Migração Parcial de um Banco de Dados Relacional para um Banco de Dados NoSQL na Nuvem Através de Adaptações Não-intrusivas: Um Relato de Experiência

Caio H. Costa[1], **Lincoln S. Rocha**[2], **Nabor C. Mendonça**[3], **Paulo Henrique. M. Maia**[1]

[1]Mestrado Acadêmico em Ciências da Computação (MACC)
Universidade Estadual do Ceará (UECE)
Campus do Itaperi, Av. Paranjana, 1700, CEP 60740-903 Fortaleza - CE

[2]Universidade Federal do Ceará (UFC)
Campus de Quixadá, Av. José de Freitas Queiroz, 5002, CEP 63900-00 Quixadá - CE

[3]Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321, Edson Queiroz, CEP 60811-905 Fortaleza - CE

caiohc@gmail.com, lincolnrocha@ufc.br, nabor@unifor.br, pauloh.maia@uece.br

Abstract. *This paper reports the experience of partially migrating the relational database of a legacy system to a NoSQL database in the cloud in order to resolve performance issues. Part of the relational database data was adapted and migrated to the NoSQL database. The adaptations of the two applications that make up the system were performed using a non-intrusive approach based on aspect-oriented programming and dynamic features of the Groovy programming language. After the necessary adjustments, the system became able to simultaneously access the relational database and the new DynamoDB database on Amazon cloud.*

Resumo. *Este trabalho relata a experiência de migração parcial do banco de dados relacional de um sistema legado para um banco de dados NoSQL na nuvem, a fim de solucionar problemas relacionados a desempenho. Parte dos dados do banco de dados relacional foi adaptada e migrada para o banco de dados NoSQL. As adaptações das duas aplicações que compõem o sistema foram realizadas por meio de uma abordagem não-intrusiva baseada em programação orientada a aspectos e funcionalidades dinâmicas da linguagem de programação Groovy. Após as adaptações necessárias, as duas aplicações passaram a acessar simultaneamente a base de dados relacional e a nova base de dados DynamoDB, na nuvem da Amazon.*

1. Introdução

Este trabalho relata a migração parcial da base de dados relacional de uma aplicação legada para um banco de dados NoSQL na nuvem, a fim de solucionar problemas de desempenho oriundos do crescimento exponencial da sua maior, e mais importante, tabela. Trata-se de um sistema de monitoramento veicular composto de duas aplicações: uma *web*, chamada RastroBR, e outra *standalone*, chamada RBRDriver.

Ambas as aplicações compartilham o conceito de **posição** na descrição dos seus domínios. Uma posição caracteriza a localização georeferenciada de um veículo monitorado pelo sistema. Os dados relativos à posição do veículo são enviados a uma base de dados em intervalos fixos de um minuto. Como consequência do contínuo aumento no número de veículos monitorados, o volume de dados armazenados na tabela `posicao` passou a crescer em uma taxa exponencial. Desse modo, em conseqüência do tamanho[1] e do rápido crescimento da tabela `posicao`, operações de consulta e de inserção de registros nessa tabela tonaram-se cada vez mais lentas. Para amenizar esse problema, um escalonamento vertical no servidor de banco de dados foi realizado algumas vezes. Porém, além dessa ser uma solução cara, ela apresenta limites práticos de implementação.

De acordo com [Sadalage and Fowler 2013], bancos de dados relacionais não foram projetados para serem escalonados horizontalmente. Já os bancos de dados NoSQL orientados a agregados, por outro lado, foram projetados para lidar com grandes volumes de dados por meio do escalonamento horizontal. Desse modo, uma alternativa baseada em um banco de dados NoSQL tornou-se a mais viável para o problema descrito.

Uma vez que o banco de dados NoSQL DynamoDB [Sivasubramanian 2012], na nuvem da Amazon, foi escolhido, era necessário adequar e migrar os dados da tabela `posicao` para uma coleção do DynamoDB e adaptar as duas aplicações que compõem o sistema para que ambas passassem a trabalhar com o DynamoDB. Porém, essa solução deveria ser utilizada apenas para a entidade posição. Todas as outras entidades deveriam continuar na base de dados relacional por causa das restrições de integridade que não são oferecidas em bases de dados NoSQL, além das mesmas não darem suporte a relacionamentos entre coleções. Além disso, a adaptação das aplicações deveria ser realizada da forma menos intrusiva possível a fim de evitar a inserção de erros e facilitar o trabalho dos desenvolvedores que não conheciam bem a arquitetura do sistema.

O restante do artigo está dividido da seguinte forma: a Seção 2 discute os principais trabalhos relacionados; a Seção 3 apresenta a arquitetura das duas aplicações que compõem o sistema; a Seção 4 descreve a migração dos dados da tabela `posicao` para uma coleção no DynamoDB; a adaptação das duas aplicações é relatada na Seção 5; por fim, a Seção 6 traz as conclusões e sugestões de trabalhos futuros.

2. Trabalhos Relacionados

Em [Jamshidi et al. 2013], uma revisão sistemática identifica 23 pesquisas focadas em migrações de aplicações legadas para a nuvem. Os autores classificam as migrações em: substituição, parcial, pilha completa, e "cloudificação". De acordo com essa classificação, a migração que o relato de experiência deste trabalho aborda é caracterizada como uma migração por substituição. Nenhuma das pesquisas que fizeram parte da revisão sistemática em [Jamshidi et al. 2013] é caracterizada como tal.

Lições aprendidas durante a transição de uma grande base de dados relacional para um modelo híbrido, que combina um banco de dados relacional e um banco de dados NoSQL, são relatadas em [Schram and Anderson 2012]. O sistema abordado nesse trabalho é composto de serviços e foi adaptado de forma pouco intrusiva, pois a composição

[1]Durante o período de escrita deste artigo, o tamanho da tabela `posicao` era, aproximadamente, 110GB.

dos serviços era realizada por meio da injeção de dependência do *framework* Spring, injetando em tempo de execução implementações concretas em referências de tipos abstratos. A aplicação RastroBR realiza injeção de dependência por meio do *framework* Spring, porém sua adaptação foi realizada utilizando aspectos para que a modificação fosse mais granular e pontual.

Em [Vu and Asal 2012] é realizada uma análise das principais plataformas de nuvem do mercado. Três exemplos de migração de aplicações são apresentados. A diferença entre os exemplos apresentados em [Vu and Asal 2012] para o relato deste trabalho é que em [Vu and Asal 2012] todas as modificações foram realizadas de forma intrusiva, uma vez que não se tratava de uma aplicação comercial em produção.

O trabalho [Chauhan and Babar 2011] relata a migração por pilha completa [Jamshidi et al. 2013] da ferramenta Hackystat para a nuvem. Os serviços que compõem a ferramenta foram implantados separadamente em instâncias da nuvem Amazon EC2 que utilizam o recurso de elasticidade. Os autores concluem que sistemas que dividem a lógica do negócio entre aplicação e base de dados, por meio de gatilhos e outras funcionalidades, são mais difíceis de migrar porque a nuvem deve oferecer um banco de dados que possibilite que as mesmas estruturas possam ser executadas. Essa é uma das razões que fez com que a migração da base de dados das aplicações RBRDriver e RastroBR fosse parcial, pois a mesma contém gatilhos que fazem parte da lógica do negócio.

Os autores em [Vasconcelos et al. 2013] propõem uma abordagem não-intrusiva baseada em eventos para adaptar o código de uma aplicação legada para o ambiente de nuvem levando em conta as restrições dessa nova plataforma. Nessa abordagem, o código fonte da aplicação é interceptado usando, por exemplo, programação orientada a aspectos, e seu fluxo de execução é direcionado para um serviço similar na nuvem através da utilização de um *middleware* baseado em eventos. Como prova de conceito, os autores sugerem a utilização de aspectos para interceptar a persistência de arquivos em disco e armazená-los na nuvem sem modificar o código legado da aplicação. A adaptação relatada neste trabalhou foi baseada nessa idéia, porém os aspectos foram utilizando para desviar a persistência de dados de uma base de dados relacional para uma base de dados NoSQL na nuvem.

3. Arquitetura do Sistema

O RBRDriver é uma aplicação *standalone*, escrita em Groovy [König et al. 2007], cuja principal função é decodificar os pacotes enviados pelos veículos monitorados e persistir suas posições na base de dados, além de enviar comandos e configurações para os equipamentos instalados nos veículos. Cada posição decodificada é inserida como um registro na tabela `posicao` no SGBD relacional PostgreSQL.

O RBRDriver utiliza o padrão DAO, em combinação com o padrão *Abstract Factory*, para interagir com a base de dados. A classe `PositionCodec` é responsável por decodificar e persistir os dados recebidos. Essa classe possui uma referência do tipo `DaoFactory`, uma interface, e requisita a ela objetos DAO para consultar dados e persistir as posições dos veículos na tabela `posicao`.

O tipo concreto da fábrica de objetos DAO, a referência do tipo `DaoFactory`, é definido em um arquivo de configuração. Nesse arquivo é definida a classe `SqlFactory`

como fábrica de objetos DAO. Portanto, todo objeto DAO retornado por objetos `DaoFactory` são DAOs específicos para interagir com bancos de dados relacionais. Não é possível alterar essa configuração para que uma implementação de fábrica específica para uma base de dados NoSQL em particular seja utilizada. Dessa maneira, toda a família de objetos DAO seria alterada. Isso não é interessante porque, além de persistir as localizações dos veículos, a aplicação também executa outras funções, como envio de comandos e configurações, e os dados necessários para a execução dessas funções necessitam dos relacionamento e integridade oferecidos pela base de dados relacional.

O RastroBR é uma aplicação *web* que permite que os usuários acompanhem o deslocamento dos veículos, mantenham os cadastros de clientes, usuários e veículos, e obtenham relatórios de deslocamento e estatísticas. Assim como o RBRDriver, o RastroBR utiliza o padrão DAO em conjunto com o padrão *Abstract Factory* para implementar a camada de acesso ao banco de dados. No RastroBR as classes DAO utilizam o *framework* GORM [Fischer 2009], sendo esse, por sua vez, implementado sobre os *frameworks* Spring e Hibernate.

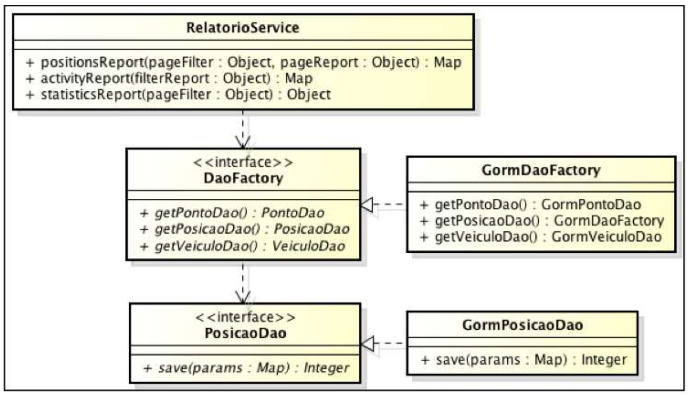

Figura 1. Interfaces e classes DAO, do RastroBR, relacionadas à consulta de posições.

Os relatórios oferecidos pelo RastroBR aplicam regras de negócio sobre registros da tabela `posicao`, tendo seus desempenhos afetados por ela. As classes de serviço dos relatórios possuem uma referência cujo tipo é a interface `DaoFactory`, por meio da qual obtêm as classes DAO para interagir com a base de dados. Através da capacidade de injeção de dependência do *framework* Spring, a classe concreta que implementa a interface `DaoFactory` pode ser facilmente trocada por meio de um script de configuração. Em tempo de execução, a referência do tipo `PosicaoDao` obtida é um objeto do tipo `GormPosicaoDao` que consulta as posições da tabela `posicao` para criar os relatórios requisitados (Figura 1).

4. Migração dos dados

O banco de dados escolhido, o DynamoDB, é um banco de dados NoSQL orientado a agregados [Sadalage and Fowler 2013]. O conceito de agregado é bastante apropriado para a arquitetura escalonada horizontalmente. Um agregado informa ao banco de dados quais dados devem ser tratados como uma única unidade indivisível, e, por isso, os mesmos devem ser mantidos em um mesmo servidor. No entanto, dois agregados diferentes, mesmo sendo de uma mesma coleção, não precisam ser mantidos em um mesmo servidor.

A ausência de relacionamentos entre agregados, como aqueles mantidos por meio de chaves estrangeiras em bancos de dados relacionais, propicia a divisão dos dados entre vários nós. Porém, por causa dessa característica, não é possível, em uma única consulta, realizar junções entre agregados de coleções diferentes. A persistência dos dados do sistema dependia do relacionamento entre entidades oferecido pelo modelo relacional.

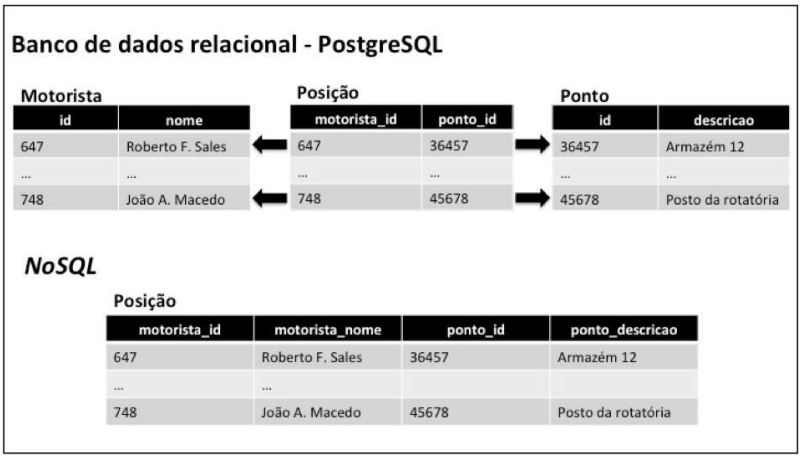

Figura 2. Adaptação dos dados da tabela `posicao` para uma coleção NoSQL.

Apesar de sua importância, a tabela `posicao` armazena apenas dados históricos que não são editados. Essa particularidade permitiu adaptar seus dados para uma coleção NoSQL. A adequação dos dados foi realizada unindo em um único agregado cada registro da tabela `posicao` com os registros de outras tabelas referenciados pelas chaves estrangeiras dela[2]. Assim, os agregados da coleção resultante fornecem, em apenas uma consulta, todos os dados necessários para a emissão dos relatórios do sistema. A Figura 2 mostra, de forma simplificada[3], a tabela `posicao` referenciando outras tabelas, e como seus dados foram adaptados para uma coleção trazendo os dados dos registros referenciados para junto dos registros da própria tabela, criando um agregado.

5. Adaptação do Sistema

Para utilizar técnicas de programação orientada a aspectos na adaptação da aplicação RastroBR, foi utilizado o *framework* Spring AOP (*Aspect Oriented Programming*). No Spring AOP, os aspectos são implementados utilizando classes Java comuns. É possível indicar ao *framework* quais classes implementam aspectos utilizando XML em um arquivo de configuração, e nas classes, especificar os *joint points* e *advices* por meio de *annotations*.

A estratégia adotada foi criar um aspecto contendo um *advice* do tipo *around*, que intercepta o método `getPosicaoDao` da interface `DaoFactory` e retorna uma implementação DAO apropriada para o DynamoDB, ao invés de uma implementação para um banco de dados relacional, como em todo o resto da aplicação. No RastroBR, uma implementação da interface `DaoFactory` é injetada nos serviços utilizando a injeção de dependência do Spring e, quando esses requisitam um DAO, obtêm DAOs de uma mesma família, no caso, implementações para um banco de dados relacional.

[2]O escopo deste trabalho não aborda as consequências referentes à perda de normalização dos dados.

[3]Os demais campos das tabelas não são exibidos por serem dispensáveis para o entendimento da solução.

Para modificar esse comportamento foi implementado um *advice* apenas para as classes de serviços relacionadas aos relatórios que dependem da tabela `posicao`. Esse *advice* intercepta o *pointcut* correspondente à chamada de método que retorna um DAO da entidade `posicao`, e, ao invés de retornar uma classe que faça parte da família de classes definida pela injeção de dependência, retorna uma implementação própria para interagir com a coleção `posicao` no banco DynamoDB, mas que também implementa a interface `PosicaoDao`. Um trecho de código do aspecto que contém esse *advice* é exibido na Figura 3. A linha 13 exibe o `pointcut` que intercepta a chamado ao método `getPosicaoDao` da interface `DaoFactory`, e a linha 15 retorna uma implementação DAO para o DynamoDB.

```
10 @Aspect
11 class DaoFactoryAspect {
12
13     @Around("execution(* rastro.br.dao.DaoFactory.getPosicaoDao())")
14     Object invoke(ProceedingJoinPoint joinPoint) throws Throwable {
15         new DynamoDbPosicaoDao(DynamoDbClient.instance)
16     }
```

Figura 3. *Advice* retorna um DAO DynamoDB ao invés de uma DAO relacional.

Com a implementação do aspecto contendo o *advice* exibido na Figura 3, um objeto da classe `RelatorioService` chama o método `getPosicaoDao` da interface `DaoFactory` para consultar a a tabela `posicao`. Nesse momento, um *advice* da classe `DaoFactoryAspect` intercepta essa chamada e, ao invés do fluxo normal acontecer, que seria a classe *Factory* (que implementa a interface `DaoFactory`) retornar uma instância de acordo com sua família, por exemplo um `GormPosicaoDao`, o aspecto retorna para a classe cliente uma instância do tipo `DynamoDbPosicaoDao`. A classe `DynamoDbPosicaoDao` também implementa a interface `PosicaoDao` e é com essa interface que a classe cliente, `RelatorioService`, interage. Portanto, não houve nenhuma incompatibilidade e o código da classe `RelatorioService` não necessitou de alterações.

Para que o aspecto seja ativado, passando a interceptar os *pointcuts* que foram especificados em seus *advices*, é necessário indicar para Spring AOP quais classes são aspectos. Isso é feito através da definição do *bean* no script de configuração dos Spring *beans* da aplicação.

Semelhantemente ao RastroBR, no RBRDriver foi implementado um aspecto, utilizando AspectJ [Kiselev 2002], contendo um *advice* do tipo *around*, para capturar as chamadas ao método `getPosicaoDao` da interface `DaoFactory`. No entanto, o *advice* do aspecto estava sendo executado diversas vezes ao invés de ser executado apenas uma vez para cada chamada ao método `getPosicaoDao` da interface `DaoFactory`. As funcionalidades dinâmicas da linguagem Groovy não permitiam o funcionamento correto do aspecto.

O artigo [McClean 2006][4] apontou uma solução. Em uma aplicação Groovy, para cada classe carregada na memória existe uma metaclasse correspondente que possui todas

[4]MCCLEAN, J. Painless AOP with Groovy. InfoQueue, 2 October 2006. Disponivel em: ¡http://www.infoq.com/articles/aop-with-groovy¿. Acesso em: 27 Janeiro 2014..

as propriedades e métodos da classe à qual ela está relacionada. De acordo com o protocolo que rege as chamadas de método em aplicações Groovy, uma chamada de método para um objeto é direcionada para uma instância da metaclasse deste objeto, ao invés do objeto propriamente dito. Groovy permite trocar a metaclasse de uma classe em tempo de execução, alterando dinamicamente o comportamento de todas as instâncias dessa classe. Dessa maneira, é possível alterar dinamicamente o comportamento de uma classe sem modificar seu código fonte. Uma das maneiras de fazer isso é pôr a metaclasse substituta, nomeada de forma adequada, em um pacote específico, cujo nome e posição na hierarquia de pacotes depende da classe cuja metaclasse será substituída.

Como qualquer classe Groovy, a classe `SqlPosicaoDao`, responsável por salvar as posições dos veículos na tabela `posicao`, possui uma metaclasse com cópias de todos os seus métodos e propriedades. Para cada método chamado na classe `SqlPosicaoDao`, na realidade, o método equivalente é chamado na metaclasse. Ou seja, quando o método `save` é chamado na classe `SqlPosicaoDao`, o que realmente acontece é uma chamada ao método `invokeMethod` na metaclasse de `SqlPosicaoDao`, que por sua vez chama o método `save` da metaclasse. Portanto, era necessário substituir a metaclasse padrão fornecida pelo compilador por uma metaclasse com uma implementação adequada para o DynamoDB.

Isso foi realizado através do método citado anteriormente, a substituição da metaclasse padrão da classe `SqlPosicaoDao` por meio da adição de uma nova metaclasse, no pacote apropriado, no *classpath* da aplicação. A nova metaclasse foi nomeada `SqlPosicaoDaoMetaClass` e foi colocada no pacote `groovy.runtime.metaclass.com.azultecnologia.dao.sql` seguindo o padrão necessário para que a substituição da metaclasse padrão ocorra. Na metaclasse substituta foi implementado um código que utilizasse o DAO `DynamoDbPosicaoDao` para armazenar a posição no banco de dados DynamoDB e retornar o mesmo valor que seria retornado pelo método `save` da classe `SqlPosicaoDao`.

Depois da alteração, ao chamar o método `save` da classe `SqlPosicaoDao`, o método `invokeMethod` da nova metaclasse é executado e a classe `DynamoDbPosicaoDao` é utilizada para armazenar a posição. Por fim, o número de posições armazenadas é retornado. Assim, um efeito semelhante a um *advice* do tipo *around* é obtido.

6. Conclusões

A utilização de aspectos para adaptar a aplicação RastroBR, como sugerido em [Vasconcelos et al. 2013], para modificar aplicações de forma não intrusiva, se mostrou valiosa. A AOP foi utilizada não para tratar um interesse que permeia toda a aplicação, mas apenas a persistência de um domínio em uma base de dados alternativa. Essa abordagem pode ser utilizada para tratar diversos outros casos onde seja necessário adaptar código de aplicações de forma não intrusiva.

É difícil adaptar uma aplicação para que toda a sua base de dados relacional seja migrada para um banco de dados NoSQL. De acordo com [Sadalage and Fowler 2013], quando uma aplicação utiliza um banco de dados NoSQL, seu domínio deve ser projetado desde o início tendo em mente as características desse tipo de banco de dados. Bases de dados NoSQL não possuem muitos dos recursos de relacionamento entre entidades e inte-

gridade que os arquitetos já possuem em mente no momento em que projetam o domínio de uma aplicação. Porém, como visto neste relato de experiência, em alguns casos, é possível migrar parte dos dados quando estes apresentam características que permitam uma adaptação.

As metaclasses Groovy foram utilizadas de forma muito semelhante a aspectos, a fim de alcançar o objetivo de modificar a aplicação de forma não-intrusiva. As funcionalidades dinâmicas da linguagem Groovy podem ser utilizadas como aspectos em aplicações Groovy, evitando a adição de mais um *framework* na aplicação.

Pretendemos, como trabalho futuro, aplicar as técnicas utilizadas na migração relatada neste artigo com outras soluções de bancos de dados NoSQL e realizar um comparativo relacionado a desempenho, facilidade de implementação, gerenciamento oferecido pela nuvem, entre outros. Outro trabalho interessante é utilizar apenas as funcionalidades da linguagem Groovy para capturar os pontos de interesse das aplicações e comparar essa abordagem com a que utiliza programação orientada a aspectos.

Referências

Chauhan, M. A. and Babar, M. A. (2011). Migrating service-oriented system to cloud computing: An experience report. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 404–411. IEEE.

Fischer, R. (2009). *Grails Persistence with GORM and GSQL*. Apress, 1st edition.

Jamshidi, P., Ahmad, A., and Pahl, C. (2013). Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, 1(2):142–157.

Kiselev, I. (2002). *Aspect-Oriented Programming with AspectJ*. Sams, 1st edition.

König, D., Laforge, G., King, P., Champeau, C., D'Arcy, H., Pragt, E., and Skeet, J. (2007). *Groovy In Action*. Manning Publications Co., 1st edition.

Sadalage, P. J. and Fowler, M. (2013). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 1st edition.

Schram, A. and Anderson, K. M. (2012). Mysql to nosql: data modeling challenges in supporting scalability. In *SPLASH 12 Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 191–202. ACM.

Sivasubramanian, S. (2012). Amazon dynamodb: a seamlessly scalable non-relational database service. In *SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730. ACM.

Vasconcelos, M. A., Barbosa, D. M., Maia, P. H. M., and Mendonça, N. C. (2013). Uma abordagem baseada em eventos para adaptação automática de aplicações para a nuvem. In *VEM 2013: I Workshop Brasileiro de Visualização, Evolução e Manutenção de Software*.

Vu, Q. H. and Asal, R. (2012). Legacy application migration to the cloud: Practicability and methodology. In *2012 IEEE Eighth World Congress on Services*, pages 270–277. IEEE.