıllardır endüstride yaygın olarak kullanılan ve güçlü bir platformdur. AWS Lambda üzerinde bu iki dili karşılaştırmak geliştiricilere hangi dilin daha iyi performans sağladığı konusunda değerli bir bakış açısı sunabilir.

Bu çalışmada Node.js ve Java'nın AWS Lambda üzerindeki performansını karşılaştırmak için çeşitli kriterler kullanılacaktır. Bu kriterler arasında başlatma süresi, fonksiyon yürütme süresi, bellek kullanımı ve ölçeklenebilirlik gibi faktörler yer alacaktır. Elde edilen bulgular, geliştiricilere AWS Lambda'da hangi programlama dilinin tercih edilmesi gerektiği konusunda rehberlik edebilir ve bulut tabanlı uygulama geliştirmeye yönelik stratejilerin geliştirilmesine katkıda bulunabilir.

BÖLÜM II

Bulut Bilişim, Sunucusuz Mimari ve AWS Lambda, Golang ve Node js Programlama Dilleri Tanımları ve Özellikleri

Bu bölümde, bulut bilişim sunucusuz mimari ve AWS Lambda ile ilgili özelliklerden bahsedilmiştir. Bunlar başlıklar halinde verilecektir.

2.1. Bulut Bilişim

Bulut bilişim, sunucular, depolama, veritabanları, ağ oluşturma, yazılım, analiz ve daha fazlası gibi bilişim hizmetlerini internet üzerinden edinme ve kullanma yeteneğidir. Bulut bilişim, talep üzerine bilgi işlem kaynakları sağlayarak kendi fiziksel sunucularınızı kurma, kendi yazılımınızı çalıştırma ve kendi veritabanlarınızı yönetme ihtiyacını ortadan kaldırmanıza olanak tanımaktadır. Ofisteki bir bilgisayara bağlanmaya gerek kalmadan verilere, uygulamalara ve bilgi işlem kaynaklarına dünyanın her yerinden erişebilmeyi sağlamaktadır. Sonuç olarak bulut bilişim, daha hızlı inovasyon, esnek kaynaklar ve ölçek ekonomisi sunmaktadır.

2.1.1. Bulut Bilişimin En Önemli Faydaları

- Sabit giderleri değişken giderlerle değiştirme - Veri merkezlerine ve sunuculara yoğun yatırım yapmak yerine, yalnızca bilgi işlem kaynaklarının kullanım miktarına göre ödeme yapılmasını sağlamaktadır. Bu durum bir veri merkezine yatırım yapılıp sabit bir gidere sahip olmaktansa, yapılan kullanım kadar ödeme yapılmasını sağlayarak değişken giderlere neden olmaktadır.

- Ölçeklenme ekonomilerinden yararlanma - Yüzbinlerce müşterinin kullanımı bulutta toplandığı için AWS gibi sağlayıcılar daha yüksek ölçek ekonomisi elde edebilmektedir, bu da daha düşük kullandıkça öde fiyatları anlamına gelmektedir. Bulut bilişim sağlayıcıları, müşterilere dinamik olarak kaynak tahsisi yapma ve ihtiyaç duydukları kadar kaynağa erişmelerini sağlama yeteneği sunmaktadır. Bu işletmelerin talepleri arttıkça veya azaldıkça kaynakları otomatik olarak ölçeklendirebilmelerini sağlamaktadır. Lambda ölçeklendirme ise bu bağlamda özellikle AWS Lambda adlı hizmette kullanılan bir kavramdır.

- Kapasiteyi tahmin etmeye son verme - Bulut bilişim altyapı kapasitesi ihtiyaçlarınıza ilişkin tahmin yürütmeyi ortadan kaldırmayı sağlamaktadır. Bir uygulamayı dağıtmadan önce kapasite kararı verildiğinde, genellikle ya pahalı boş kaynaklarla ya da sınırlı kapasiteyle uğraşılmaktadır. Bulut bilişimle bu sorunlar ortadan kalkmaktadır. İhtiyaç duyulduğu kadar çok veya az kapasiteye erişebilir ve yalnızca

birkaç dakika önceden bildirimde bulunarak ölçeği istenildiği gibi artırıp azaltabilmeyi sağlamaktadır.

- Hızı ve çevikliği artırma - Bulut bilgi işlem ortamında, BT kaynaklarını geliştiricilerinizin kullanımına sunma süresini haftalardan yalnızca dakikalara indirildiği anlamına gelmektedir. Bu deneme ve geliştirme için gereken maliyet ve süre önemli ölçüde düşük olduğundan, kuruluşun çevikliğinde önemli ölçüde bir artışla sonuçlanmaktadır.
- Veri merkezlerini işletmek ve bakımını yapmak için para harcamayı bırakma - Altyapıya değil, işinizi farklılaştıran projelere odaklanmanız sağlamaktadır. Bulut bilişim, sunucuları rafa kaldırma, istifleme ve güçlendirme gibi ağır işlerden ziyade kendi müşterilerinize odaklanmanıza olanak tanımaktadır.
- Dakikalar içinde küreselleşme - Uygulamanın yalnızca birkaç tıklamayla dünyanın çeşitli bölgelerinde kolayca dağıtılmasını sağlamaktadır. Bu minimum maliyetle daha düşük gecikme süresi ve daha iyi bir deneyim sağlanabileceği anlamına gelmektedir.

2.2. Sunucusuz Mimari

Sunucusuz mimari, altyapıyı yönetmeye gerek kalmadan uygulama ve hizmetler oluşturup çalıştırmayı sağlamaktadır. Sunucusuz mimari ile uygulamanız hâlâ sunucularda çalışmaktadır ancak sunucu yönetiminin tamamı AWS tarafından yapılmaktadır. Uygulamaları, veritabanlarını ve depolama sistemlerini çalıştırmak için sunucuları tedarik etmemize, ölçeklendirmemize ve bakımını yapmamıza gerek yoktur.

Sunucusuz mimari kullanarak, geliştiriciler bulutta veya şirket içinde sunucuları veya çalışma zamanlarını yönetme ve çalıştırma konusunda endişelenmek yerine temel ürünlerine odaklanabilmektedirler. Bu azaltılmış genel gider, geliştiricilerin, ölçeklenebilir ve güvenilir harika ürünler geliştirmek için harcayabilecekleri zaman ve enerjiyi geri kazanmalarına olanak tanımaktadır.

2.2.1. Sunucusuz Mimarinin Avantajları

- Uygun maliyetli: Sunucusuz bilgi işlem, sunucu bakımı ve yükseltmesi için ödeme yapılan geleneksel sunucu modellerinin aksine, yalnızca kullanılan kaynaklar için ödeme yapıldığı için maliyetlerin azaltılmasına yardımcı olmaktadır (Cleary, 2024). .
- Ölçeklenebilirlik: Sunucusuz bilgi işlem, uygulamanın taleplerini karşılamak için otomatik olarak ölçeklendirilmektedir. Yani eğer uygulama daha fazla kaynağa ihtiyaç duyarsa sunucusuz mimari daha fazla kaynağı otomatik olarak tahsis edebilmektedir.

- Yüksek kullanılabilirlik: Sunucusuz mimari dağıtılmış ve yedekli olduğundan, yüksek kullanılabilirliğe ulaşılmasına yardımcı olmaktadır, bu da hizmet kesintisi veya arıza riskinin daha düşük olduğu anlamına gelmektedir.

- Artan geliştirme hızı: Geliştiricilerin altyapı yönetimi konusunda endişelenmelerine gerek yoktur ve kodlamaya odaklanabilmektedirler, bu da geliştirme sürecinin hızlanmasını sağlamaktadır.

- Kolay bakım: Sunucusuz bilgi işlem, geliştiricilerin sunucuları veya işletim sistemlerini yönetmek yerine kod güncellemelerine odaklanmasına olanak tanımaktadır.

2.3. Amazon Web Servisleri

Amazon Web Services (AWS) platformu, dünyanın her yerindeki veri merkezlerinden 200'den fazla tam özellikli hizmet sağlar ve dünyanın en kapsamlı bulut platformudur.

AWS, ölçeklenebilir ve uygun maliyetli bulut bilişim çözümleri sağlayan çevrimiçi bir platformdur.

AWS, şirketlerin ölçeklenmesine ve büyümesine yardımcı olmak için bilgi işlem gücü, veritabanı depolama, içerik dağıtımı vb. gibi çeşitli isteğe bağlı işlemler sunan, geniş çapta benimsenen bir bulut platformudur.

2.3.1. Amazon Web Servislerinin Avantajları

AWS (Amazon Web Services), bulut bilişim alanında lider bir sağlayıcı olarak birçok avantaj sunmaktadır.

- Geniş Hizmet Yelpazesi: AWS, çeşitli hizmetler sunarak altyapı, veritabanı, yapay zeka, depolama, güvenlik ve daha pek çok alanda geniş bir hizmet yelpazesine sahiptir. Bu, farklı iş gereksinimlerine uygun çözümler sağlamaktadır.

- Ölçeklenebilirlik ve Esneklik: AWS, işletmelerin ihtiyaçlarına göre kolayca ölçeklenebilmektedir. Kaynakları anında artırabilir veya azaltabilir, böylece iş yüklerine dinamik bir şekilde uyum sağlayabilmektedir.

- Yüksek Güvenlik: AWS, müşterilerinin verilerini güvende tutmak için kapsamlı güvenlik önlemleri sunmaktadır. Veri merkezleri fiziksel güvenlikten, ağ güvenliğine ve kimlik doğrulamaya kadar çeşitli güvenlik katmanlarıyla korunmaktadır.

- Maliyet Verimliliği: AWS, işletmelerin yalnızca kullandıkları kaynaklar için ödeme yapmalarını sağlamaktadır. Bu donanım satın almak ve bakım yapmak gibi

maliyetleri azaltmaktadır ve ölçeklenebilirlikle birlikte işletmelerin maliyet etkinliğini artırmaktadır.

- Küresel Altyapı: AWS, dünya çapında birçok bölgede veri merkezleri bulundurarak hızlı erişim ve düşük gecikme süreleri sağlamaktadır. Bu kullanıcıların ve uygulamaların her yerden yüksek performanslı bir şekilde çalışmasını sağlamaktadır.

- Yenilikçi Hizmetler: AWS, sürekli olarak yeni hizmetler ve özellikler ekleyerek müşterilerine rekabet avantajı sağlamaktadır. Bu sayede işletmeler, en son teknolojileri kullanarak inovasyonlarını hızlandırabilmektedir.

- Yönetim Kolaylığı: AWS, kullanımı kolay bir yönetim arayüzü ve otomatikleştirilmiş yönetim hizmetleri sunmaktadır. Bu, iş yüklerini yönetmeyi ve operasyonel verimliliği artırmayı kolaylaştırmaktadır.

2.4. AWS Lambda

AWS Lambda, Amazon Web Services'in (AWS) sunucusuz hesaplama hizmetidir. Geleneksel sunucu tabanlı yaklaşımlardan farklı olarak, AWS Lambda'da altyapı yönetimi ve sunucu bakımıyla ilgilenmenize gerek yoktur. Bu hizmet, uygulamaları yürütmek için gerekli olan kaynakları otomatik olarak yönetmektedir ve sadece fonksiyonlarınızın çalıştığı süre boyunca ödeme yapmanızı sağlamaktadır.

AWS Lambda'nın ana prensibi, olay odaklı bir yaklaşıma dayanır. Bu, Lambda fonksiyonlarının belirli olaylar meydana geldiğinde tetiklenmesi ve bu olaylara yanıt vermesi demektir. Örneğin, bir dosya yüklendiğinde veya bir HTTP isteği yapıldığında bir Lambda fonksiyonu çalıştırılabilir.

Lambda, farklı programlama dillerinde fonksiyonların yazılmasına olanak tanımaktadır. Başlıca desteklenen diller arasında Node.js, Python, Java, C#, Go ve Ruby bulunmaktadır. Bu sayede geliştiriciler, tercih ettikleri dili kullanarak Lambda fonksiyonlarını oluşturabilmektedirler.

AWS Lambda'nın temel avantajlarından biri, kullanıcıların altyapıya ve sunucu yönetimine odaklanmak zorunda kalmadan uygulamalarını hızlı bir şekilde dağıtabilmeleridir. Ayrıca, Lambda'nın ölçeklenebilirliği sayesinde, iş yüklerinin artan taleplerine dinamik olarak uyum sağlanabilmektedir.

Genel olarak, AWS Lambda, geliştiricilere hızlı, esnek ve maliyet etkin bir şekilde uygulamalarını çalıştırma ve ölçekleme imkânı sunan güçlü bir sunucusuz hesaplama çözümüdür.

2.4.1. AWS Lambda’nın Avantajları

AWS Lambda'nın birçok avantajı bulunmaktadır.

- Maliyet Etkinliği: AWS Lambda, yalnızca kullanılan işlem süresi ve kaynak miktarı için ödeme yapılmasını sağlamaktadır. Geleneksel sunucu tabanlı yaklaşımlara kıyasla, kaynakların sürekli olarak kullanılması gerekmediği için maliyetler genellikle daha düşüktür.

- Ölçeklenebilirlik: Lambda, otomatik olarak ölçeklenebilmektedir. Artan iş yüklerine anında yanıt verir ve gerektiğinde daha fazla kaynağı dinamik olarak tahsis edilmesini sağlamaktadır. Bu, uygulamanın performansını artırırken, operasyonel karmaşıklığı azaltmaktadır.

- Yönetim Kolaylığı: AWS Lambda, altyapı yönetimi ve sunucu bakımı gibi operasyonel görevlerden geliştiricileri kurtarmaktadır. AWS tarafından otomatik olarak yönetilmektedir, bu da geliştiricilerin uygulamalarını daha hızlı bir şekilde dağıtmalarına ve güncellemeler yapmalarına olanak tanımaktadır.

- Hızlı Başlatma Süresi: Lambda fonksiyonları, hızlı başlatma sürelerine sahiptir. Bu, fonksiyonların hızlı bir şekilde başlatılmasını ve uygulamanın hızlı bir şekilde yanıt vermesini sağlamaktadır.

- Olay Tabanlı Mimariyi Destekleme: AWS Lambda, olay odaklı bir mimariyi desteklemektedir. Belirli olaylar meydana geldiğinde Lambda fonksiyonlarının tetiklenmesini sağlamaktadır. Örneğin, dosya yükleme, veritabanı güncelleme veya HTTP isteği gibi olaylar Lambda fonksiyonlarını tetikleyebilmektedir.

- Çoklu Programlama Dilini Destekleme: AWS Lambda, birçok farklı programlama dilini desteklemektedir. Geliştiricilere tercih ettikleri dili kullanarak fonksiyonlarını yazma esnekliği sağlamaktadır.

2.4.2. AWS Lambda Soğuk Başlangıç Durumu

Bir lambda fonksiyonu ilk kez çağırıldığında, yürütülmesi ve yanıt verilmesi biraz zaman almaktadır. Aynı fonksiyon ile yapılan herhangi bir sonraki tekrar çağrı, ilk çağırmadan çok daha hızlı olacaktır. Bu nedenle, bir fonksiyon ile yapılan ilk çağrıya "soğuk başlatma" denmektedir.

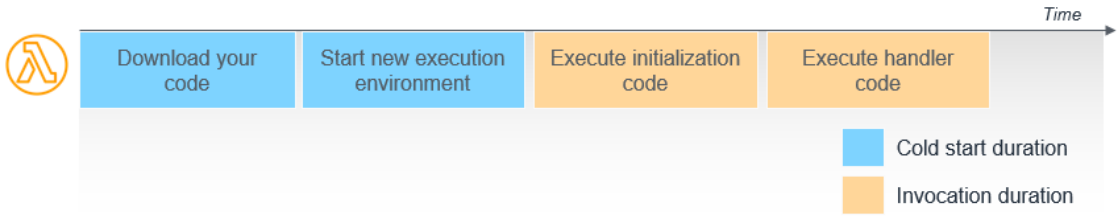
Soğuk başlatmanın bağlı olduğu birkaç faktör bulunmaktadır:

- Çalışma zamanı: Her programlama dili ve teknolojinin kendi başlangıç süresi vardır. Örneğin, Python gibi bazı dillerin soğuk başlatma süresi, Java gibi diğer dillere kıyasla daha kısa olabimektedir.

- Bellek sınırı: AWS Lambda'da fonksiyonu tanımlarken, bellek sınırı seçilebilmektedir. Daha yüksek bellek sınırları, daha iyi donanım altyapısı ve işlemci

kaynaklarına erişim sağlayabilir, bu da daha hızlı bir soğuk başlatma süresine katkıda bulunabilir.

- Kod: Lambda fonksiyonunun kod yapısı ve boyutu, soğuk başlatma süresini etkileyebilmektedir. Büyük ve karmaşık bir kod tabanı, daha uzun başlatma sürelerine yol açabilmektedir. Kodun optimize edilmesi, bağımlılıklarının azaltılması ve gerekli olmayan bileşenlerin kaldırılması, başlatma sürelerini iyileştirebilir.
- Bağımlılıklar ve dış hizmetlere erişim: Lambda fonksiyonunun başlatılması sırasında dış hizmetlere (örneğin, veritabanlarına veya API'lere) erişim gerektiren bağımlılıklar, başlatma süresini uzatabilmektedir. Bu nedenle, dış hizmetlere olan bağımlılıkları ve erişimi mümkün olduğunca optimize etmek önemlidir.



Şekil 1: AWS Lambda Soğuk Başlatma Durumu

Şekil 1'de görüldüğü gibi kodu ayarlamanın ilk iki adımına "soğuk başlangıç" adı verilir. Lambda'nın fonksiyonu hazırlaması için gereken süre genel çağrı süresini arttırmaktadır (Beswick, 2021).

2.4.3. AWS Lambda Sıcak Başlatma Durumu

Soğuk başlatmanın ardından AWS Lambda kaynak yönetimini ve performansı iyileştirmek için yürütme ortamını belirleyici olmayan bir süre boyunca korumaktadır. Bu süre içerisinde aynı fonksiyon için başka bir istek gelmesi durumunda servis ortamı yeniden kullanabilmektedir. Yürütme ortamı zaten mevcut olduğundan ve kodu indirip başlatma kodunu çalıştırmaya gerek olmadığından, bu ikinci istek genellikle daha hızlı tamamlanmaktadır. Buna "sıcak başlangıç" denilmektedir.

2.5. Programlama Dilleri

Bir programlama dili, yazılım programları oluşturmak için kullanılan talimatlar ve sözdizimidir. Programlama dillerinin bazı temel özellikleri şunlardır:

- Sözdizimi: Bir programlama dilinde kod yazmak için kullanılan belirli kurallar ve yapı sözdizimi olarak ifade edilmektedir.
- Veri Türleri: Bir programda depolanabilen değerlerin türleri, örneğin sayılar, dizeler ve mantıksal değerlerdir.
- Değişkenler: Değerleri depolayabilen adlandırılmış bellek konumlarıdır.

- Operatörler: Değerler üzerinde işlemler gerçekleştirmek için kullanılan sembollerdir. Örnek olarak toplama, çıkarma ve karşılaştırma verilebilir.
- Kontrol Yapıları: Bir programın akışını kontrol etmek için kullanılan if-else ifadeleri, döngüler ve fonksiyon çağrıları gibi ifadelerdir.
- Kütüphaneler ve Çerçeveler: Ortak görevleri yerine getirmek ve geliştirmeyi hızlandırmak için kullanılan önceden yazılmış kod koleksiyonlarıdır.

2.5.1. Java Programlama Dili

Java, kendisi bir platform olarak kullanılabilen çoklu platform desteğine sahip, nesne yönelimli ve ağ odaklı bir dildir. Mobil uygulamalardan kurumsal yazılımlara, büyük veri uygulamalarından sunucu tabanlı teknolojilere kadar her şey için kodlama yapmak için hızlı, güvenli ve güvenilir bir programlama dilidir.

Java kodunun kaynak kodu, Java derleme işlemi sırasında Bytecode adı verilen ara bir ikili koduna derlenir. Makine bu Bytecode'u doğrudan yürütemez. Bu Bytecode'u anlayan bir sanal makine olan Java Sanal Makinesi veya JVM bulunur. JVM, Bytecode'u hedef bilgisayar makine koduna dönüştüren bir Java yorumlayıcısı içerir. JVM, platforma özgüdür, yani her platformun kendi JVM'si vardır. Ancak, uygun JVM bir kez makineye kurulduğunda, herhangi bir Java Bytecode kodu çalıştırılabilir.

Java uygulama geliştirme ve uygulama, aşağıdaki aşamalara ayrılmaktadır:

- Derleme: Bir derleyici olarak adlandırılan özel bir uygulama, Java programımızı bir sanal Java makinesi (JVM) üzerinde yürütür.
  Derleyici, kaynak kodu JVM bytecode veya JVM tarafından okunan makine koduna dönüştürmektedir.
- Yorumlama: Yorumlama, makine kodunu analiz etme ve çalıştırma işlemidir. Yorumlayıcı, bu görevleri gerçekleştiren benzersiz bir programdır.

2.5.2. Node.js Programlama Dili

JavaScript, dünyadaki en popüler programlama dillerinden bir tanesidir.

Node.js, hızlı ve ölçeklenebilir sunucu tarafı ve ağ uygulamaları oluşturmaya yönelik tek iş parçacıklı, açık kaynaklı, platformlar arası bir çalışma zamanı ortamıdır. V8 JavaScript çalışma zamanı motoru üzerinde çalışmaktadır ve olay odaklı I/O mimarisini kullanmaktadır (Mudholkar, 2024). Bu da Node.js'i verimli ve gerçek zamanlı uygulamalar için uygun kılmaktadır.

Node.js, Lambda işlevi tarafından verimli bir şekilde desteklenen dillerden biridir.

Sunucusuz uygulamalar oluşturmak için Node.js kullanmanın en büyük faydalarından bir tanesi çok sayıda eşzamanlı isteği işleyebilme yeteneğidir. Olay odaklı, I/O modeli sayesinde Node.js, minimum ek yük ile yüzlerce hatta binlerce eşzamanlı bağlantıyı yönetebilmektedir.

Node.js, bir derleme işlemine ihtiyaç duymaz çünkü JavaScript kodunu yorumlayarak çalıştırmaktadır. JavaScript kodun doğrudan bir derleme adımına ihtiyacı yoktur. Node.js, JavaScript dosyalarını çalıştırmak için gerekli olan yorumlayıcıya sahiptir ve kodu çalıştırırken derleme işlemine ihtiyaç duymamaktadır. Bu nedenle, Node.js ile kod yazmak ve çalıştırmak oldukça hızlı ve basittir.

2.6. DynamoDB

DynamoDB, kullanıcıların her miktarda veriyi depolayabilen ve alabilen veritabanları oluşturmasına olanak tanımaktadır ve her miktarda trafiğe hizmet verebilmektedir. Trafiği sunucular üzerinden otomatik olarak dağıtarak yüksek performans sağlamaktadır. Hızlı, sorunsuz bir şekilde ölçeklenebilen, tam olarak yönetilen bir NoSQL veritabanı hizmetidir. (Tillu, 2024)

DynamoDB'nin avantajları aşağıda listelenmektedir:

- DynamoDB, altyapıyı yönetme ihtiyacını ortadan kaldırmaktadır. AWS tarafından tamamen yönetilen bir hizmet olduğu için veritabanıyla ilgili konfigürasyon, yedekleme, güncelleme ve ölçeklendirme gibi operasyonlar AWS tarafından otomatik olarak gerçekleştirilmektedir.

- DynamoDB, yüksek performanslı okuma ve yazma işlemleri sağlamaktadır. İşlem hızı, milisaniye düzeyinde ölçülür ve talebe göre otomatik olarak ölçeklendirilebilmektedir. Hem depolama kapasitesi hem de okuma/yazma kapasitesi dinamik olarak artırılabilir veya azaltılabilir.

- AWS'in altyapısı sayesinde DynamoDB, yüksek oranda dayanıklılık ve güvenilirlik sağlamaktadır. Veriler, birden fazla bölgeye otomatik olarak yedeklenmektedir ve çeşitli felaket kurtarma senaryolarına karşı korunmaktadır.

- DynamoDB'nin maliyet modeli, kullanılan kaynaklar üzerinden esnek bir şekilde ayarlanabilmektedir. Ölçeklenebilir yapısı sayesinde gereksiz kapasiteler üzerinden maliyetler oluşmaz ve ihtiyaca göre ödeme modeli uygulanmaktadır.

- NoSQL veritabanı olarak DynamoDB, esnek veri modeline sahiptir. İhtiyaca göre belirli bir yapıya bağlı kalmadan veri modellenebilmektedir.

AWS tarafından yönetilen altyapı sayesinde, düşük gecikme süreleri ve yüksek erişilebilirlik sağlanmaktadır.

2.7. CloudWatch Metrics

CloudWatch Metrics, AWS'un bulut tabanlı izleme ve yönetim hizmetinin temel bir bileşenidir. AWS CloudWatch, kullanıcıların bulut altyapılarını, uygulamalarını ve hizmetlerini izlemelerine, logları toplamalarına, izlemelerini oluşturmalarına ve analiz etmelerine olanak tanımaktadır.

CloudWatch Metrics, çeşitli AWS hizmetlerinden (örneğin, Amazon EC2, Amazon RDS, AWS Lambda vb.) toplanan verilerin ölçüm değerlerini içermektedir. Bu veriler, işlemci kullanımı, bellek kullanımı, ağ trafiği, depolama kullanımı gibi sistem ve uygulama düzeyindeki performans metriklerini kapsamaktadır. Bu metrikler, bulut altyapısının performansını izlemek, kapasite planlaması yapmak, sorunları tespit etmek ve hizmet düzeyi hedeflerini belirlemek için kullanılabilmektedir.

CloudWatch Metrics'in temel özellikleri şunlardır:

- AWS hizmetlerinden ve özelleştirilmiş uygulamalardan gelen verileri toplamaktadır.
- Metrik verileri AWS CloudWatch'da depolanmaktadır.
- Kullanıcıların gerçek zamanlı olarak metrik verilerini izlemesine olanak tanımaktadır.
- Metrik verileri kullanarak AWS kaynaklarını otomatik olarak ölçeklendirebilmektedir.
- Belirli metrikler belirli eşik değerlerini aştığında bildirimler göndermek için alarmlar oluşturabilmektedşr.
- Olaylar (events) ve değişiklikler hakkında bilgi sağlamaktadır.

Genel olarak, CloudWatch metrikleri, AWS ortamında performansı izlemek ve yönetmek için kritik bir araçtır ve işletmelerin bulut altyapılarını etkin bir şekilde çalıştırmalarına yardımcı olmaktadır.

BÖLÜM III

Gereç ve Yöntem

Bu araştırma, deneysel bir çalışma olacak ve kıyaslama yapılacak olan iki programlama dili olan Node.js ve Java'nın AWS Lambda performansına etkileri değerlendirilecektir.

3.1. Araştırmanın Planı

Proje kapsamında AWS Lambda fonksiyonları için kullanılan programlama dili farklılıklarının Lambda fonksiyonun perfomansına etkisi incelenmektedir.

Node js ve Java programlama dilleri kullanılarak 2 tane Lambda fonksiyonu oluşturulmuştur. Her iki lambda fonksiyonuda aynı iş yüküne sahiptir. AWS Lambda fonksiyonu oluşturulurken bellek olarak ise 512 MB seçilmiştir.

Lambda fonksiyonlarında veritabanı işlemlerini yapmak için AWS'in DynamoDB servisi kullanılmıştır. Bunun için öncelikle DynamoDB üzerinde "Customer_Test" tablosu oluşturulmuştur. Tabloda partition key olarak "CustomerId" kullanılmıştır. Her iki dilde yazılan Lambda fonksiyonları ile "Customer_Test" tablosu üzerinde veritabanı işlemleri yapılmıştır.

3.2. Araştırmanın Evreni

Bu araştırmanın evreni, AWS Lambda platformunda çalıştırılabilen Node.js ve Java programlama dilleriyle yazılmış Lambda fonksiyonlarını içermektedir. Bu araştırma, AWS Lambda üzerinde çalışan tüm Node.js ve Java fonksiyonlarının performansını anlamak için tasarlanmıştır.

Bu fonksiyonlar, farklı işlevlere sahip olabilir ve farklı performans karakteristiklerine sahip olabilir. Araştırmanın evreni, AWS Lambda platformunun genel özelliklerini ve Node.js ve Java fonksiyonlarının performansını kapsamaktadır.

3.3. Veri Toplama Araçları

Araştırmada kullanılan veri toplama araçları AWS CloudWatch Metrics, DynamoDB ve AWS Lambda'dır.

3.4. Verilerin Değerlendirilmesi

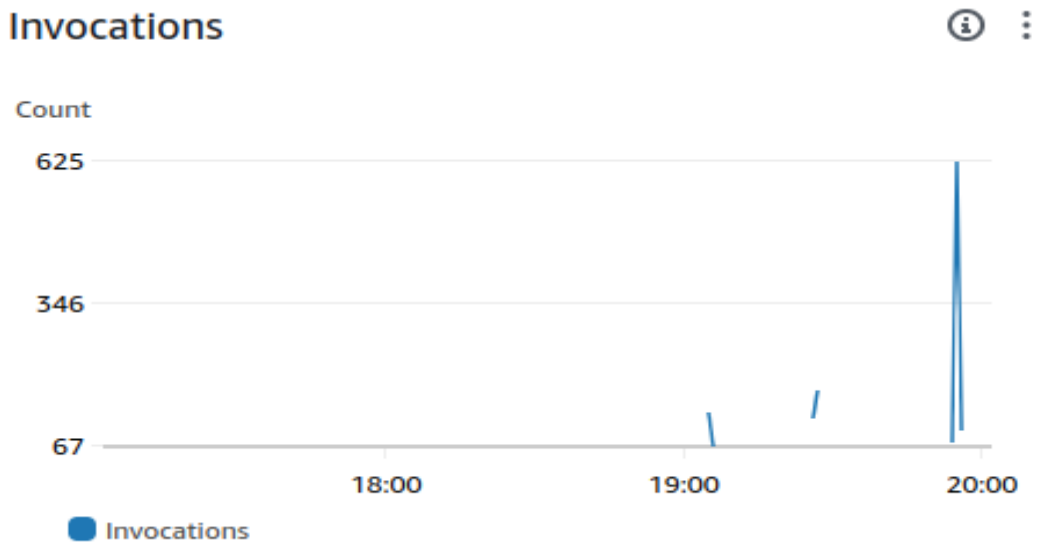
Oluşturulan AWS Lambda fonksiyonlarına yapılan çağrılar sonucunda AWS CloudWatch Metrics tarafından bellek kullanımı, çağrı sayısı, çalışma süresi ve ölçeklenebilirliğin gözlemlenebileceği grafikler oluşmuştur. Java ve Node.js programlama dilleri kullanılarak yazılan Lambda fonksiyonlarına Postman üzerinden yinelemeli çağrılar yapılmıştır. Bu çağrılar sonucunda fonksiyonların yanıt sürelerini kapsayan bir rapor oluşturulmuştur. Her iki Lambda fonksiyonunun yanıt süreleri incelenerek performans karşılaştırması yapılmıştır.

BÖLÜM IV

Bulgular

4.1. AWS Lambda Java Fonksiyonu Cloudwatch Metriklerinin Değerlendirilmesi

AWS Lambda performansını ölçmek için java programlama dili kullanılarak DynamoDB'de oluşturulan tablodan veri çekme, ekleme ve çıkarma işlemleri yapılmıştır. Veritabanı işlemleri yapılan Java Lambda fonksiyonunu 200 defa çağırdığımızda oluşan AWS Lambda servisinin cloudwatch performans metrikleri grafikleri aşağıda değerlendirilmiştir.



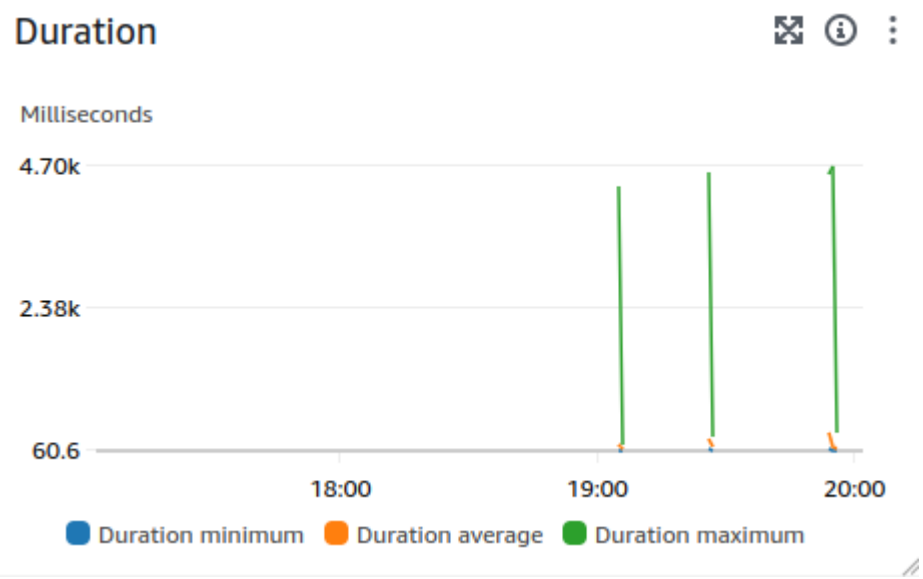
Şekil 2: AWS Lambda Java Çağrı Grafiği

Şekil 2'de yer alan grafikte, belirli saat dilimlerinde lambda fonksiyonuna yapılan çağrıları gösteren bir grafiktir. Grafik, saat 18:00'den 20:00'e kadar olan zaman aralığını kapsamaktadır.

Şekil 2'deki grafiği incelediğimizde:

- Saat 19:00 civarında çağrı sayısının artarak yaklaşık 150'ye ulaştığı görülmektedir.
- Saat 20:00 civarında çağrı sayısının keskin bir artış göstererek 625'e ulaştığı görülmektedir.

Şekil 2’deki grafik, belirli saatlerdeki çağrı aktivitesindeki değişiklikleri ve zirve noktalarını görmek için kullanışlı bir araçtır. Özellikle saat 20:00'deki keskin artış, bu saatte bir aktivite yoğunluğu ya da özel bir olay olabileceğine işaret etmektedir.



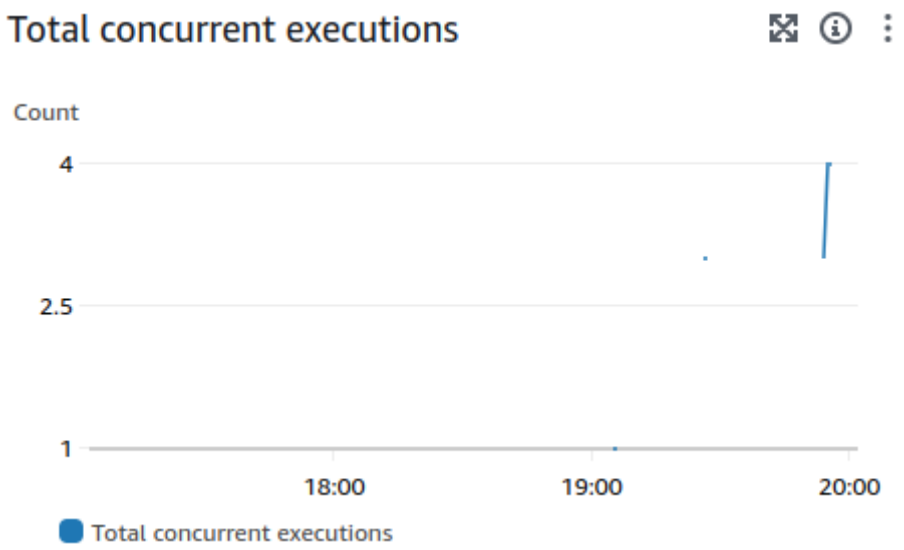
Şekil 3: AWS Lambda Java Yanıt Süresi Grafiği

Şekil 3’de yer alan grafikte fonksiyonun yanıt süresinin farklı zamanlarda nasıl değiştiğini gösteren üç ana değer (minimum, ortalama, maksimum) bulunmaktadır. Grafik milisaniye birimini kullanarak süre ölçümlerini ifade etmektedir.

Şekil 3’deki grafikteki verilere bakarak:

- Saat 19:00'da, minimum süre 60.6 milisaniye civarında, maksimum süre ise 2.38k ile 4.70k arasında bir değer almaktadır.
- Saat 20:00'da, gösterilen değerler saat 18:00 ile benzer yanıt sürelerine sahiptir. Burada da minimum süre 60.6 milisaniye civarında, maksimum süre ise yine 4.70k (4700 milisaniye) civarındadır.

Şekil 3’deki grafikte, özellikle bir sürecin performansını ve yanıt süresinin günün farklı zamanlarında nasıl değiştiğini izlemek için faydalı olabilmektedir. Maksimum sürelerin yüksek olduğu noktalar sistem performansında bir darboğaz olduğunu veya belirli saatlerde yoğunluk olduğunu gösterebilir.



Şekil 4: AWS Lambda Java Toplam Eşzamanlı Yürütme Sayısı Grafiği

Şekil 4'de yer alan grafikte, toplam eşzamanlı yürütme sayısını göstermektedir. Grafikte zamana bağlı olarak eşzamanlı yürütme sayısını (0'dan 4'e kadar) göstermektedir. Grafiğin büyük bir kısmında herhangi bir değişiklik görünmezken, 19:00 civarında yürütme sayısında keskin bir artış gösteren bir sıçrama olduğu gözlemlenmiştir ve eşzamanlı yürütme sayısının 4'e ulaştığı görülmüştür. Bu, belirtilen saatte sistemin eşzamanlı olarak dört işlem yürüttüğünü belirtmektedir.

Şekil 5: AWS Lambda Bellek Kullanım Grafiği

Şekil 5'de yer alan bellek kullanım grafiği, bir AWS Lambda fonksiyonunun zaman içindeki bellek kullanımını göstermektedir.

Şekil 5’deki grafikten yapılan gözlemler:

- Zaman periyodunun büyük bölümünde (15:00'dan 20:00'a kadar) bellek kullanımı kararlı ve neredeyse sabittir. Yüzde 31'in biraz üzerinde seyretmektedir.
- Saat 20:00 civarında bellek kullanımında önemli bir artış gözlemlenmiştir ve bellek kullanımı keskin bir şekilde %32''ye çıkmaktadır.

Şekil 5’deki grafik, Lambda fonksiyonunun belirtilen zaman diliminin büyük bölümünde tahsis edilen hafızanın yaklaşık %31'ini tutarlı bir şekilde kullandığını ve ardından saat 20:00'de bellek talebinde ani bir artış olduğunu göstermektedir. Bu ani artış, iş yükündeki ani bir artıştan veya daha fazla bellek gerektiren belirli bir işlemden kaynaklanıyor olabilir.

Şekil 6: AWS Lambda Java CPU Kullanım Grafiği

Şekil 6’da yer alan grafik, belirli bir zaman aralığında bir bilgisayarın CPU kullanımını gösteren bir kullanım grafiğidir.

Şekil 6’daki grafiğe bakıldığında, CPU kullanım süresinin büyük bir kısmında oldukça düşük seviyelerde olduğu görülmekte; bununla birlikte 19:00'a doğru bir zaman diliminde CPU kullanım süresinde keskin bir artış yaşandığı, bu sürenin neredeyse 1.19k milisaniyeye (yaklaşık 1190 milisaniye) çıktığı gözlemlenmektedir. Bu durum anlık bir performans artışı veya yoğun işlem gerektiren bir uygulamanın çalıştırılmasından kaynaklanabilmektedir.

Bu tür grafikler, sistem performansını izlemek ve potansiyel sorunları belirlemek için kullanılmaktadır. Örneğin, bu grafiği analiz eden bir sistem yöneticisi, 19:00 civarında meydana gelen yüksek CPU kullanımını araştırmak isteyebilir.

4.2. AWS Lambda Node.js Fonksiyonu Cloudwatch Metriklerinin Değerlendirilmesi

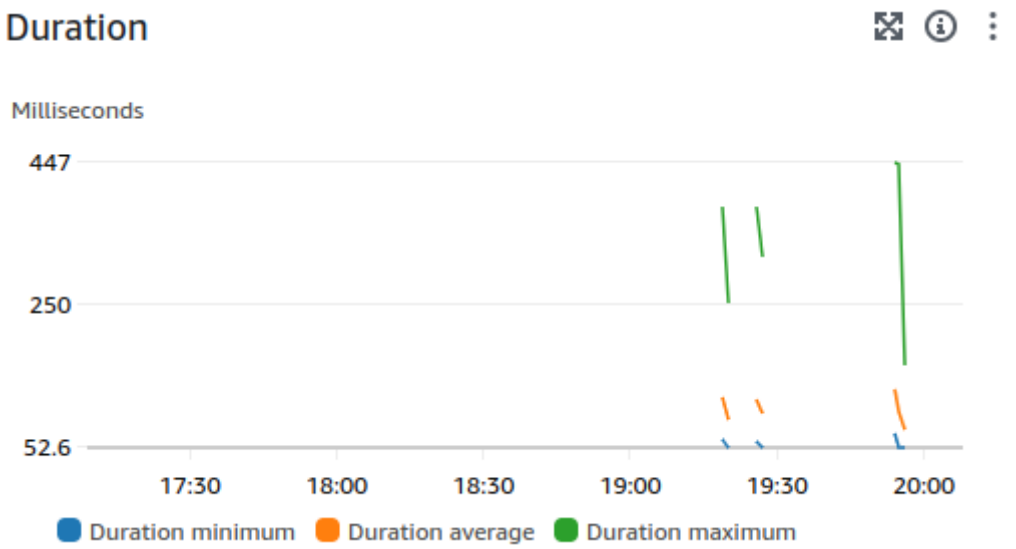
AWS Lambda performansını ölçmek için Node.js programlama dili kullanılarak DynamoDB'de oluşturulan tablodan veri çekme, ekleme ve çıkarma işlemleri yapılmıştır. Veritabanı işlemleri yapılan Node.js Lambda fonksiyonunu 200 defa çağırdığımızda oluşan AWS Lambda servisinin cloudwatch performans metrikleri grafikleri aşağıda değerlendirilmiştir.

Şekil 7: AWS Lambda Node.js Çağrı Grafiği

Şekil 7'de yer alan grafik, belirli bir zaman aralığında yapılan çağrı sayısını gösteren bir zaman serisi grafiğidir. Grafiğe göre, saat 20:00 civarında çağrı sayısında ani bir artış gözlemlenmektedir. Bu pik noktasında 625 çağrıya ulaşılmıştır. Bu artıştan önce 19:30 civarında da küçük bir artış gözlemlenmektedir. Diğer zaman dilimlerinde ise çağrı sayıları daha düşük ve nispeten sabit kalmıştır.

Şekil 7'deki grafiği incelediğimizde:

- Saat 19:00 civarında çağrı sayısı artarak yaklaşık 180'e ulaştığı görülmektedir.
- Saat 20:00 civarında çağrı sayısının keskin bir artış göstererek 625'e ulaştığı görülmektedir.



Şekil 8: AWS Lambda Node.js Yanıt Süresi Grafiği

Şekil 8'de yer alan grafikte fonksiyonun yanıt süresinin farklı zamanlarda nasıl değiştiğini gösteren üç ana değer (minimum, ortalama, maksimum) bulunmaktadır. Çizgi grafiği üç farklı renkte veri serisini göstermektedir. Bunlar mavi, turuncu ve yeşil renkleridir. Bu renkler sırasıyla minimum süre, ortalama süre ve maximum süre değerlerini temsil etmektedir.

Şekil 8'deki grafik zaman içinde süre parametrelerinin nasıl değiştiğini göstermektedir. Saat19:00'dan sonra maximum süre değerinde büyük bir artış olduğu ve en yüksek değerin 20:00 civarında gerçekleştiği görülmektedir. Diğer taraftan, minimum süre ve ortalama süre değerlerinin daha düşük ve daha stabil olduğu gözlemlenmektedir.

Şekil 9: AWS Lambda Node.js Toplam Eşzamanlı Yürütme Sayısı Grafiği

Şekil 9’da yer alan grafik, belirli bir zaman aralığında yapılan toplam eşzamanlı yürütme sayısını göstermektedir.

Şekil 9’daki grafik, 19:00 civarında bir noktada dikey olarak 4 değerine ulaşan keskin bir artış göstermektedir ve bu da total eşzamanlı yürütmeler sayısının bu zamanda 4 olduğunu ifade etmektedir. Bu, genellikle bir sistemde, belirli bir zaman diliminde yürütülen görev ya da işlemlerin sayısını izlemek için kullanılan bir grafik türüdür.

Şekil 10: AWS Lambda Node.js Bellek Kullanım Grafiği

Şekil 10’da yer alan grafikte, belirli bir zaman aralığında yüzde (%) cinsinden ölçeklendirilmiş bir hafıza kullanım grafiği görülmektedir.

Şekil 10’daki grafiğin gösterim aralığı %24 ile %25 arasında oldukça dar bir yüzdelik dilimde gerçekleşmektedir ve grafiğin neredeyse sabit bir değerde, yani %24,5 civarında olduğunu göstermektedir. Bu, ölçülen zaman dilimi boyunca hafıza kullanımının neredeyse sabit kaldığını ve önemli bir değişiklik göstermediğini ifade etmektedir. Bu tür grafikler, sistem performansını izlemek ve potansiyel sorunları tespit etmek için kullanılmaktadır.

Şekil 11: AWS Lambda Node.js CPU Kullanım Grafiği

Şekil 11’de yer alan grafik, belirli bir gün içinde belli saatlerde CPU kullanımını milisaniye cinsinden göstermektedir. Grafik, saat 15:00'den 20:00'e kadar olan süreyi kapsamaktadır.

Şekil 11'deki grafikte, saat 20:00 civarında CPU kullanımının ani bir şekilde 160 milisaniyeden fazla arttığı görülmektedir. Bu, işlemci aktivitesinde büyük bir artışa işaret edebilmektedir. Bu tür grafikler, fonksiyonun performansını izlemek ve potansiyel olarak gerekli iyileştirmeleri veya sorun gidermeleri yapmak için kullanılmaktadır.

4.3. AWS Lambda Java ve Node.js Fonksiyonlarının Yanıt Sürelerine Göre Performans Değerlendirmesi

Ekte yer alan yanıt süreleri incelendiğinde, fonksiyona yapılan ilk çağrı bir AWS Lambda fonksiyonunun soğuk başlatma durumundaki davranışını göstermektedir. Soğuk başlatma, bir Lambda işlevinin ilk kez çağrıldığında veya belirli bir süre boyunca çağrılmadığında gerçekleşen bir durumdur. Fonksiyonun ilk kez çağrılması veya belirli bir süre boyunca çağrılmaması durumunda, AWS altyapısı fonksiyonu yükler ve başlatır. Bu işlem, normal çalışma sürecinden daha uzun sürebilir ve genellikle Lambda fonksiyonunun performansını etkileyen bir faktördür. Daha sonra yapılan ardışık çağrılarda lambda fonksiyonu daha hızlı yanıt verebilir. Bu durum ise sıcak başlatma olarak isimlendirilmektedir.

Ekte yer alan Java Lambda fonksiyonunun yanıt süreleri incelendiğinde ilk çağrımda yanıt süresinin 7841 ms olduğu gözlemlenmiştir. Java Lambda fonksiyonuna yapılan ikinci çağrıda ise yanıt süresinin 475 ms olduğu gözlemlenmiştir. İki çağrının yanıt süreleri arasında ciddi bir fark olduğu gözlemlenmiştir. Bu durum Javanın soğuk başlatma durumunda performansının düşük olduğunu göstermektedir. İkinci çağrıdan sonraki çağrılarda Java Lambda fonksiyonu hızlı yanıt sürelerine sahiptir. Bu durumda Java Lambda fonksiyonu yüklenmiş ve çalışır durumdadır.

Ekte yer alan Node.js Lambda fonksiyonunun yanıt süreleri incelendiğinde ilk çağrımda yanıt süresinin 1073 ms olduğu gözlemlenmiştir. Node.js Lambda fonksiyonuna yapılan ikinci çağrıda ise yanıt süresinin 284 ms olduğu gözlemlenmiştir. İki çağrının yanıt süreleri arasında Java fonksiyonuna kıyasla çok daha az bir fark olduğu gözlemlenmiştir. Bu durum Node.js Lambda fonksiyonun soğuk başlatma durumunda performansının Java'ya kıyasla çok daha yüksek olduğu gözlemlenmektedir. İkinci çağrıdan sonraki çağrılarda Node.js Lambda fonksiyonu daha hızlı yanıt sürelerine sahiptir. Bu durumda Node.js lambda fonksiyonu yüklenmiş ve çalışır durumdadır.

Node.js ve Java Lambda fonksiyonları için sıcak başlatma sürelerini kıyasladığımızda bu sürelerin her iki fonksiyonda benzer sürelere sahip olduğunu gözlemlenmektedir. Her iki fonksiyonda sıcak başlatma durumunda hızlı yanıt sürelerine sahiptir.

Soğuk başlatma durumunu hafifletmek için bazı optimizasyonlar yapılabilir. Örneğin, fonksiyonun bellek kullanımını ve dış bağımlılıklarını azaltmak için kodu optimize edebilir veya fonksiyonu düzenli olarak çağırarak soğuk başlatma durumunu en aza indirebiliriz.

Performans değerlendirmesi:

- Verilen yanıt sürelerinin oldukça değişken olduğu görülüyor. Bazı çağrılar çok kısa sürede tamamlanırken, bazıları daha uzun sürelerde tamamlanıyor. Bu değişkenlik, fonksiyonun çalıştığı çevreye, kullanılan programlama diline, çağrıların içeriğine ve fonksiyonun kendi iç mantığına bağlı olabilmektedir. Java Lambda fonksiyonunda yanıt sürelerinin Node.js Lambda fonksiyonuna göre daha yüksek değerler aldığı gözlemlenmiştir.
- Birkaç çağrının oldukça uzun süre aldığı görülmektedir. Java Lambda fonksiyonu için özellikle 7.841 ms, Node.js Lambda fonskiyonu için ise 1073 ms gibi yüksek bir süreye sahip olduğu gözlemlenmiştir. Bu durum fonksiyonun beklenen performansını ciddi şekilde etkileyebilir. Bu uzun sürelerin nedenleri incelenmeli ve gerekirse iyileştirmeler yapılmalıdır.
- Bazı çağrıların sürelerinin iyileştirilmesi gerekebilir. Bu süreleri azaltmak için kod optimizasyonu, gereksiz işlemlerin kaldırılması veya dış bağımlılıkların iyileştirilmesi gibi adımlar düşünülebilir.
- Bazı çağrılar, belirli bir aralıkta tutarlı bir şekilde benzer sürelerde tamamlanıyor gibi görünmektedir. Bu istikrarlı performans, fonksiyonun belirli koşullar altında ne kadar sürede çalışabileceği hakkında bir fikir vermektedir.

4.4. Sonuç

Proje kapsamında AWS Lambda üzerinde kullanılan Java ve Node.js programlama dillerinin karşılaştırmaları sonucunda performans, soğuk başlatma, sıcak başlatma konularında değerlendirmeler yapılmıştır.

Java ve Node.js ile yazılan AWS Lambda fonksiyonları sıcak başlatma durumunda iyi bir performans sunmaktadır. Ancak bu fonksiyonların soğuk başlatma durumunda performanslarını değerlendirildiğinde Java fonksiyonunun performansının Node.js'e kıyasla çok daha düşük olduğu gözlemlenmiştir.

AWS Lambda'nın Java'da soğuk başlatma süresi, diğer dillere kıyasla daha yüksek olmasının nedenleri aşağıda listelenmektedir:

- Java, çalışma zamanında kodun derlenmesi için JIT derlemesi kullanır. Bu, kodun ilk çalıştırılmasında derleme süresinin ve işlevin başlatılmasının gecikmesine neden olabilmektedir.
- Java, tip güvenliği, hata denetimi ve diğer güvenlik önlemleri gibi özellikler sunmaktadır. Bu, Java'nın uygulamanın başlatılması ve yürütülmesi için daha fazla kaynağı gerektirebilir ve dolayısıyla soğuk başlatma süresini artırabilir.
- Java, genellikle büyük ve karmaşık kütüphaneleri ve bağımlılıkları içermektedir. Bu kütüphanelerin ve bağımlılıkların yüklenmesi ve başlatılması, soğuk başlatma süresini artırabilmektedir.

- Java, JVM üzerinde çalışmaktadır ve JVM'nin başlatılması ve yüklenmesi zaman alabilir. Bu da soğuk başlatma süresini etkileyebilmektedir (Smith ve Sailes, 2022).

- Diğer dillerde olduğu gibi Java'nın Lambda üzerinde çalışması için optimize edilmesi ve önbellek kullanımı, soğuk başlatma süresini azaltmaya yardımcı olabilir, ancak bu işlem genellikle diğer dillere göre daha uzun sürebilmektedir.

Bununla birlikte, AWS Lambda, Java'nın soğuk başlatma süresini minimize etmek için sürekli olarak iyileştirmeler yapmaktadır ve bazı optimizasyonlar ile Java'da Lambda kullanımının performansını arttırabilmektedir. Örneğin, Lambda fonksiyonunu küçük parçalara bölmek, fonksiyonların soğuk başlatma süresini azaltabilmektedir. Ayrıca, AWS Lambda'nın yeni özelliklerini ve güncellemelerini takip etmek, performansı artırmamıza yardımcı olabilmektedir.

Kaynaklar

- Benjamin Smith & Mark Sailes. (2022). Optimizing AWS Lambda function performance for Java. https://aws.amazon.com/tr/blogs/compute/optimizing-aws-lambda-function-performance-for-java/
- Cliff Crosland. (2023). Serverless Speed: Rust vs. Go, Java, and Python in AWS Lambda Functions. https://blog.scanner.dev/serverless-speed-rust-vs-go-java-python-in-aws-lambda-functions/
- James Beswick. (2021). Operating Lambda: Performance optimization – Part 1. https://aws.amazon.com/tr/blogs/compute/operating-lambda-performance-optimization-part-1/
- Manik Mudholkar. (2024). The V8 JavaScript Engine. https://medium.com/@manikmudholkar831995/the-v8-javascript-engine-d1434ca77c96
- Liam Cleary. (2024). Top benefits and disadvantages of serverless computing. https://www.techtarget.com/searchcloudcomputing/tip/Top-benefits-and-disadvantages-of-serverless-computing
- Jay Tillu. (2024). What is Amazon DynamoDB?. https://jaytillu.medium.com/what-is-amazon-dynamodb-03f6861b62d4

Ekler

AWS Lambda Java Kodu:

```
package javalambda;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.DeleteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.spec.PutItemSpec;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.Item;
/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<Object, Object> {
```

```
    private static final String TABLE_NAME = "Customer_Test";
    private static final DynamoDB dynamoDB = new
DynamoDB(AmazonDynamoDBClientBuilder.defaultClient());
    @Override
    public String handleRequest(Object input, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("Event: " + input);
        try {
            List<Map<String, Object>> items = getCustomerList(logger);
            logger.log("Customer List: " + items);
            deleteCustomer("9099844444", logger);
            putCustomerInfo(logger);
            deleteCustomer("9099844444", logger);
            return "Success.";
        } catch (Exception e) {
            logger.log("Error: " + e.getMessage());
            throw new RuntimeException("Error: " + e.getMessage());
        }
    }
    private List<Map<String, Object>> getCustomerList(LambdaLogger logger) {
        Table table = dynamoDB.getTable(TABLE_NAME);
        List<Map<String, Object>> allItems = new ArrayList<>();
        table.scan().pages().forEach(page -> {
            List<Map<String, Object>> pageItems = new ArrayList<>();
            page.forEach(item -> {
                pageItems.add(item.asMap());
            });
            allItems.addAll(pageItems);
        });
        return allItems;
```

```
    }
    private void deleteCustomer(String customerId, LambdaLogger logger) {
        Table table = dynamoDB.getTable(TABLE_NAME);
        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
                .withPrimaryKey("CustomerId", customerId);
        try {
            DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);
            logger.log("Customer ID " + customerId + " successfully deleted.");
        } catch (Exception e) {
            logger.log("Error deleting customer ID " + customerId + ": " + e.getMessage());
            throw new RuntimeException("Error deleting customer ID " + customerId + ": " +
e.getMessage());
        }
    }
private void putCustomerInfo(LambdaLogger logger) {
    List<Item> customerList = new ArrayList<>();
        customerList.add(newCustomerMap("1264444372", "Emily", "Doe",
"john.doe@example.com"));
        customerList.add(newCustomerMap("2766743243", "Jane", "Smith",
"jane.smith@example.com"));
        customerList.add(newCustomerMap("3264446237", "Michael", "Johnson",
"michael.johnson@example.com"));
        customerList.add(newCustomerMap("3474448474", "Emily", "Williams",
"emily.williams@example.com"));
        customerList.add(newCustomerMap("6434446474", "David", "Brown",
"david.brown@example.com"));
        customerList.add(newCustomerMap("9099844444", "Sarah", "Jones",
"sarah.jones@example.com"));
        customerList.add(newCustomerMap("9054894384", "James", "Taylor",
"james.taylor@example.com"));
        customerList.add(newCustomerMap("9032844484", "Anna", "Martinez",
"anna.martinez@example.com"));
```

```
        customerList.add(newCustomerMap("4747444484", "Robert", "Garcia",
"robert.garcia@example.com"));
        customerList.add(newCustomerMap("8944458544", "Emma", "Lopez",
"emma.lopez@example.com"));
    Table table = dynamoDB.getTable(TABLE_NAME);
    for (Item customer : customerList) {
        try {
            PutItemSpec putItemSpec = new PutItemSpec().withItem(customer);
            table.putItem(putItemSpec);
        } catch (Exception e) {
            logger.log("Error putting data: " + e.getMessage());
            throw new RuntimeException("Error putting data: " + e.getMessage());
        }
    }
}
private Item newCustomerMap(String customerId, String customerName, String
customerSurname, String email) {
     Item item = new Item()
            .withPrimaryKey("CustomerId", customerId)  // Primary key column and value
            .withString("customerName", customerName)  // Additional columns and values
            .withString("customerSurname", customerSurname)
            .withString("email", email);
    return item;
}
}
```

AWS Lambda Node.js Kodu

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import {
  DynamoDBDocumentClient,
  PutCommand,
```

```
  ScanCommand,
  DeleteCommand
} from '@aws-sdk/lib-dynamodb';
const dynamoDBClient = DynamoDBDocumentClient.from(new DynamoDBClient({
region: 'eu-central-1' }), {
  marshallOptions: {
    removeUndefinedValues: true
  }
});
export const handler = async (event) => {
  const customerList = await getCustomerList();
  console.log(customerList);
  await deleteCustomer('126372')
  await putCustomerInfo();
  const response = {
    statusCode: 200,
    body: JSON.stringify('Success.'),
  };
  return response;
};

async function getCustomerList(){
  const { Item } = await dynamoDBClient.send(
    new ScanCommand({
      TableName: 'Customer_Test'
    })
  );
  return Item;
}
async function deleteCustomer(customerId){
```

```
    const params = {
      TableName: 'Customer_Test',
      Key: {
        CustomerId: customerId
      }
    };
  await dynamoDBClient.send(new DeleteCommand(params));
}
async function putCustomerInfo() {
  try {
      const customerList = [
    {
        customerId: '126372',
        customerName: "Emily",
        customerSurname: "Doe",
        email: "john.doe@example.com"
    },
    {
        customerId: '276673243',
        customerName: "Jane",
        customerSurname: "Smith",
        email: "jane.smith@example.com"
    },
    {
        customerId: '32646237',
        customerName: "Michael",
        customerSurname: "Johnson",
        email: "michael.johnson@example.com"
    },
```

```
{
   customerId: '347874',
   customerName: "Emily",
   customerSurname: "Williams",
   email: "emily.williams@example.com"
},
{
   customerId: '643674',
   customerName: "David",
   customerSurname: "Brown",
   email: "david.brown@example.com"
},
{
   customerId: '909984',
   customerName: "Sarah",
   customerSurname: "Jones",
   email: "sarah.jones@example.com"
},
{
   customerId: '905489438',
   customerName: "James",
   customerSurname: "Taylor",
   email: "james.taylor@example.com"
},
{
   customerId: '9032848',
   customerName: "Anna",
   customerSurname: "Martinez",
   email: "anna.martinez@example.com"
```

```
  },
  {
     customerId: '474748',
     customerName: "Robert",
     customerSurname: "Garcia",
     email: "robert.garcia@example.com"
  },
  {
     customerId: '895854',
     customerName: "Emma",
     customerSurname: "Lopez",
     email: "emma.lopez@example.com"
  }
];
  for (const customer of customerList) {
    const params = {
     TableName: 'Customer_Test',
     Item: {
       CustomerId: customer.customerId,
       customerName: customer.customerName,
       customerSurname: customer.customerSurname,
       email: customer.email
     }
    };
    await dynamoDBClient.send(new PutCommand(params));
  }
} catch (error) {
  console.error('Put data error:', error);
  throw new Error('Put data error.');
```

```
  }
}
```


AWS Lambda Java Response Times


```
{
  "results": [
    {
      "id": "2b9cd0a3-1bed-4db1-9bba-28e642e2c378",
      "name": "AWS Lambda Java",
      "time": 171,
      "responseCode": {
        "code": 200,
        "name": "OK"
      },
      "times": [
        7841,  475,  334,  295,  265,  317,  394,  398,  293,
249,  323,  393,  214,  275,  213,  271,  300,  288,  297,
300,  251,  329,  397,  262,  167,  243,  175,  206,  216,
234,  234,  229,  252,  191,  294,  473,  9848,  161,  234,
292,  229,  256,  213,  262,  291,  297,  240,  167,  169,
290,  220,  376,  291,  205,  592,  288,  230,  258,  219,
372,  191,  212,  160,  165,  331,  157,  222,  170,  178,
169,  230,  163,  173,  169,  165,  145,  168,  152,  226,
174,  170,  175,  162,  200,  175,  263,  295,  291,  194,
226,  243,  158,  259,  286,  223,  161,  389,  252,  241,
289,  228,  253,  306,  158,  210,  209,  276,  293,  292,
159,  148,  153,  314,  169,  138,  153,  153,  145,  134,
252,  138,  245,  295,  144,  162,  146,  158,  359,  396,
144,  286,  158,  150,  144,  155,  140,  147,  142,  150,
145,  127,  141,  137,  148,  286,  174,  150,  147,  128,
148,  131,  148,  128,  225,  162,  327,  154,  146,  130,
174,  151,  215,  154,  142,  259,  157,  165,  252,  153,
139,  159,  155,  183,  146,  157,  146,  182,  150,  130,
228,  139,  138,  159,  134,  254,  147,  236,  388,  246,
194,  151,  142,  134,  128,  248,  240,  360,  141,  139,
171
```

```
    ]
   }
  ],
  "count": 200,
  "totalTime": 60830,
  "collection": {
   "requests": [
    {
     "id": "2b9cd0a3-1bed-4db1-9bba-28e642e2c378",
     "method": "GET"
    }
   ]
  }
}
```


AWS Lambda Node.js Response Times


```
{
  "results": [
   {
    "id": "6ff68604-adb5-4fe6-ab82-04cb05ed486d",
    "name": "AWS Lambda Node.js",
    "time": 217,
    "responseCode": {
     "code": 200,
     "name": "OK"
    },
    "times": [
      1073,      284,      150,      229,      222,      174,      242,      225,      190,
166,      185,      321,      196,      200,      385,      155,      317,      240,      161,
```

```
149,  212,  217,  253,  147,  286,  327,  139,  207,  157,
142,  285,  139,  357,  164,  216,  161,  160,  251,  294,
203,  157,  466,  248,  223,  166,  193,  176,  249,  134,
200,  145,  151,  186,  150,  272,  149,  148,  144,  213,
152,  230,  142,  214,  146,  150,  129,  148,  268,  125,
137,  153,  258,  310,  146,  127,  324,  136,  172,  234,
397,  134,  247,  150,  648,  295,  157,  414,  171,  143,
391,  233,  148,  136,  145,  163,  141,  355,  159,  226,
296,  141,  240,  142,  139,  143,  151,  147,  223,  128,
137,  318,  147,  121,  207,  145,  237,  295,  122,  168,
161,  130,  145,  161,  154,  135,  157,  266,  128,  212,
135,  126,  189,  188,  127,  179,  151,  134,  128,  127,
128,  192,  130,  145,  295,  377,  144,  130,  124,  197,
138,  124,  131,  142,  163,  144,  129,  142,  176,  154,
194,  140,  129,  267,  143,  120,  115,  147,  131,  122,
136,  148,  267,  170,  124,  189,  147,  152,  124,  127,
171,  143,  130,  138,  264,  126,  153,  136,  135,  145,
138,  130,  173,  151,  126,  137,  166,  126,  127,  138,
217
      ]
    }
  ],
  "count": 200,
  "totalTime": 38252,
  "collection": {
    "requests": [
      {
        "id": "6ff68604-adb5-4fe6-ab82-04cb05ed486d",
        "method": "GET"
      }
    ]
  }
}
```