PROVDEPLOY: APOIO À COLETA DE DADOS DE PROVENIÊNCIA EM SCRIPTS DE EXECUÇÃO DE CÓDIGOS CIENTÍFICOS.

Liliane Neves de Oliveira Kunstmann

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Marta Lima de Queirós Mattoso
Daniel Cardoso Moraes de Oliveira

Rio de Janeiro
Abril de 2020

PROVDEPLOY: APOIO À COLETA DE DADOS DE PROVENIÊNCIA EM SCRIPTS DE EXECUÇÃO DE CÓDIGOS CIENTÍFICOS.

Liliane Neves de Oliveira Kunstmann

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Marta Lima de Queirós Mattoso
Daniel Cardoso Moraes de Oliveira

Aprovada por: Profa. Marta Lima de Queirós Mattoso
Prof. Daniel Cardoso Moraes de Oliveira
Prof. Alexandre de Assis Bento Lima
Profa. Carla Osthoff Ferreira de Barros

RIO DE JANEIRO, RJ – BRASIL
ABRIL DE 2020

Kunstmann, Liliane Neves de Oliveira

ProvDeploy: apoio à coleta de dados de proveniência em scripts de execução de códigos científicos./Liliane Neves de Oliveira Kunstmann. – Rio de Janeiro: UFRJ/COPPE, 2020.

IX, 56 p.: il.; 29, 7cm.

Orientadores: Marta Lima de Queirós Mattoso
Daniel Cardoso Moraes de Oliveira

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2020.

Referências Bibliográficas: p. 47 – 56.

1. Proveniência. 2. Virtualização. 3. Aplicações científicas. I. Mattoso, Marta Lima de Queirós *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais, Eliane e Aelson.

Agradecimentos

A Deus, Criador de todas as coisas, por sua infinita Graça e Misericórdia.

Ao meu marido, Ronaldo, por todo carinho, compreensão e suporte nos momentos difíceis.

Aos meus pais, Eliane e Aelson, que sempre foram grandes incentivadores de todos os meus projetos.

Aos meus irmãos, Lívia e Marcus Vinícius, pelo apoio.

À minha orientadora, professora Marta Mattoso, por sempre exigir excelência, por dedicar seu tempo a meu desenvolvimento e por todo o esmero que teve com meu trabalho.

Ao meu coorientador, professor Daniel de Oliveira, por todo o apoio e dedicação, por sua ajuda inestimável em problemas técnicos e por seus apontamentos sempre visando a melhoria de meu trabalho.

Aos professores do PESC/COPPE, em especial, ao professor Geraldo Xexéo, pelos direcionamentos no início do Mestrado.

Aos meus professores do curso de Ciência da Computação da UFRRJ por me incentivarem e proporcionarem as ferramentas necessárias o meu desenvolvimento pessoal e profissional.

À minha querida amiga Jamile com quem sempre pude contar e confiar.

Aos meus amigos, que sempre me apoiaram oferecendo ajuda e suporte emocional. Especialmente agradeço: Ricardo Luiz, Egberto e Michel por tornarem meu início nesta jornada mais leve e alegre. Por todos nutro grande admiração

Agradeço também a todo o pessoal do NACAD e da secretaria do PESC pelo suporte em questões administrativas. Em especial: Ana Paula Rabelo, Patrícia Leal, Ricardo, Mara Prata e Gutierrez da Costa.

Aos professores Alexandre Assis e Carla Ostoff por aceitarem participar da minha banca oferecendo valiosas contribuições.

Aos colegas com quem trabalhei durante o mestrado - Luiz Gustavo, Vitor Silva, Renan Souza, Elaine Tady, Leonardo Azeredo e Eduardo Mangeli. À colega Débora Pina pela força e incentivo diários.

A todos, minha eterna gratidão!

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROVDEPLOY: APOIO À COLETA DE DADOS DE PROVENIÊNCIA EM SCRIPTS DE EXECUÇÃO DE CÓDIGOS CIENTÍFICOS.

Liliane Neves de Oliveira Kunstmann

Abril/2020

Orientadores: Marta Lima de Queirós Mattoso
Daniel Cardoso Moraes de Oliveira

Programa: Engenharia de Sistemas e Computação

Aplicações científicas em larga escala se caracterizam por invocar diversas bibliotecas de *software* e produzir grandes quantidades de dados por meio de *scripts*. Comumente esses *scripts* envolvem processamento de alto desempenho (PAD) na execução de seus códigos científicos. A complexidade de preparação desses *scripts* para execução é alta, pois utilizam muitos componentes externos, que geram uma grande pilha de *software*. Indo ao encontro das dificuldades de configuração e execução, a virtualização baseada em contêineres vem facilitando o empacotamento de pilhas de *software* em *scripts*. No entanto, o uso de contêineres em ambientes PAD apresenta desafios com a segurança do ambiente e a sobrecarga na execução da aplicação. Um outro complicador que aumenta a pilha de *software* na execução de *scripts* são os serviços de coleta de dados de proveniência. Dados de proveniência proporcionam poder analítico e de monitoramento. Para facilitar a adoção de serviço de proveniência em ambientes PAD, esta dissertação apresenta a *ProvDeploy*. O objetivo da *ProvDeploy* é direcionar a composição da virtualização da aplicação, para execução em PAD com contêineres, incorporando serviços de coleta de dados de proveniência de modo sistemático e com poucos passos de configuração. Os experimentos realizados com a *ProvDeploy* para execução de diversos *scripts* de códigos científicos evidenciaram a redução do esforço necessário. A adoção de serviços de coleta de proveniência em ambientes PAD foi facilitada e não houve sobrecarga no desempenho das aplicações executadas em contêineres com a *ProvDeploy*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROVDEPLOY: SUPPORT FOR PROVENANCE DATA CAPTURE ON EXECUTION OF SCIENTIFIC SCRIPTS

Liliane Neves de Oliveira Kunstmann

April/2020

Advisors: Marta Lima de Queirós Mattoso
Daniel Cardoso Moraes de Oliveira

Department: Systems Engineering and Computer Science

Large-scale scientific applications are characterized by using many software libraries and producing large amounts of data through scripts. Usually, those scripts require High Performance Computing (HPC) on scientific code execution. The complexity to prepare those scripts for execution is high due to the amount of third-party software, that compose the software stack. Container-based virtualization helps on configuration and execution of script applications. However, the use of containers in HPC environments faces challenges of environment security and overhead on the application execution. Another complicating component that increases the software stack size are the provenance data capture services. The capture of provenance data provides analytical and monitoring capabilities. To ease the adoption of provenance data capture services in scripts using HPC environments, this dissertation presents *ProvDeploy*. The main goal of *ProvDeploy* is to guide the composition of the script's virtualization, for execution in HPC with containers, integrating the provenance data capture services in a systematic way and with a few configuration steps. The experiments performed with *ProvDeploy* over diverse scripts of scientific code showed the reduction of the necessary effort. The adoption of provenance data capture services in HPC environments has been facilitated and there was no overhead on the performance of applications executed in containers with *ProvDeploy* .

Sumário

Lista de Figuras **ix**

1 Introdução **1**

2 Virtualização em contextos científicos **8**

2.1 Máquinas virtuais 9

2.2 Virtualização baseada em contêineres 10

2.3 Contêineres em ambientes de PAD 16

2.4 Coleta de dados de proveniência em ambientes virtualizados 18

2.5 Trabalhos relacionados 20

3 Abordagem proposta: ***ProvDeploy*** **24**

3.1 Linha de experimento da *ProvDeploy* 25

3.2 Definição do modelo de custo 26

3.3 A arquitetura da *ProvDeploy* 28

4 Avaliação experimental **34**

4.1 Estudo de caso: WordCount 35

4.2 Estudo de caso: AlexNet 39

4.3 Estudo de caso: Modelo substitutivo para quantificação de incertezas - DenseED 42

5 Conclusões **45**

Referências Bibliográficas **47**

Lista de Figuras

2.1 Diferentes arquiteturas de *hypervisor*. 10

3.1 Linha de experimento da *ProvDeploy* 26
3.2 Componentes da *ProvDeploy* 28
3.3 Visualização do catálogo da *ProvDeploy* 29
3.4 Funcionamento da *ProvDeploy* 31
3.5 Adição de uma nova imagem 32
3.6 Inicialização de execução 33

4.1 Abstração do dados coletados no *WordCount* 36
4.2 Abstração do dados coletados na *Alexnet* 40
4.3 Abstração do dados coletados na *DenseED* 43

Capítulo 1

Introdução

Nas últimas décadas, tem-se observado o crescimento das Ciências Computacionais e Engenharias (*Computational Science and Engineering* - CSE) para além de suas origens (Matemática e Física) [1]. Essa expansão tem revolucionado os mais diversos campos de estudo, de Medicina à Economia. O impacto da CSE na vida humana é difícil de ser avaliado, uma vez que modelagem e simulação têm transformado processos, projetos e o funcionamento de sistemas, e tem se tornado presente em diversas as áreas. As aplicações em CSE são baseadas em modelos matemáticos e simulações computacionais baseadas em métodos numéricos, sendo responsáveis por aeronaves mais eficientes, transistores de maior densidade, processos químicos mais eficazes, elevação na qualidade de equipamentos médicos de imageamento, entre outros [1].

As aplicações científicas se utilizam de muitos componentes [2] e *softwares* de terceiros, seja para visualização, conversão de formatos, compressão de dados, monitoramento da execução ou mesmo de depuração e análise de desempenho. Estes componentes são usualmente invocados por meio de *scripts*. A configuração, instalação e implementação desses *scripts* é complexa devido à pilha de softwares a serem invocados para cada um de seus componentes.

Um exemplo deste tipo de *script* é o *libmesh-sedimentation* [3], que modela uma aplicação em CSE que simula correntes turbidíticas tipicamente encontradas em processos geológicos, que podem dar a indicação de formação de reservatórios de petróleo. O *script libmesh-sedimentation* implementa as simulações por meio de invocações de diferentes bibliotecas, dentre elas: a *libmesh* [4], um solucionador de equações que adota o método de elementos finitos e o *Paraview Catalyst* [5] para visualização de dados in-situ.

Os problemas que as aplicações em CSE se propõem a resolver, geralmente, requerem processamento de alto desempenho (PAD), e, mesmo em ambientes PAD, estas aplicações podem demorar horas ou mesmo dias até que sua execução seja finalizada [1]. É importante observar que tanto a modelagem dessas soluções como a

escolha dos parâmetros são tarefas complexas. Assim, após a execução, a aplicação pode produzir resultados não satisfatórios, o que, em geral, acarreta novas execuções após ajustes nas configurações da aplicação. Esse processo é bem custoso do ponto de vista de tempo de execução e esforço de ajustes de parâmetros e configurações. A descoberta de quais dados precisam ser analisados e a decisão de quais ajustes finos realizar em parâmetros da simulação é difícil, uma vez que os dados de configuração, parâmetros e resultados intermediários estão frequentemente espalhados em diferentes arquivos e desconectados do passo de tempo correspondente à sua geração [6, 7].

A análise de dados científicos para a sintonia fina de parâmetros e configuração da aplicação CSE pode ser enriquecida por meio da coleta de dados de proveniência. Sendo assim, além de invocar bibliotecas com *softwares* de terceiros para métodos numéricos, visualização, perfilagem de código e monitoramento, torna-se necessário invocar sistemas de coleta de dados de proveniência e de análise de dados.

BUNEMAN *et al.* [8] definem proveniência de dados como a "descrição da origem de um elemento de dados e o processo através do qual este chegou ao banco de dados". Para DAVIDSON e FREIRE [9], a proveniência não se resume à origem de elementos de dados, mas se refere a registros da derivação de um conjunto de resultados, incluindo os processos que levaram aos resultados. Ainda nessa forma mais abrangente da proveniência, segundo SIMMHAN *et al.* [10] a coleta de dados de proveniência pode ser utilizada para aprimorar qualidade e veracidade de dados, caraterísticas importantes dada a crescente complexidade na geração de conjuntos de dados. A coleta de dados de proveniência favorece também a reprodução e o compartilhamento de resultados, facilitando o trabalho colaborativo, a validação e confiabilidade de resultados. Existem diversos trabalhos que coletam dados de proveniência para apoiar as mais variadas necessidades inerentes a uma aplicação, *e.g.,* escalonamento de tarefas [11], tolerância a falhas [12], dimensionamento de recursos [13, 14] e privacidade de dados [15]. Aplicações em CSE também podem se beneficiar da coleta de dados de proveniência para depurar, monitorar execuções, analisar seu desempenho, escalonar tarefas, detectar anomalias, auxiliar a tomada de decisão, realizar adaptações, *etc.*, provendo assim mais controle e auxilio ao usuário. Por todas essas vantagens, as aplicações em CSE vêm demandando a coleta de dados de proveniência, pois esta pode adicionar valor significativo às análises das grandes quantidades de dados gerados e consumidos pelas simulações.

Como aplicações em CSE demandam PAD e têm execuções que podem se estender por dias ou semanas [16], é importante que a sobrecarga adicionada pela ferramenta de coleta de dados de proveniência seja baixa. Uma ferramenta que atende com sucesso esse requisito é a DfAnalyzer [3]. A DfAnalyzer é uma biblioteca de software que pode ser invocada por *scripts* em CSE sobre ambientes PAD

para o monitoramento, a depuração, a condução e a análise de fluxos de dados. DfAnalyzer funciona a partir da coleta de dados estratégicos do domínio CSE e o registro de dados de proveniência, possibilitando consultas em tempo de execução. DfAnalyzer se destaca, dentre outras soluções de coleta de dados de proveniência, por permitir trabalhar junto a aplicações de CSE em PAD sem interferir no desempenho da aplicação ou exigir o uso de um portal ou ambiente específico de execução, conforme evidenciado por SILVA *et al.* [3], CAMATA *et al.* [17], PINA *et al.* [18].

Apesar dos múltiplos benefícios da coleta de dados de proveniência com ferramentas como a DfAnalyzer [3, 17], o uso de abordagens desse tipo, em geral, implica a adição de bibliotecas de coleta dados de proveniência à pilha de *software* do *script* do usuário. Estes componentes da biblioteca de coleta de dados podem ser de apoio a máquinas virtuais Java, bibliotecas de compressão de dados, *etc.* Tomemos como exemplo a DfAnalyzer, que necessita de um Sistema de Gerência de Banco de Dados (SGBD) para armazenar dados e prover consultas. Tal SGBD é responsável por armazenar os dados de proveniência e permitir ao usuário elaborar consultas (em SQL, por exemplo) para realizar suas análises durante a execução da simulação. O uso de um SGBD não é uma característica somente da DfAnalyzer. Outras soluções também fazem uso de SGBDs para facilitar a análise de dados, como o Komadu [19] e o ProvenanceCurious [20]. Em muitos casos, o processo de adição de chamadas a bibliotecas de *software* à pilha de componentes *software* da aplicação do usuário poder ser um processo longo, laborioso e propenso a erros, se executado manualmente. Muitas vezes, por conta dessa dificuldade e do tempo necessário para a configuração, o usuário desiste de utilizar uma solução de coleta de dados de proveniência, perdendo o poder analítico provido por tais soluções.

Os ambientes de PAD possuem uma arquitetura sofisticada com objetivo de atender a múltiplos usuários, ofertando alto desempenho e vazão à execução de aplicações, ao mesmo tempo que precisam garantir segurança e estabilidade de todo o ambiente [2]. Com isto, a gerência de aplicações não é simples e não se dá de maneira similar à gerência em computadores pessoais em que o usuário possui todos os privilégios. As aplicações científicas crescem em complexidade e em tecnologias (*e.g.,* bibliotecas e componentes que visam a otimizar seu desempenho), o que gera a demanda de ambientes cada vez mais customizáveis [2, 21].

Na tentativa de configurar o ambiente necessário para executar suas aplicações em CSE, um usuário pode enfrentar diversos problemas como conflitos com os *softwares* ou versões presentes no sistema nativo, ou em áreas de outros usuários [2]. As restrições vigentes podem fazer com que muitos dos *softwares* necessários precisem que suas instalações sejam feitas a partir da compilação de códigos fonte, sem que nem sempre haja acesso aos códigos e que, além de trabalhoso, pode exigir do usuário conhecimentos para além de seu escopo. Outro complicador ocorre quando

passos implícitos na instalação de componentes de *software* podem comprometer o funcionamento da aplicação ou gerar comportamento indesejado ou desconhecido. Componentes legados, depreciados [1] ou descontinuados, podem exigir versões de outras bibliotecas, o que pode gerar ainda mais conflitos. Além disso, *softwares* descontinuados não oferecem apoio ao usuário ou ao sistema operacional, e muitas vezes não são facilmente substituíveis.

Este cenário de instalação e configuração de bibliotecas de *softwares* aumenta em complexidade com a coleta de dados de proveniência em ambientes de PAD. O usuário que deseja os benefícios que a captura de dados de proveniência provê, terá de adicionar ao tempo de configuração, o tempo para configurar a solução de proveniência que deseja utilizar. A coleta de dados de proveniência pode trazer conflitos de configuração ao ambiente desejado pelo usuário, por que uma vez que não há isolamento de ambientes, uma ferramenta em uma versão necessária ao serviço de captura de proveniência, não necessariamente é igual ao necessário à aplicação de CSE.

Para ilustrar os problemas de configuração de um *script* e sua pilha de *softwares* em aplicações CSE, seja o *workflow* denominado rtmUq [22], que executa uma simulação de migração reversa no tempo (RTM), um dos métodos para imageamento sísmico que consome modelos bayesianos de velocidade e produz seções sísmicas migradas empilhadas. Esse *workflow* faz uso de coleta de dados de proveniência com a dfa-lib-cpp, biblioteca de *software* de apoio às chamadas da DfAnalyzer. A dfa-lib precisa da versão do GCC 7.3.*, ao mesmo tempo que o rtmUq precisa utilizar Intel MPI com GCC 4.8.*. Outro exemplo pode ser observado na configuração de uma aplicação que use o Komadu para coletar dados de proveniência. Na fase de instalação do Komadu é necessário compilar os módulos *Model* e *XML* do ProvToolBox-0.4.0 [23], que precisam da biblioteca Java para criar e converter representações ao modelo de dados do padrão W3C PROV [24]. Considere-se a plataforma Java adotada na versão 13, no entanto, na versão 11 foram removidos os módulos JAXB, JAX-WS, JTA, JAF, CORBA, entre outros e os módulos e ferramentas a estes relacionados. Os módulos JAXB e JAX-WS são necessários na compilação de todos os módulos do ProvToolBox-0.4.0 utilizados pelo Komadu. Desta forma, se alguma outra aplicação do usuário precisar de uma versão do Java igual ou posterior à 11, isso passa a ser um problema. Uma maneira de resolver a falta dos módulos em questão é modificar manualmente os arquivos pom.xml de todos módulos do ProvToolBox-0.4.0 de modo que o mesmo adicione ao Maven os módulos ausentes durante a compilação, mas esse conhecimento não necessariamente é de domínio do usuário. Outra solução possível é instalar mais de uma JVM no

[1]Componentes depreciados são aqueles que, apesar de funcionarem, não são recomendados porque há outro que exerce a mesma função de forma mais coesa ou correta.

ambiente, e fazer com que cada aplicação "aponte" para a JVM com a versão necessária do Java. Entretanto, essa solução faz com que o ambiente possua múltiplas versões da JVM, o que piora (e muito) a manutenção do mesmo. Esses pequenos exemplos já dão a dimensão da complexidade em se configurar e compor uma pilha de *software* em CSE.

Com vistas a facilitar o desenvolvimento e implantação de sistemas, a tecnologia de contêineres começou a ser amplamente utilizada [25]. A conteinerização pode ser descrita como uma virtualização mais leve, de rápida inicialização, que mitiga o problema de configuração por meio da criação de ambientes de execução isolados e portáveis entre diferentes máquinas, possibilitando a execução de diversas aplicações sobre o mesmo sistema operacional [26]. Contêineres são instâncias de imagens e imagens podem ser definidas como arquivos somente de leitura (*read only*), que contêm as informações (bibliotecas, arquivos de configuração, *scripts*, *etc*) necessárias à execução de um determinado ambiente. Uma imagem é gerada com vistas a reproduzir um ambiente. Dentro de uma imagem de contêiner fica encapsulado todo um ambiente de maneira isolada. Algumas tecnologias de contêineres possuem registros públicos como o *Docker Hub*[2], *Singularity Hub*[3] e o NGC[4], onde é possível encontrar imagens que contém determinados *softwares* para serem baixados e utilizados, arquivos de configuração de contêineres e imagens otimizadas. Entretanto, uma vez que o número de componentes necessários à aplicação, pode ser muito grande, talvez não haja imagem que atenda tais requisitos, e por fim, o usuário pode terminar por ter que escrever uma imagem personalizada [27] o que muitas vezes pode não ser simples.

Contêineres comerciais possibilitaram o desenvolvimento da arquitetura de microsserviços [28]. Um microsserviço pode ser definido como uma funcionalidade de um sistema que pode ser desenvolvida, disponibilizada, testada e escalada de maneira independente. Os contêineres utilizados em aplicações comerciais, são de simples reutilização, e, para criação de imagens personalizadas, há pouco esforço envolvido. No contexto de contêineres para aplicações científicas este reuso não é simples e muitas tarefas de configuração precisam ser feitas pelo usuário quando há necessidade de personalização [27]. Além disso, a conteinerização não se aplica tão facilmente em contextos de alto desempenho, por motivos de complexidade do ambiente de execução, de segurança e desempenho [29].

A portabilidade oferecida pela conteinerização se mostra cada vez mais necessária aos ambientes de PAD. Apesar de existirem soluções de conteinerização voltadas para PAD [21, 30, 31] e para facilitar o seu uso [2], os contêineres ainda são pouco utili-

[2]https://hub.docker.com/

[3]https://singularity-hub.org/

[4]https://ngc.nvidia.com/

zados pelo usuário de PAD [27]. Uma das principais razões para a dificuldade nessa adoção é que mesmo com contêineres, ainda é complexo realizar a configuração das múltiplas bibliotecas necessárias ao ambiente PAD e às aplicações, assim, o tempo necessário para executar tal tarefa é significativo [27]. Dessa forma, é desejável que exista uma abordagem que evite que a adição de uma biblioteca de coleta de dados de proveniência à pilha de software da aplicação em CSE torne a fase de configuração e preparação de ambientes ainda mais complexa e custosa com relação ao tempo do usuário.

A maneira como a geração de imagens de soluções de coleta de dados de proveniência é criada, é determinante na sua incorporação à aplicação em CSE. A imagem pode definir a flexibilidade que o usuário terá, permitindo ou não a troca de componentes e, no caso de coleta de dados de proveniência, facilita ou não o acesso aos dados e funcionalidades do coletor de proveniência. Para melhorar a configuração e instalação do coletor dentro de um contêiner, o usuário precisa entender seu funcionamento, seus componentes, as operações de escrita, as possíveis requisições, os arquivos de configuração, o uso de SGBDs (podendo utilizar mais de um), as possíveis bibliotecas de otimização para o coletor, *etc.* Se o usuário conteinerizar toda a aplicação de coleta de proveniência no mesmo contêiner de sua aplicação, ele pode perder a flexibilidade de modificar componentes como os SGBDs, versões de compiláveis e outros arquivos de configuração. Como o tempo para criação de imagens de contêineres já é grande, alterar algum componente do contêiner significa ter que criar uma nova imagem. Ainda, os conflitos de *software* que seriam resolvidos por meio do uso de contêineres podem voltar a acontecer dentro do contêiner. E finalmente, o usuário pode perder o interesse em acoplar a coleta de proveniência à sua aplicação, abrindo mão de benefícios como análise de dados, monitoramento, qualidade, confiança, *etc.*

Observado isto, o problema de pesquisa desta dissertação trata de auxiliar a adoção de ferramentas coleta de dados de proveniência em *scripts* em CSE que dependam de ambientes de PAD. Esta dissertação propõe realizar esta tarefa por meio do uso facilitado de contêineres para a geração de imagens envolvendo múltiplas bibliotecas de *software* e diferentes acessos a dados. Desenvolvida para esta dissertação, a *ProvDeploy* pode ser descrita como um serviço de apoio à adoção de coleta de dados de proveniência em *scripts* em CSE que executam em ambientes de PAD preferencialmente. Na arquitetura proposta, a *ProvDeploy* busca auxiliar a execução dos *scripts* do usuário, ao mesmo tempo que automaticamente provê um serviço de coleta dados de proveniência diminuindo a quantidade de tempo gasto pelo usuário em tarefas de configuração, diminuindo as barreiras de implantação de soluções de coleta de dados em *scripts*. A adoção da *ProvDeploy* permite ao usuário desfrutar dos benefícios da coleta de dados, sem perder tempo ou desempenho de

sua aplicação, ao mesmo tempo em que não aumenta significativamente a sobrecarga adicionada pela ferramenta. Utilizando a *ProvDeploy*, o usuário diminui significativamente o total de passos que precisa executar para chegar ao ambiente desejado para execução de seus *scripts* sem ter que adicionar tempo para configuração de um serviço de coleta de dados de proveniência. Apesar dos contêineres adicionarem uma sobrecarga ao *script* de execução da aplicação, nos experimentos realizados com a *ProvDeploy* foi observada uma redução no tempo de execução das aplicações. Isso ocorreu, devido à economia de tempo com a preparação de imagens e da geração do esquema do banco de dados. Como *scritps* em aplicações científicas são executados repetidamente, o tempo total do experimento diminuiu com a *ProvDeploy*.

Além desta introdução, esta dissertação está organizada em outros quatro capítulos. O Capítulo 2 apresenta os conceitos relacionados a esta dissertação como abordagens de virtualização, suas vantagens e desvantagens de uso em ambientes PAD, e, são apresentados alguns trabalhos que visam a resolver problemas de virtualização e/ou integração de ferramentas de coleta de dados de proveniência em PAD. O Capítulo 3 descreve a abordagem proposta por esta dissertação que é apoiar a adoção de ferramentas de coleta de dados de proveniência em ambientes PAD através da *ProvDeploy* que busca através da linha de experimento algébrica propor uma metodologia para composição e execução de contêineres em PAD e um modelo de custo que objetiva avaliar vantagens e desvantagens de uma determinada configuração de contêiner. O Capítulo 4 apresenta através de estudos de casos com três aplicações, como a *ProvDeploy* diminui o número de passos para a execução de *scripts* no ambiente conteinerizado e o Capítulo 5 conclui esta dissertação.

Capítulo 2

Virtualização em contextos científicos

Esta dissertação se utiliza de tecnologias de virtualização no apoio à automatização da execução de *scripts* científicos com coleta de dados de proveniência. A virtualização é uma técnica muito antiga apontada como um dos componentes mais importantes da computação em nuvem [32]. Seu sucesso se deu devido à redução de custo e complexidade das aplicações, assim como sua flexibilidade e escalabilidade [25, 33], além de prover isolamento [34]. Apesar das tentativas de adoção de máquinas virtuais em PAD [35–37], a sobrecarga adicionada pela estrutura das máquinas virtuais é alta (*e.g.*, a inicialização lenta), e por isso seu uso foi evitado [38, 39], favorecendo contextos muito específicos [35].

Máquinas virtuais são estruturas complexas e como consequência disso, adicionam uma sobrecarga significativa à carga de trabalho computacional, o tempo para inicializar ou finalizar é alto [34]. Uma alternativa ao seu uso é a virtualização baseada em contêiner ou conteinerização, que provê rápida inicialização e finalização, mantendo a flexibilidade, portabilidade e escalabilidade oferecida pelas máquinas virtuais. Apesar da virtualização baseada em contêineres ter se tornado popular por meio do Docker [26], a ideia de isolar processos sem a necessidade de simular um *hardware* diferente para cada processo é antiga, primeiramente executada com os LXC (*Linux Containers*) [40]. Em ambientes comerciais a conteinerização tem sido adotada com êxito [29]. O Docker atualmente é a solução mais adotada para contêineres em meios comerciais, oferecendo reprodutibilidade e portabilidade a aplicações de modo geral. Entretanto, o uso de contêineres em PAD só se tornou comum após a introdução de outras tecnologias de contêiner como o Singularity [2].

Neste capítulo são apresentados conceitos de virtualização (Seção 2.1) as diferentes abordagens de virtualização, suas vantagens e desvantagens e como podem ser utilizadas em ambientes de PAD (Seções 2.2 e 2.3). A Seção 2.4 apresenta as dificuldades encontradas na adoção de coleta de dados de proveniência em ambientes

virtualizados. Por último, a Seção 2.5 apresenta diferentes trabalhos que buscam auxiliar a coleta de dados de proveniência em aplicações.

2.1 Máquinas virtuais

GOLDBERG [41] define máquinas virtuais como "uma duplicata eficiente e isolada de uma máquina real", sua utilização se tornou uma alternativa para vários sistemas de computação por conta dos custos, portabilidade e mesmo segurança. Outras vantagens das máquinas virtuais que podem ser citadas são balanceamento de carga e tolerância a falhas. Tais características permitem maior flexibilidade e menor complexidade no gerenciamento de recursos físicos em ambientes virtualizados [42].

Quanto ao funcionamento, máquinas virtuais emulam um computador completo, *hardware* e *software*, permitindo múltiplas máquinas virtuais em uma mesma máquina física. Para tal tarefa, é necessária uma camada de virtualização, *hypervisor* ou monitor de máquina virtual que pode ser um *firmware* ou *software*. Esta camada pode estar sobre um sistema operacional (SO) hospedeiro (*hypervisor* tipo 2), ou diretamente sobre o *hardware* da máquina hospedeira (*hypervisor* tipo 1) como ilustrado na Figura 2.1. Os *hypervisors* tipo 1 ou *bare-metal hypervisors*, diretamente instalados no *hardware*, são os mais eficientes, com melhor desempenho e não apresentam conflitos com definições de SO, ou dispositivos, uma vez que não há a presença de um SO hospedeiro. Esse fato também provê maior segurança, e isolamento entre diferentes máquinas virtuais executando no mesmo *hardware*. Os *hypervisors* tipo 1 são os mais utilizados comercialmente e alguns exemplos são VMware ESXi [1], Microsoft Hyper-V e KVM [2] que é de código aberto. Os *hypervisors* tipo 2 são instalados sobre um SO e dependem do mesmo para gerenciar os recursos da máquina hospedeira. Este tipo de abordagem se origina na virtualização x86, quando o *hypervisor* era implantado como uma camada superior de *software*. Neste tipo de virtualização, existe a presença de um SO como base, isto gera um inevitável aumento na latência uma vez que todas as atividades da máquina virtual passam pelo SO hospedeiro. Isso gera vulnerabilidades no SO hospedeiro, comprometendo assim todas as máquinas virtuais neste hospedadas. O uso comercial de *hypervisors* tipo 2 não é comum, sendo mais utilizados em computadores pessoais para fins que não objetivam desempenho e não há muito interesse em segurança como teste de *software*. Alguns exemplos de *hypervisors* tipo 2 são o VMware Fusion [3], o Oracle VM VirtualBox [4] e o Oracle VM Server for x86 [5].

[1] https://www.vmware.com/br/products/esxi-and-esx.html
[2] https://www.linux-kvm.org/
[3] https://www.vmware.com/br/products/fusion.html
[4] https://www.virtualbox.org/
[5] https://www.oracle.com/br/virtualization/vm-server-for-x86/

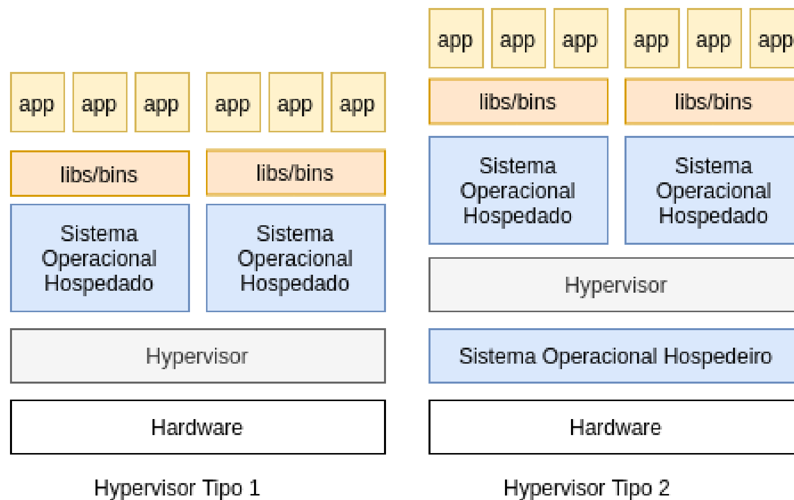

Figura 2.1: Diferentes arquiteturas de *hypervisor*. Adaptado de [35]

Máquinas virtuais foram amplamente adotadas em ambientes comerciais, sendo fundamentais para serviços de computação em nuvem, uma vez que a virtualização permite isolamento de cargas, portabilidade, controle e distribuição de recursos. A portabilidade de máquinas virtuais é uma característica que chama atenção de usuários de PAD, pois permite migrar ambientes de execução virtualizados entre diferentes máquinas físicas. Desta forma, o mesmo código pode ser executado em máquinas distintas, descartando a necessidade de cumprir os requisitos da pilha de *software* para cada hospedeiro, aumentando a produtividade ao evitar retrabalho [35]. Por outro lado, há uma sobrecarga adicionada causada pela complexidade de virtualizar *hardware* e *software* [25] e pela necessidade de garantir isolamento [43]. Ainda, o espaço ocupado em disco é grande pois é necessário armazenar um SO completo para cada máquina hospedada, o que gera alto custo de inicialização e parada e baixa densidade por hospedeiro [25, 44]. Estas desvantagens são negligenciáveis em ambientes comerciais, pois a perda de desempenho não é tão significativa, mas em aplicações científicas, em que há uma maior necessidade de desempenho, estas características se tornaram um impeditivo a uma adoção abrangente em ambientes PAD. Por isto algumas soluções de virtualização voltadas para ambientes PAD foram desenvolvidas e obtiveram sucesso inicial (FutureGrid [45], Magellan [46], Chameleon Cloud [47], Hobbes [48]), porém seus benefícios foram restritos a problemas específicos [29]. Outra abordagem de virtualização mais recente é a virtualização baseada em *contêiner* ou em nível de SO apresentada na próxima seção.

2.2 Virtualização baseada em contêineres

Contêineres podem ser compreendidos como “recipientes que oferecem um ambiente o mais próximo possível como o que você obteria de uma máquina virtual, mas sem

a sobrecarga que acompanha a execução de um *kernel* separado e a simulação de todo o *hardware*"[49]. Contêineres realizam a virtualização em nível de SO compartilhando recursos através de abstração de camadas de rede, processos entre outros [26]. A conteinerização modifica o SO existente para prover um isolamento extra. Este processo envolve adicionar um identificador do contêiner para cada processo e adicionar verificações de controle de acesso a cada chamada do sistema [34]. A virtualização baseada em contêiner é uma opção mais leve que a virtualização baseada em *hypervisor* [26] e pode-se destacar que existe a possibilidade de transportar códigos em nível de usuário, há compartilhamento de *kernel* sem perda do isolamento, e quando comparados às maquinas virtuais, é possível executar um número ostensivo de contêineres em um mesmo hospedeiro com inicialização e parada rápida [44].

Contêineres encapsulam as dependências e coleções de *software* necessárias a execução de uma tarefa específica em um ambiente isolado, criando um ecossistema de *software* portável. De maneira distinta às máquinas virtuais, contêineres dispensam a abstração de *kernel* uma vez que compartilham este recurso do hospedeiro e também não fazem abstração de *hardware* e não possuem uma camada de monitoramento(*hypervisor*), isto diminui significativamente o tempo de inicialização. As camadas de virtualização adicionais das máquinas virtuais geram a necessidade de tradução das instruções do SO hospedado para o SO hospedeiro [50], o que também adiciona sobrecarga às operações. Estas características favorecem aplicações comerciais, visto que na atualidade, estas possuem um grande número de dependências de diferentes aplicações, que podem gerar conflitos. O uso de contêineres atende este problema executando estas aplicações, de maneira fragmentada, em diferentes contêineres, uma vez que a densidade por hospedeiro é alta [44].

Assim como máquinas virtuais, os contêineres executam a pilha de *software* sobre o *hardware* disponibilizado. Nas aplicações executadas em nuvens computacionais, contêineres são utilizados para criar serviços em larga-escala com baixo acoplamento, onde cada contêiner se dedica a executar apenas um processo de usuário, também chamado microsserviço. Segundo ALSHUQAYRAN *et al.* [51] "A arquitetura de microsserviços tem seu foco na divisão de um sistema em serviços pequenos e leves que são projetados para executar funções de negócio de maneira coesa, e é uma evolução da arquitetura orientada a serviço (SOA)". Cada microsserviço é um fragmento de uma funcionalidade que é implementado e operado como um sistema pequeno e independente, oferecendo acesso a sua lógica interna e aos seus dados através de uma interface de rede bem definida [52]. O uso de microsserviços facilita o desenvolvimento, proporciona rápida entrega, melhor manutenção e integração e maior autonomia. Os microsserviços são uma alternativa à construção de aplicações singulares e autossuficientes. O papel dos contêineres nesta arquitetura é encapsular estes fragmentos. Um exemplo comparativo pode ser dado através de um sistema

de biblioteca, temos que o empréstimo de livros pode ser uma das suas principais funcionalidades, no entanto, há também o cadastro de acervo, usuários, funcionários e etc. Quando este sistema é desenvolvido em uma aplicação singular, a alteração de funcionalidades implica que toda a aplicação deva ser compilada, testada e reinicializada para que seja feita a entrega. Com microsserviços, apenas a funcionalidade modificada precisa passar por este processo, que é favorecido pela inicialização e parada rápida de contêineres. Como são implementados e operados de maneira independente, indisponibilidade ou erro em um microsserviço não impossibilita o uso do sistema, apenas do fragmento específico. Outra vantagem do uso de contêineres nesta abordagem, é a possibilidade de aumentar ou diminuir os recursos utilizados por cada microsserviço. Esta divisão de um sistema em fragmentos menores através da utilização de contêineres também facilita a configuração, pois cada contêiner encapsulará a parte da pilha de *software* que corresponde ao fragmento executado.

Enquanto máquinas virtuais dependem do *hypervisor* para interagir com o SO hospedeiro, contêineres fazem chamadas diretamente ao *kernel*. Contêineres podem ter um *daemon* que é um programa que executa em segundo plano no SO hospedeiro e que gerencia a execução, criação e interação com contêineres. Algumas tecnologias de contêiner no lugar do *daemon* possuem um *launcher*, que possui as mesmas características, exceto a necessidade de execução com segundo plano. Em contêineres a escalabilidade é alcançada a partir de balanceamento de carga e serviço de provisionamento. Para aumentar a utilização do sistema, muitos contêineres são hospedados em uma mesma máquina física. Contêineres são instâncias de imagens, que são arquivos somente leitura que contêm bibliotecas, dependências e arquivos necessários a execução de uma determinada configuração. Imagens também podem servir de base para criação de novas imagens, estas são armazenadas local ou remotamente. Isto possibilita portabilidade e reprodutibilidade de ambientes, diminuindo as diferenças entre ambientes de desenvolvimento, teste e produção. É garantido o funcionamento de aplicações desenvolvidas em diferentes ambientes. O uso de registros públicos onde as imagens podem ser baixadas e utilizadas instantaneamente é bastante útil para construção de imagens. Contudo, alterar imagens é um processo demorado, uma vez que a compilação precisa ser feita novamente, para acelerar este processo, algumas tecnologias de conteinerização implementam *copy-on-write*. *Copy-on-write* é uma técnica que otimiza a cópia de recursos de dados em um computador, quando um arquivo é copiado sem que hajam modificações, a cópia é uma referência ao arquivo original e um novo arquivo só é escrito quando são feitas modificações sobre o arquivo original. Isto possibilita uma instanciação mais rápida com utilização de ponteiros para os arquivos existentes [43]. O *copy-on-write* também possibilita a sobreposição de camadas em contêineres, onde diferente contêineres podem ter o mesmo contêiner base ou servir de base para outros contêineres [43].

Soluções de contêineres e suas principais diferenças são listadas a seguir:

- **Linux Containers (LXC)** - projeto iniciado em 2008, definido como uma interface para acesso aos recursos de contenção do *kernel* Linux, dando suporte ao isolamento e multilocação [53]. Estes contêineres oferecem virtualização completa de modo que é possível executar múltiplas instâncias de Linux sobre o mesmo *kernel*, com inicialização ágil [54]. Esta foi a solução de contêineres mais utilizada até 2013, quando foi lançado o Docker. O LXC possui algumas funcionalidades do *kernel*, seu controle de recursos é feito através de *cgroups*, o isolamento por processo através de *namespaces* (de processo, usuário e rede) e o *chroot* faz o controle aparente do sistema de arquivos.

- **Docker** [26] - é a solução mais adotada no mercado [29], presente nas maiores plataformas de serviços de computação em nuvem como AWS, Azure e Google Cloud, vem sendo amplamente utilizado na indústria [53]. Os contêineres Docker são baseados na tecnologia dos *Linux Containers*, que são apontados como o núcleo do Docker. O diferencial inicial do Docker foi prover uma interface intuitiva para utilizar tecnologias em nível de *kernel*(linux containers, *cgroups* e sistemas de arquivo *copy-on-write*) [26], e para fazer o isolamento dos contêineres do hospedeiro, faz uso de *namespaces* em nível de *kernel*. Os *namespaces* de usuário separam dados de usuário do hospedeiro e do contêiner, assegurando que o superusuário do contêiner não possua os mesmos privilégios no hospedeiro. Os *namespaces* de processo são responsáveis por exibir e gerenciar apenas processos executando no contêiner. E os *namespaces* fornecem ao contêiner um dispositivo de rede próprio e um endereço de IP virtual. Outro componente fornecido pelo LXC ao Docker é o *control groups (cgroups)* que permite o controle dos recursos utilizados pelos contêineres como memória, espaço em disco, e operações de entrada e saída [26]. *Cgroups* também gera muitas métricas sobre os recursos consumidos por contêineres e o Docker faz o controle dos recursos consumidos por processos dentro dos contêineres de modo que haja uma divisão justa de recursos por processo [26].

  As instâncias Docker quando em execução, são subprocessos do Docker *daemon*. O Docker *daemon* executa na máquina hospedeira, responsável pela inicialização de contêineres, seu nível de isolamento, monitoramento para disparo de ações (reinicialização, parada, *etc*) e geração de sessões interativas [55]. Este pode modificar tabelas de IP no SO hospedeiro e criar interfaces de rede. O *daemon* do Docker também é responsável pelo gerenciamento de imagens, enviar ou baixar de registros remotos, compilar a partir arquivos de configuração de contêiner(*dockerfiles*), *etc*. O Docker *daemon* precisa ser executado com privilégios de superusuário no SO hospedeiro e é remotamente

controlado através de um *socket* UNIX e também permite o controle via TCP, sem necessidade de uso da interface *shell* no hospedeiro. A posse desse *socket* determina quais usuários tem acesso privilegiado aos contêineres no hospedeiro através dos comandos do Docker. Os acessos via *socket* possibilitam acesso de superusuário no hospedeiro. O *daemon* então se torna alvo potencial para ataques que podem acontecer do hospedeiro para os contêineres através de modificação de configurações de segurança ou dos contêineres para o hospedeiro com envio de comandos de superusuário para o hospedeiro [55]. A execução de código não confiável pelo *daemon* com privilégios de superusuário através de imagens com conteúdo malicioso ou durante a transferência de imagens quando o uso de certificados TLS de entidades certificadoras é desabilitado facilita ataques do tipo "*man-in-the-middle*"[55, 56]. É observado que a escalada de privilégios dentro de contêineres Docker a partir do *daemon*, torna a máquina vulnerável a ataques maliciosos uma vez que acessando um contêiner, comandos podem ser enviados ao SO hospedeiro [21, 29, 39].

O Docker distribui aplicações na forma de imagens [57] que são arquivos, somente leitura, compostos de múltiplas camadas, estas imagens podem ser sobrepostas, como camadas, que contêm os arquivos binários, as bibliotecas necessárias a estes e arquivos de configuração para execução de uma determinada aplicação [25], estas imagens podem ser importadas ou exportadas. Contêineres Docker, são executados a partir de imagens Docker, que podem ser locais ou remotas, armazenadas em registros públicos. A possibilidade de compartilhar facilmente imagens deu início ao *Docker Hub* [57], um registro público da Docker Inc. que tem por função distribuir imagens oficiais ou da comunidade, fornecidas diretamente por empresas de *software*, ou da comunidade de usuários do Docker. As imagens nos registros são armazenadas em repositórios que permitem versionamento e manutenção [57]. O *Docker Hub* possui repositórios públicos e privados que podem ser oficiais ou da comunidade. Os repositórios de comunidade são muito maiores em número e em imagens disponibilizadas quando comparados aos repositórios oficiais. As imagens existentes possuem um grande número de vulnerabilidades, quando consideradas todas as versões, muitas não recebem atualizações por longos períodos e as vulnerabilidades são herdadas a partir de contêineres bases [57]. Quanto ao desempenho, as imagens de comunidade podem não ter sido geradas a partir da observação das boas práticas do Docker que visam a construção de imagens eficientes.

- **rkt** - lançado pelo CoreOS Inc., esta tecnologia tem a proposta prover maior segurança [31], sua interface é um arquivo executável [58]. O uso de *daemons*

confiáveis é dispensado e o uso de *namespace* de usuário é opcional [31]. Esta tecnologia também possui muitas funcionalidades desnecessárias à ambientes PAD. O rkt suporta imagens Docker e provê uma linguagem própria pra especificação de imagens [31]. Como não possui um *daemon* centralizado, este é facilmente integrável com sistemas de inicialização e sistemas avançados em *cluster*. As principais funcionalidades desta tecnologia objetivam segurança, composição e padronização. O rkt divide as permissões a partir de uma função, esta função é projetada para evitar o uso de privilégios elevados de maneira desnecessária, também fornece a verificação de assinatura padrão de imagens [39] e outros recursos avançados para considerações de segurança. A padronização é feita por meio de múltiplas ferramentas possam compartilhar um mesmo ambiente, que é um critério condutor para o usuário compilar e empacotar imagens de contêineres. Rkt é projetado para prover compatibilidade retroativa a vários padrões de contêiner. Por conta da segurança presente em contêineres rkt, esta é apontada como um opção para ambientes PAD [39], porém, esta abordagem requer privilégios elevados de administrador para execução de parte de suas funcionalidades.

A despeito dos benefícios da conteinerização, pode-se observar que o isolamento provido por contêineres é fraco, pois a virtualização se dá apenas em nível de SO [59]. Além disso, há interferência no desempenho quando muitos contêineres executam juntos em uma mesma máquina física [44]. Essa falta de isolamento traz vulnerabilidades ao ambiente de execução. Para garantir maior isolamento é indicado o uso conjunto de contêineres com máquinas virtuais [26, 44] provendo maior segurança ao ambiente. Esta combinação mantêm o isolamento e a facilidade e rapidez de inicialização e parada, possibilitando a rápida entrega, mas não é viável em PAD. O tempo de compilação é proporcional a quantidade e complexidade dos componentes presentes na pilha de *software* encapsulada no contêiner. Alterar imagens implica realizar novamente sua compilação completa. Quanto ao desempenho, contêineres são muito mais eficientes quando comparados à máquinas virtuais, mas no caso das aplicações científicas a sobrecarga adicionada ainda é considerável. Contêineres são agnósticos à *hardware*, alterações que visam melhoria de desempenho precisam ser feitas pelo usuário.

Algumas características de contêineres como isolamento e portabilidade, são desejáveis em ambientes PAD. Entretanto, as desvantagens listadas anteriormente são fatores que inviabilizam sua adoção direta para aplicações científicas. Por isto algumas soluções com foco em ambientes PAD foram desenvolvidas e são apresentadas na próxima seção.

2.3 Contêineres em ambientes de PAD

Contêineres dentro de ambientes de alto desempenho, são interessantes por facilitar a reprodutibilidade e portabilidade de ambientes de execução. As soluções de contêiner desenvolvidas para ambientes PAD visam promover maior praticidade no processo de preparação de ambientes, permitindo flexibilidade para utilizar gerenciadores de pacotes, autonomia na instalação de bibliotecas e liberdade de customização. No entanto, parte dessas abordagens dependem que o Docker esteja presente na máquina ou não são portáveis. O uso de contêineres em PAD se dá da seguinte maneira: uma imagem é compilada contendo a aplicação do usuário, suas dependências, possíveis dados de entrada, variáveis de ambiente, comandos de inicialização, etc. Uma vez gerada esta imagem o usuário pode transferi-la entre diferentes ambientes e realizar a execução de sua aplicação sem esforço adicional. Havendo necessidade de alteração de algum componente contido na imagem, essa deve ser compilada novamente, que é um processo custoso. Contêineres também são muito utilizados em aplicações comerciais para resolver conflitos de dependências, isto é feito através da separação dos componentes conflitantes em diferentes contêineres [26]. Essa característica é proveitosa para ambientes PAD, mas descartada pois o indicado é que o usuário empacote toda a sua aplicação e dependências correspondentes em um mesmo contêiner. Se a pilha de *software* da aplicação for complexa, quando encapsulada em uma mesma imagem, os conflitos de dependência continuarão a existir, porém, dentro do contêiner.

A seguir são listadas diferentes abordagens de contêineres voltadas para ambientes PAD.

- **Shifter** [30] - desenvolvido pelo *National Energy Research Scientific Computing* (NERSC) [21] é um pacote que permite que imagens criadas por usuários sejam executadas em seu supercomputador, as imagens podem ser Docker ou outros formatos. Com o Shifter é possível ao usuário criar uma imagem do sistema operacional desejado e instalar suas pilhas de *software* e dependências. Especificamente para imagens Docker, o Shifter permite a execução em qualquer outro centro PAD que disponibilize o uso do Docker, e com relação ao Docker, o Shifter possui melhorias de desempenho, especialmente para bibliotecas compartilhadas. Esta é a primeira solução de contêiner voltada para PAD, como foco em pilhas de *software* definidas pelo usuário. Assim como Docker, essa solução demanda acesso privilegiado e a sobrecarga é considerada significativa para aplicações em larga escala.

- **CharlieCloud** [31] - desenvolvido pelo Los Alamos National Lab.(LANL), é um conjunto de *scripts* que permite executar um *cluster* virtual de máquinas

virtuais em um computador pessoal ou em um supercomputador. Baseado em Docker, porém com remoção de privilégios, permite criar e atualizar imagens, executar múltiplas máquinas virtuais em um *cluster* virtual. As máquinas virtuais podem se comunicar via rede e com aplicações externas ao ambiente. O objetivo principal é permitir aos usuários ter um *cluster* virtual contendo nós computacionais onde há acesso privilegiado, enquanto estes privilégios são restritos ao *cluster* virtual não afetando os recursos computacionais da máquina física. Esta abordagem não apresenta sobrecarga significativa em aplicações MPI.

- **Singularity** [21] - com foco em mobilidade de ambientes, é a solução mais adotada em centros de PAD e busca atender usuários e ambientes PAD. Singularity proporciona meios seguros de capturar e distribuir *software* e ambientes computacionais. Assim como outras soluções, o Singularity é capaz que baixar e executar imagens Docker, no entanto, permite a criação de imagens próprias [21]. O Singuarity vem com suporte nativo a GPU e MPI. Contêineres Singularity executam como processos do usuário responsável pela execução e as permissões do usuário dentro do contêiner são iguais às suas permissões fora. Não há um *daemon* que executa como superusuário, e não existe possibilidade de escalada de privilégios dentro do contêiner. Por atender a contextos científicos, contêineres Singularity dão suporte a SO legados, e de maneira distinta aos contêineres Docker, as boas práticas indicam o encapsulamento de todos os componentes e dependências da aplicação do usuário. O principal objetivo é permitir a execução em diferentes ambientes a partir de uma única imagem. O arquivo de configuração para criação de imagens Singularity é similar à arquivos RPM, com pares chave-valor que definem informações sobre a imagem e seções separadas onde o usuário pode copiar arquivos, instalar bibliotecas, definir variáveis de ambiente, definir comandos de inicialização e etc. Diferente do *dockerfile*(arquivo de configuração para imagens Docker), não há um conjunto de instruções próprias pré-definidas, os comandos para criar imagens Singularity a partir do arquivo de definição são os comandos do SO Linux que a imagem utiliza.

  O Singularity também possui um registro público de imagens e arquivos de definição, o Singularity Hub. O Singularity Hub tem por propósito compilar automaticamente imagens e implantar contêineres rapidamente. Assim como o Docker Hub, este registro possui versionamento e faz uso do GitHub para tal. As imagens e arquivos de configuração presentes no Singularity Hub são compiladas e verificadas pelo Singularity Hub.

2.4 Coleta de dados de proveniência em ambientes virtualizados

Os dados de proveniência se referem à origem ou procedência de algo. Os dados de proveniência são utilizados para apontar origem ou histórico de um determinado elemento de dado. Em aplicações científicas este tipo de dado pode auxiliar na tomada de decisão e compreensão das atividades [10]. Com a coleta de proveniência, é possível entender e refazer todo o caminho que um dado percorreu e todas as transformações sofridas por este. Perguntas relativas à execução de uma aplicação são facilmente respondidas, como : *(1) o que ou quem originou um determinado elemento de dado?; (2) quando ou como um elemento de dado foi modificado?; (3) quais dados foram gerados a partir da entrada inicial.* A coleta de dados de proveniência pode ser feita de maneira automática ou através de instrumentação.

Ferramentas de coleta de dados de proveniência automática tem por característica uma integração [60] e implantação fáceis. A granularidade dos dados coletados é alta, isto não é viável para aplicações de CSE, pois estas já geram um volume grande de dados. Esta granularidade dificulta a análise dos dados pois o usuário tem de utilizar técnicas e programas de processamento e análise de dados. Em muitos casos, o usuário deseja observar comportamentos específicos, então parte significativa dos dados coletados é descartada. Outra característica das ferramentas de coleta automática é a adição de sobrecarga significativa [60], o que também inviabiliza sua adoção em PAD.

Ferramentas de coleta de dados de proveniência baseadas em instrumentação necessitam que o usuário adicione as chamadas para coleta ao seu *script*. Isto implica acesso ao código, o que não é um problema para aplicações de CSE [16]. A granularidade dos dados é definida pelo usuário e a sobrecarga adicionada depende do número de chamadas adicionadas ao *script* [60]. Uma vez que o usuário define quais dados serão coletados, a análise é se torna simplificada dispensando a necessidade de utilizar programas e técnicas para processamento e análise de dados. Algumas ferramentas de coleta de dados de proveniência baseadas em instrumentação que podem ser citadas são a DfAnalyzer [61] e o YesWorkflow [62], tratados nesta dissertação como coletores de dados de proveniência. Estas ferramentas não são de fácil implantação [60] por utilizarem diferentes componentes e bibliotecas que precisam ser instalados, configurados separadamente e executados em uma ordem específica. Quando em ambientes PAD, os problemas relativos a configuração se agravam, uma vez que o usuário tem restrições quanto a privilégios. Muitos componentes que são baixados automaticamente através de gerenciadores de pacotes ou junto a outras bibliotecas, precisam ser instalados ou compilados direto do código fonte que é uma tarefa demorada, trabalhosa, propensa a erros e pouco documentada. Um compo-

nente presente em diferentes ferramentas de coleta de dados de proveniência é o SGBD, podendo ou não ser relacional, estes sistemas comumente são instalados com privilégios de superusuário, pois gerenciam e criam arquivos e índices dentro destes arquivos. Um SGBD cria instâncias próprias, executa requisições de entrada e saída, e se utiliza de *buffers* para reduzir entrada e saída. Estes sistemas ainda dependem de informações do SO hospedeiro como informações regionais (formatação numérica, idioma e etc). E quando dentro de contêineres é importante que estes sejam projetados de modo a mapear estes arquivos e capturar as informações necessárias fora do contêiner, pois em caso de falha, estes arquivos poderão ser recuperados. Ainda, SGBDs assim como outras aplicações, se comunicam via rede utilizando portas, e estas precisam estar expostas dentro do contêiner.

Atualmente, é possível encontrar imagens de aplicações de coleta de dados de proveniência em registros públicos de contêineres. Porém, essas imagens, em muitos casos, são apenas para testes dentro do contêiner, perdendo os dados quando o usuário sai do contêiner. Também é importante observar que essas imagens encapsulam toda a ferramenta de proveniência, e alteração resulta em muito tempo de compilação pra criar uma nova imagem. Utilizar os conceitos de microsserviços para ferramentas de coleta de dados de proveniência é conveniente quando existe uma necessidade grande de flexibilidade e troca de componentes. Alguns componentes dessas ferramentas, como SGBDs, possuem imagens oficiais em registros públicos prontas para uso em conjunto com as instruções para uso sem perda de dados, abertura de portas e conexão. Ainda, imagens pequenas, que contêm poucas configurações, têm compilação rápida, o que acelera a alteração de componentes. Entretanto, quanto mais instâncias de contêiner executando juntas maior a sobrecarga para inicialização e parada [44, 63]. Do ponto de vista da portabilidade, múltiplas imagens precisam ser transferidas. É necessário um balanceamento entre portabilidade e flexibilidade, de modo que seja fácil modificar componentes em pouco tempo não perdendo mobilidade. Propõe-se com a *ProvDeploy* dividir os componentes das ferramentas de coleta de dados de proveniência a partir da funcionalidade que executam. Deste modo, componentes relacionados a armazenamento são conteinerizados em conjunto observando as necessidades que o armazenamento de dados através de contêineres requer. A aplicação de coleta de dados de proveniência é encapsulada juntamente com seus arquivos de configuração e bibliotecas. A imagem para execução da aplicação do usuário é conteinerizada junto das bibliotecas de apoio à coleta de dados de proveniência em *scripts*.

2.5 Trabalhos relacionados

Contêineres possibilitam a criação de cópias da pilha de *software* do usuário de maneira isolada podendo ser transferidas posteriormente para diferentes máquinas físicas. Isso diminui o tempo gasto na manutenção da pilha de *software* em diferentes máquinas físicas, pois essas cópias quando criadas estão prontas para execução, dependendo apenas que alguma tecnologia de conteinerização compatível com a cópia esteja presente na máquina física. Para auxiliar a coleta de proveniência e a execução de aplicações, diferentes abordagens utilizando contêineres são propostas e em alguns casos combinando ferramentas de maneira complementar. Nesta seção são apresentados trabalhos com foco em auxiliar o uso de contêineres em PAD e/ou adoção de coleta de dados de proveniência.

A crescente adoção de contêineres em PAD, que proporciona portabilidade e reprodutibilidade às aplicações, motivou o apoio a contêineres pelo Sistema de Gerência de *Workflows* Científicos (SGWfC) Pegasus [64]. Atendendo à necessidade de disponibilizar as imagens a todos os nós, foram adicionadas ao Pegasus [64], as competências necessárias para oferecer apoio à diferentes tecnologias de contêineres em variados ambientes de execução [2]. O Pegasus permite ao usuário descrever suas *pipelines* na forma de um grafo direcionado acíclico de tarefas, com formato portável e agnóstico ao ambiente. O apoio à contêineres se dá para diferentes tecnologias (Docker, Singularity e Shifter), em ambientes distribuídos, e há suporte a registros públicos (Docker Hub e Singularity Hub). Dentro do Pegasus, os contêineres são utilizados como dependências de dados de entrada. No entanto, a solução é restrita para uso no Pegasus que como um SGWfG faz toda a gerência da aplicação adicionando sobrecarga significativa. O Pegasus armazena os dados de execução em um SGBD e disponibiliza estes dados para monitoramento da execução, mas não para consultas durante a execução.

Uma solução que utiliza o SGWfC Pegasus com contêineres combinado ao dispel4py [65] é o Asterism [6]. Esta abordagem propõe a integração de um modelo *Data Intensive workflow as a Service* (DIaaS) através de contêineres. A ideia central é executar aplicações com coleta de dados em sistemas heterogêneos distribuídos, sem que o usuário se preocupe em: modificar métodos de acordo com ambientes de execução; gerenciar os dados distribuídos; paralelizar métodos; e armazenar e transferir grandes volumes de dados. Esta ferramenta pode ser utilizada em diferentes plataformas, desde que estas adotem o Docker. O dispel4py é uma biblioteca Python para descrição de *workflows* abstratos para aplicações *data intensive* distribuídas. Esta biblioteca é utilizada no gerenciamento de execuções paralelas. O Pegasus é utilizado para gerenciamento automático da transferência de dados entre diferentes infraestruturas de execução e gerenciamento da execução de aplicações. A execução

de aplicações é divida em fases, cada fase é executada por um contêiner diferente e o usuário que deseje pode criar seus ambientes de execução a partir das imagens disponibilizadas pela ferramenta. Esta abordagem, no entanto, tem seu foco em nuvens computacionais e é limitada ao Docker, que não é uma solução de contêineres amplamente adotada em ambientes PAD.

GERLACH *et al.* [7] propõe o Skyport, motivado pelas dificuldades de instalação e manutenção de *software* em diferentes ambientes (grid ou nuvens computacionais). O Skyport é uma extensão ao AWE/Shock [66], uma plataforma para análise de dados que fornece ambientes para execução de *workflows* escaláveis para dados científicos na nuvem focado em *pipelines* de Bioinformática. O Skyport reduz a complexidade para atingir o ambiente necessário para execução de aplicações complexas, provendo o AWE/Shock como um serviço na nuvem. O Skyport faz uso de contêineres Docker para resolver o problema de instalação e configuração do AWE/Shock em diferentes ambientes. O Skyport realiza a integração do Shock, um gerenciador de dados orientado a objetos, e do AWE que gerencia recursos e tarefas entre nós de execução (*workers*) durante a execução. O Shock representa os dados a partir de um identificador e dos metadados que descrevem informações de proveniência. Esta solução faz modificações ao Docker, de modo que o AWE passa a fazer a orquestração dos contêineres que executam as aplicações e as imagens são armazenadas pelo Shock.

Skyport e Asterism são soluções para disponibilização que se assemelham a *workflows* como um serviço (WaaS). WaaS são serviços que se propõem a oferecer um maneira simples, acessível e economicamente viável de configurar e executar suas aplicações a qualquer momento e de qualquer lugar. Estes serviços são *frameworks* com suporte a múltiplos usuários e são projetados para gerenciar cargas contínuas de *workflows* heterogêneos. Uma abordagem para o mesmo fim, porém, tratando de problemas relacionados ao provisionamento de recursos e problemas de escalonamento em nuvens computacionais é proposta por RODRIGUEZ e BUYYA [67]. Este trabalho é desenvolvido para escalonar cargas contínuas de *workflows* científicos submetidos por usuários a um provedor de WaaS. Para isto, é proposto o algoritmo ESPM (*Elastic resource Provisioning and Scheduling algorithm for Multiple workflows*), um algoritmo dinâmico baseado em heurística que toma decisões de provisionamento e escalonamento de recursos para satisfazer o tempo determinado para execução de *workflows* individuais, minimizando o custo de utilizar máquinas virtuais. O principal objetivo deste algoritmo é evitar a instanciação de novas máquinas virtuais, reutilizando-as quando possível. Este algoritmo estima o custo da execução considerando que as tarefas serão executadas dentro de contêineres que executam sobre máquinas virtuais. WaaS simplificam a execução de *workflows* e a coleta de dados de proveniência, entretanto, diferente da proposta desta dissertação, em am-

bientes PAD, demandam a adoção por parte da administração do ambiente. Ainda, Skyport, Asterism e Pegasus com suporte a contêineres não permitem ao usuário escolher uma ferramenta de coleta de proveniência. A ideia da *ProvDeploy*, apresentada no capítulo 3, é atenuar problemas relativos a configuração e execução de uma solução de coleta de dados de proveniência em *scripts*, disponibilizando esta através de contêineres.

Ainda dentro da disponibilização de *workflow* como um serviço (WaaS), ZHENG e THAIN [63] discutem a melhor maneira de integrar contêineres a SGWfCs. Este trabalho realiza testes de diferentes cenários de conteinerização utilizando Docker para integrar o Makeflow [68] com Work Queue [69]. Makeflow é uma ferramenta por linha de comando utilizada para execução de aplicações científicas intensivas em dados em vários sistemas de execução distribuídos. Work Queue é um mecanismo de execução leve em nível de usuário para sistemas distribuídos, este sistema possui uma biblioteca mestre e um amplo número de processos escravos que podem ser instalados e executados em múltiplas plataformas (nuvem, grid, cluster). Esta ferramenta foi modificada para fornecer com precisão a semântica utilizada pelo Makeflow. O cenários explorados são: 1) encapsular cada tarefa em nível de *workflow*; 2) executar cada processo escravo em um contêiner; 3) executar cada tarefa dentro de um contêiner; 4) executar múltiplas tarefas compatíveis dentro de um contêiner; O cenário em que contêineres eram compartilhados obteve melhores resultados já que o custo de inicialização e parada foi minimizado, entretanto, houve queda no isolamento.

Para facilitar a interoperabilidade e reaproveitamento é proposto o Boutiques [70], um sistema para compartilhamento de aplicações baseado em Docker. O objetivo principal é tornar as aplicações facilmente portáveis em contraste com os sistemas de compartilhamento existentes, que são vinculados a uma infraestrutura específica. O Boutiques representa aplicações a partir de um identificador JSON armazenado em um repositório do GitHub e uma imagem Docker contendo todas as dependências. No entanto, o Boutiques tem seu foco apenas no compartilhamento de aplicações conteinerizadas e não em sua execução.

O uso de contêineres em PAD não é intuitivo e uma ferramenta que busca incentivar o uso de contêineres facilitando sua utilização é o HPC Container Maker (HPCCM) [27]. Apesar dos registros públicos de contêineres possuírem muitas imagens e arquivos de configuração prontos, não é incomum que o ambiente PAD necessário a uma determinada aplicação não seja encontrado ou composto apenas pelas imagens ou arquivos de configuração presentes nos registros. Com isto, o usuário que deseje utilizar contêineres, necessitará escrever o arquivo de configurações manualmente, compilar e lidar com os possíveis erros deste processo, o que pode ser uma tarefa tão complexa quanto preparar o ambiente no SO da máquina física, po-

dendo até mesmo custar mais tempo. Isto se dá por que ao compilar ou instalar componentes dentro de contêineres é necessário que o usuário saiba e entenda como funcionam todos os componentes dentro do ambiente do contêiner. O HPCCM, procura auxiliar no desafio de gerar contêineres, obedecendo as boas práticas para contêineres em PAD [71]. A saída é um arquivo de configuração para gerar uma única imagem que contenha os *scripts* do usuário, os *softwares* e bibliotecas que a aplicação do usuário utiliza e etc. Quanto mais bibliotecas e componentes dentro da imagem, maior o tempo de compilação e mesmo pequenas alterações implicam que a compilação da imagem deva ser feita parcial ou completamente, a depender da tecnologia para contêiner utilizada. A pouca flexibilidade deste tipo de abordagem, não é viável quando há necessidade de alteração de *scripts*, componentes e bibliotecas, que é o caso de aplicações de coleta de proveniência. O HPCCM se concentra na configuração de ambientes, não em sua execução, e pode ser utilizado junto com a *ProvDeploy* que é apresentada no próximo capítulo.

Capítulo 3

Abordagem proposta: *ProvDeploy*

Aplicações de coleta de dados de proveniência possuem componentes específicos. Dentre estes componentes podemos citar o SGBD, servidores *web*, bibliotecas de apoio e arquivos de configuração, que, e em alguns casos exigem versões específicas de *software*. As bibliotecas pré-existentes no ambiente do usuário que servem a aplicação principal, de onde os dados serão coletados, podem entrar em conflito com as bibliotecas das ferramentas de coleta de dados de proveniência, podendo ocasionar em, nos melhores casos (excetuando-se erros e incompatibilidades), *upgrade* ou *downgrade* destas bibliotecas. Ainda, a instalação de certos componentes como SGBDs pode consumir muito tempo, é trabalhosa e, em alguns casos, mal documentada.

Com o objetivo de atenuar os problemas relativos à configuração e execução quando se deseja adotar soluções de coleta de dados de proveniência, propõe-se a *ProvDeploy*. A *ProvDeploy* pode ser descrita como um serviço de apoio à adoção automatizada de uma abordagem para coleta de dados de proveniência em *scripts* científicos. A *ProvDeploy* disponibiliza o coletor de dados de proveniência desejado pelo usuário e por meio do uso de contêineres executa a aplicação principal em conjunto com o sistema de coleta de dados de proveniência. A *ProvDeploy* utiliza imagens preparadas para execução de diferentes aplicações com coleta dados de proveniência. Esta coleta é incorporada de modo sistemático e por meio do mínimo de passos de configuração possível (definidos por meio de um modelo de custo proposto na Seção 3.2). A *ProvDeploy* automatiza diversos desses passos necessários à execução de aplicações por meio de contêineres em ambientes PAD com coleta de dados de proveniência. A seguir, apresentamos os elementos associados à *ProvDeploy*, como a linha de experimento utilizada para modelar o processo de preparação para execução, o modelo de custo e a arquitetura.

3.1 Linha de experimento da *ProvDeploy*

Uma vez que experimentos científicos computacionais envolvem a execução de variações da mesma aplicação com diferentes abordagens, programas e parâmetros. Uma não sistematização do processo de execução e captura de dados de proveniência destes experimentos faz de sua composição um tarefa complexa. A linha de experimento algébrica (LEA) [72] propõe uma metodologia que utiliza uma álgebra de *workflows* centrada em dados que introduz um modelo de dados uniforme. Uma LEA representa um experimento em uma linha de produto [73], modelando suas diversas variações e propõe a implementação em um modelo único e integrado. Opcionalidade e variabilidade são conceitos fundamentais para representar a variação em uma LEA, onde a opcionalidade define atividades que podem ou não ser suprimidas em um processo e a variabilidade define atividades que podem ser representadas por mais de uma alternativa abstrata.

Conceitos importantes da LEA utilizados nesta dissertação são os conceitos de atividades opcionais e atividades compulsórias. Atividades opcionais são atividades que podem ou não estar presentes em qualquer derivação de processo. Atividades opcionais tem a característica de não modificar entradas e saídas. Atividades compulsórias são atividades presentes em todas as variações de um processo abstrato válido. Na Figura 3.1 os retângulos com linhas pontilhadas representam atividades opcionais, retângulos com linhas sólidas, onde linhas sólidas duplicadas representam atividades mandatórias com possibilidade de variação, e os retângulos em azul representam as possibilidades de variação.

Uma vez que a composição de contêineres pode ser executada de diferentes maneiras, propõe-se utilizar a LEA apresentada na Figura 3.1 como uma metodologia ou um *guideline* para o uso de contêineres para executar aplicações com captura de dados de proveniência. São apresentadas cinco atividades: (i) escolha do ambiente, (ii) levantamento dos requisitos de *software*,(iii) criação dos contêineres, (iv) inicialização do coletor de proveniência e (v) execução. A escolha do ambiente é uma atividade compulsória pois cada ambiente possui suas particularidades, por exemplo, níveis diferentes de permissão, presença de gerenciadores de pacotes, *etc.* Todo o processo de configuração está ligado ao ambiente onde o usuário deseja executar sua aplicação (*script* científico). O levantamento dos requisitos de *software* é uma atividade compulsória em que o usuário escolhe como fazer a configuração, pois pode haver mais de uma maneira de instalar um mesmo *software*, como por exemplo, Anaconda e Pip que servem para instalação de módulos Python. É nesta atividade que o modelo de custo apresentado na Seção 3.2 deve ser aplicado para buscar a configuração com custo mínimo em razão de passos executados pelo usuário.

A atividade de criação dos contêineres também é compulsória e pode variar de

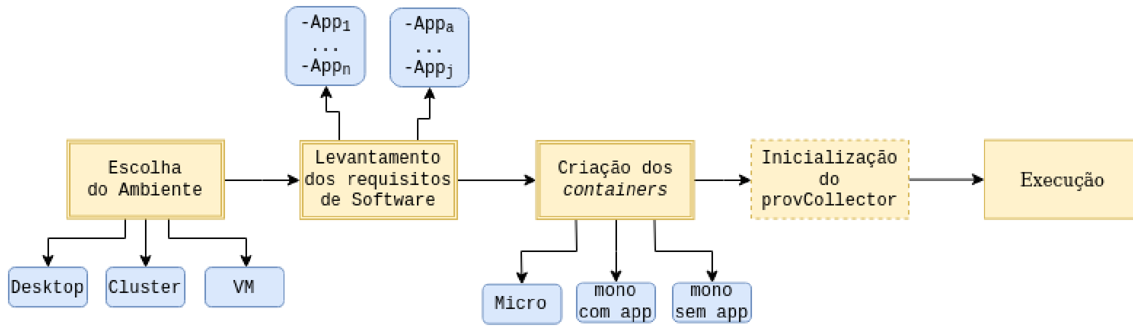

Figura 3.1: Linha de experimento proposta pela *ProvDeploy*

diversas formas, dependendo do propósito do usuário, este pode criar uma única imagem que atenda a todos requisitos (ou não) de *software* da aplicação encapsulando conforme proposto por KURTZER *et al.* [21]. Quando a aplicação é inserida dentro de uma imagem de contêiner, a execução é facilitada uma vez que apenas uma imagem será gerada, logo, o número de componentes a serem portados e mantidos é menor. O não encapsulamento de aplicações favorece o reaproveitamento da imagem, possibilitando alterar a aplicação sem a necessidade de compilar o contêiner novamente, outro benefício de não encapsular a aplicação é que diferentes aplicações que utilizem configuração similar, ou parte dos componentes presentes no contêiner também poderão ser executadas pelo mesmo contêiner. Os contêineres propostos pela *ProvDeploy* não realizam encapsulamento de aplicações visando um reuso. Com relação a inicialização do coletor de proveniência, esta tarefa é opcional pois, o usuário pode ou não querer fazer coleta de proveniência e o coletor por ele utilizado pode ou não precisar ser inicializado anteriormente à execução de uma aplicação. Por último, temos a fase compulsória de execução, onde a aplicação finalmente entra em operação, no ambiente desejado e com os *contêineres* definidos pelo usuário. Estas atividades tem por objetivo direcionar todo o processo de composição de contêineres para execução de aplicações científicas.

3.2 Definição do modelo de custo

Esta dissertação propõe um modelo de custos com o objetivo de avaliar as vantagens e desvantagens de uma determinada configuração em um determinado ambiente (virtualizado ou não). Um modelo de custos consiste em uma abstração que por meio de uma série de fórmulas que tem como objetivo estimar o custo envolvido na execução de uma determinada tarefa em um ambiente computacional específico.

Um modelo de custos é comumente utilizado em um sistema que visa a escolher um modelo de execuções alternativas, usando o modelo de custos no lugar da execução propriamente dita. Para cada alternativa, a execução é simulada e seu

custo calculado por meio do modelo, buscando a solução ótima que gere o custo mínimo. O espaço de soluções também pode ser gerado com heurísticas, em muitos casos mais adequado, quando a solução ótima não pode ser obtida em tempo viável. O modelo de custo definido para a *ProvDeploy* se baseia no conceito de *passos* envolvidos na configuração e instalação do ambiente de execução e tempo total associado a este número de passos. Por meio do modelo de custos proposto nesse capítulo, buscamos encontrar o cenário com menor número de passos para configuração da pilha de componentes de *software* de uma aplicação. Para descrever o modelo de custos proposto, estabelecemos as bases para toda a notação usada daqui em diante (resumida na Tabela 3.1).

Tabela 3.1: Notação utilizada nessa dissertação

Notação	Descrição
D	Conjunto de dependências a serem satisfeitas
A	Conjunto de possíveis ações
P	Conjunto de possíveis passos
M	Conjunto de máquinas a serem usadas
cn_i	Custo normalizado de execução do passo i
cis_j	Índice de *slowdown* computacional da máquina j
x_{ij}	Variável binária que indica se o passo i é executado na máquina j
c_{ij}	Custo para executar o passo i na máquina j

Considere o conjunto de dependências $D = \{d_1, d_2, ..., d_n\}$, onde cada d_k é uma dependência a ser respeitada pela *ProvDeploy*. Cada d_i pode ser respeitada executando ao menos uma *ação* do conjunto $A = \{a_1, a_2, ..., a_n\}$, *e.g.,* configurar uma variável de ambiente, realizar o *download* de um arquivo utilizando o comando *wget*. Em suma, uma ação é atômica no contexto dessa dissertação, e considera-se que o usuário executa estas ações sozinho na máquina. Entretanto, para resolver uma dependência d_i pode ser necessário executar uma série de ações. Por exemplo, para respeitar as dependências da *dfa-lib-python* é necessário antes instalar o *pytest* e o *coverage*.

Dessa forma, chamamos de *passo* o conjunto (não unitário, $n \geq 2$) de ações executadas para cumprir uma dependência. O conjunto $P = \{p_1, p_2, ..., p_n\}$ representa os possíveis passos a serem executados de forma a cumprir uma determinada dependência d_i. Cada elemento p_i possui um custo associado em unidades de tempo normalizado chamado cn_i. Esse custo é obtido por meio de uma estimativa sobre a execução de cada ação do passo p_i e não está associado a uma execução específica em determinadas máquinas ou ambientes.

O ambiente é composto por máquinas (sejam elas homogêneas ou heterogêneas). Consideremos o conjunto $M = \{m_1, m_2, ..., m_l\}$ como l máquinas passíveis de uso pela *ProvDeploy*. Cada máquina m_j possui um índice de *slowdown* computacional

associado denominado cis_j, onde $csi_j \supset [0, ..., 1] \; | \in \mathbb{R}$. De acordo com NASCIMENTO *et al.* [74], o índice de *slowdown* computacional cis_j é um valor que é inversamente proporcional a um valor normalizado do processamento de referência de p_i. Ou seja, para descobrirmos o custo em unidades de tempo de executar o passo p_i na máquina m_j temos que multiplicar cis_j por cn_i, *i.e.*, $c_{ij} = cis_j \times cn_i$.

Dessa forma, o custo de executar uma aplicação com k dependências com a *ProvDeploy* é $\sum_{m_j \in M} \sum_{p_i \in P} x_{ij} \times c_{ij}$, onde x_{ij} é uma variável binária que indica se um determinado passo p_i necessita ser executado na máquina m_j, e é dado como:

$$x_{ij} = \begin{cases} 1, & \text{se o passo } p_i \text{ necessita ser executado,} \\ 0, & \text{caso contrário.} \end{cases}$$

O objetivo do presente modelo de custo é possibilitar a busca pela configuração S^* onde $\sum_{m_j \in M} \sum_{p_i \in P} x_{ij} \times c_{ij}$ seja minimizado. Mais formalmente, $\exists S^* | S^* = \min \sum_{m_j \in M} \sum_{p_i \in P} x_{ij} \times c_{ij}$.

3.3 A arquitetura da *ProvDeploy*

A arquitetura da *ProvDeploy* é composta de quatro componentes principais (conforme apresentado na Figura 3.2): o *Catálogo*, o *Instrumentador*, o *Deployer* e o *Inicializador* (um *script* de inicialização dos componentes). O *Catálogo* armazena as referências para diferentes coletores de dados de proveniência, ferramentas de armazenamento e ambientes de execução conteinerizados. No *Catálogo* também ficam armazenadas as informações relativas ao versionamento de arquivos de configuração de contêineres. No *Catálogo*, as imagens podem ser cadastradas pelo usuário e podem ser locais ou remotas. A *ProvDeploy* permite ao usuário consultar as imagens existentes no *Catálogo* e solicitar uma execução utilizando um valor mnemônico na chamada do *script* de inicialização. Um fragmento de como o usuário visualiza o *Catálogo* é apresentado na Figura 3.3, onde podemos ver o campo *flag* que é um mnemônico para identificação do contêiner e uma breve descrição de suas principais características.

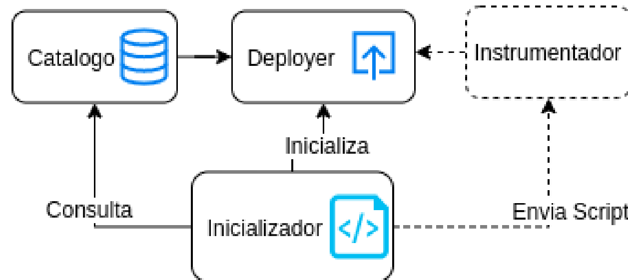

Figura 3.2: Componentes da *ProvDeploy*

```
(base) lneves@alunomovel:~/Documents/dfa-server/provdeploy/server$ python provDeploy.py -l
Flag: icc , Description: Intel compiler with dfa-lib-cpp
Flag: tensorflow , Description: Tensorflow with dfa-lib-python, tflearn, scikit, pandas and matplotlib
Flag: tensorflow2 , Description: Tensorflow 2.0 with Python 3.7, dfa-lib-python , tflearn, scikit, pandas
Flag: dfanalyzer , Description:  A data Container of dfanalyzer, MonetDB
Flag: py-readseq-modelGenerator , Description: ReadSeq, python2, java, raxml, mrbayes and garli and Ppsutils
Flag: java-readseq-modelGenerator , Description: ReadSeq, python2, java, raxml, mrbayes and garli and Jpsutil
Flag: python37 , Description: Python 3.7 with dfa-lib-python
Flag: python2 , Description: Python 2 with dfa-lib-python
Flag: dfanalyzer-with-telemetry , Description: Extended version of dfanalyzer that tracks telemetry
Flag: dataTelemetry , Description: A data Container of dfanalyzer with telemetry extension
```

Figura 3.3: Visualização do catálogo da *ProvDeploy*

A *ProvDeploy* trata os contêineres de coletores de dados de proveniência como contêineres que encapsulam toda aplicação junto com suas dependências, apenas fazendo chamadas para sua execução. Apesar da possibilidade de existência de mais de um serviço de coleta de dados de proveniência no *Catálogo*, apenas um destes pode ser definido como padrão. O serviço de coleta padrão será executado pela *ProvDeploy* automaticamente, caso seja o desejo do usuário. Os contêineres de armazenamento devem conter aplicações voltadas para realizar leitura e escrita como SGBDs e ferramentas de otimização destes processos como o FastBit [75].

O *Catálogo* da *ProvDeploy* possui também as referências de quais contêineres de armazenamento são utilizados por quais coletores de dados de proveniência. Tecnologias como *Docker*[26] e *Singularity* [21] permitem o uso dos contêineres em modo interativo (modo que permite acesso a um pseudo-terminal dentro dos contêineres). Esta funcionalidade é utilizada pela *ProvDeploy* para permitir ao usuário acesso aos dados coletados e alteração de *schemas* dos bancos de dados dentro dos contêineres de armazenamento. Os ambientes de execução conteinerizados são contêineres que não encapsulam a aplicação, estes tem por objetivo serem compilados para execução de múltiplas aplicações e para aceitarem parâmetros que o *script* de inicialização envia.

O *Instrumentador* tem papel de identificar dentro do *script* pontos de interesse do usuário e adicionar as anotações para coleta de dados de proveniência, quando necessário. Isto se dá por que alguns coletores de dados de proveniência como DfAnalyzer [61] e YesWorkflow [62] dependem de um processo de instrumentação (muitas vezes não trivial) para coleta de dados. O processo de instrumentação consiste em adicionar marcações e/ou chamadas no código para invocar um coletor de proveniência. Na versão atual da *ProvDeploy*, o processo de instrumentação foi realizado de forma manual. A ideia é que a *ProvDeploy* se aproveite de instrumentações automáticas propostas por PINA *et al.* [76] e GUEDES *et al.* [77] que venham a ser realizadas diretamente no *script* do usuário ou em frameworks como o Apache Spark, respectivamente.

O *Deployer* deve invocar o modelo de custo e apresentar ao usuário o cenário

com menor custo em total de passos. O *Deployer* também realiza o processo de preparação para a execução, fazendo uma cópia da aplicação do usuário em uma pasta temporária junto aos contêineres necessários para execução da aplicação/*script* e coleta de dados de proveniência. Caso seja o desejo do usuário que a aplicação execute remotamente, o *Deployer* realiza o acesso ao ambiente de execução a partir das informações providas pelo usuário (utilizando um arquivo com as informações de conexão do servidor de destino) e transfere os arquivos.

O *Inicializador* é responsável por conectar os componentes da *ProvDeploy*, este tem por papel acessar o catálogo de modo a consultar e adicionar imagens, inicializar a ferramenta de coleta de dados de proveniência, prover acesso aos dados, enviar ao *Deployer* os dados necessários para acesso remoto e quais arquivos serão copiados ou transferidos.

A funcionamento da arquitetura da *ProvDeploy* é ilustrado na Figura 3.4. O usuário pode submete seu *script* para a execução, utilizando o Catálogo da *ProvDeploy*. A submissão de um *script* faz com que o *Inicializador* da *ProvDeploy* busque no *catálogo* a imagem com a qual o usuário deseja executar o seu *script*, a imagem do coletor de proveniência padrão e do contêiner de armazenamento de dados. Os dados do ambiente onde ocorrerá a execução precisam ser submetidos pelo usuário (IP, usuário, senha, método de autenticação e etc). Quando o coletor de proveniência definido como padrão precisa de instrumentação, o usuário poderá requerer a instrumentação automática, a *ProvDeploy* irá chamar o *Instrumentador* que fará a instrumentação automaticamente. O *script* de saída do processo de instrumentação será copiado para execução no ambiente indicado pelo usuário. Após a instrumentação e consulta ao catálogo, o *Deployer* realiza a conexão com o ambiente de execução, quando existe esta necessidade, e faz cópias do *script* instrumentado, dos dados de entrada e das imagens que serão utilizadas no ambiente de execução.

O *script* de inicialização é implementado em Python e tem o papel de inicializar o serviço de coleta de proveniência padrão, definido pelo usuário, e de consultar o catálogo em busca da imagem que executa a aplicação desejada pelo usuário dentro de um contêiner. A partir do catálogo, onde ficam registrados os contêineres disponíveis e suas configurações, o usuário pode escolher qual melhor satisfaz os requisitos de sua aplicação. Para a execução, a *ProvDeploy* copia os dados da aplicação para o mesmo diretório do contêiner. Quando a execução necessita ser feita em um computador remoto, caso haja necessidade, as imagens dos contêineres devem ser baixadas. A *ProvDeploy* também permite adição de novos ambientes de execução e de novos serviços de coleta de proveniência.

Com relação ao armazenamento de dados de proveniência, considerando que uma mesma aplicação de coleta de proveniência pode utilizar diferentes gerenciadores de banco de dados, a *ProvDeploy* mantém o banco de dados de proveniência e a

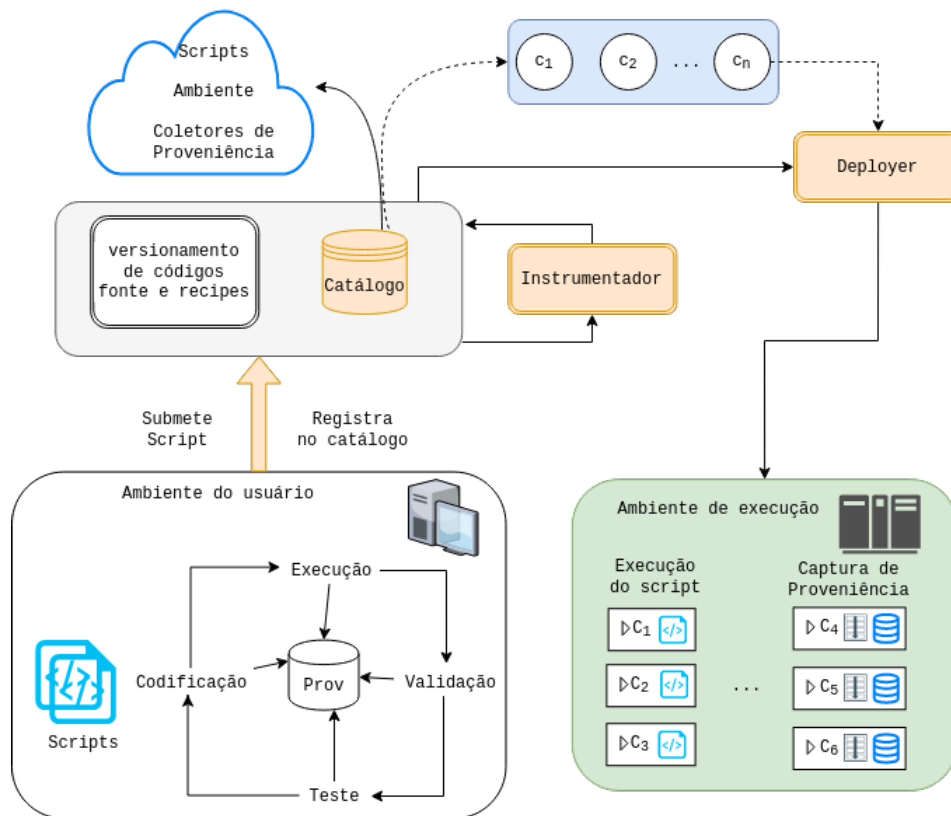

Figura 3.4: Funcionamento dos componentes da *ProvDeploy*

aplicação de coleta de proveniência em diferentes contêineres, desta forma é possível criar e utilizar diferentes bancos de dados e reutilizar para diferentes aplicações de coleta de proveniência, ou para uso em apoio a aplicações do usuário que, façam uso de contêineres. Durante e após a execução de um *script*, a *ProvDeploy* permite acesso aos dados de proveniência coletados ao possibilitar acesso ao contêiner que executa o banco de dados da aplicação de coleta de proveniência e caso exista, da interface gráfica da aplicação de coleta de proveniência.

Com relação à execução, a *ProvDeploy* trabalha com contêineres que encapsulam um grupo de configurações que pode ser utilizado para execução de diferentes aplicações, onde o *script* funciona como uma entrada para o contêiner. Com relação aos contêineres voltados exclusivamente à execução de um serviço de coleta de proveniência ou banco de dados, tudo necessário é encapsulado no contêiner e para inserir modificações, é necessário acesso direto em modo escrita.

Funcionalidades

A *ProvDeploy* conta com quatro funcionalidades principais. A seguir descrevemos cada uma delas.

- <<Listar>> - mostra todos as imagens existentes no catálogo do usuário

acompanhados por um nome identificador e uma breve descrição da configuração contida na imagem, como pode ser visualizado na Figura 3.3

- <<Iniciar>> - inicializa o serviço de proveniência definido como padrão junto a aplicação que o usuário deseja executar, este comando espera um arquivo JSON com os dados para execução da aplicação (identificador da configuração, diretório da aplicação, arquivo de execução e ambiente), a Figura 3.6 mostra um exemplo deste arquivo. Se o usuário desejar realizar a execução em uma máquina remota, os dados da máquina em questão precisarão ser preenchidos. Através do arquivo de submissão o usuário pode definir a quantidade de recursos (CPU e memória) que os contêineres poderão utilizar, no entanto, esta definição só é aplicada em ambientes que o usuário pode fazer alteração dos recursos utilizados pelo contêiner.

- <<Adicionar>> - adiciona uma nova imagem ao catálogo de configurações. Esta funcionalidade também tem por entrada um arquivo JSON com os dados da nova imagem a ser registrada no *Catálogo*. Um exemplo deste arquivo pode ser visualizado na Figura 3.5

- <<Acessar ambiente>> - permite ao usuário o acesso aos dados de proveniência coletados e alterações no modelo de dados. Esta funcionalidade também permite acesso à diferentes imagens do *Catálogo* para inspeção.

```
{
    "new_conf": [
        {
            "flag": "python2",
            "path": "recipes/application/python2/",
            "filename": "python2.sif",
            "description": "Python 2 with dfa-lib-python",
            "runcommand" : "./recipes/application/python2/python2.sif python2",
            "url" : "void",
            "tech": "singularity",
            "type": "app"

        }
    ]
}
```

Figura 3.5: Exemplo de arquivo de entrada para adição de uma nova imagem ao *Catálogo* da *ProvDeploy*

```
{"application": "convnet_RTM1_v2.py ",
"path": "path/to/app/",
"flag" : "tensorflow",
"machine" : "local",
"memory" : "",
"cpu": ""}
```

Figura 3.6: Exemplo de arquivo de entrada para inicialização de uma nova execução com coleta de proveniência

Capítulo 4

Avaliação experimental

Este capítulo apresenta a avaliação experimental da *ProvDeploy* com diferentes estudos de caso para verificar a potencial sobrecarga adicionada pela aplicação. Os estudos de caso apresentados neste capítulo foram realizados utilizando o *cluster* computacional Lobo Carneiro (LoboC), do Núcleo Avançado de Computação de Alto Desempenho (NACAD) localizado na COPPE/ Universidade Federal do Rio de Janeiro (UFRJ). O LoboC, foi utilizado nos experimentos que foram executados diretamente do SO da máquina e utilizando a *ProvDeploy* com contêineres. O LoboC é um *cluster* SGI ICE-X com 504 CPUs Intel Xeon E5-2670v3 (Haswell), totalizando 6.048 processadores. Cada nó computacional contém 24 processadores (adicional de mais 24 processadores com Hyper-Threading) com 64GB de memória RAM. Os nós computacionais são interconectados com a tecnologia InfiniBand FDR –56 Gbs (Hypercube). Além disso, esse *cluster* computacional apresenta uma partição para armazenamento de dados usando o sistema de arquivo paralelo Intel Lustre, com a capacidade de armazenamento de 500 TB de dados.

Os experimentos apresentados neste capítulo foram realizados com dez execuções sem apoio da *ProvDeploy* e dez execuções com apoio da *ProvDeploy* de maneira intercalada, para obter uma média dos tempos e observar o comportamento da *ProvDeploy*. A utilização da *ProvDeploy* foi para disponibilização da ferramenta de proveniência automaticamente e execução da aplicação com contêiner. Os experimentos sem apoio da *ProvDeploy* foram executados sobre o SO da máquina física, que é o Suse Linux Enterprise (SLE), buscando utilizar componentes já instalados através do pacote *Modules* no *cluster* LoboC. O coletor de proveniência utilizado foi a DfAnalyzer e o contêiner de armazenamento continha o MonetDB, um banco de dados colunar utilizado pela DfAnalyzer para armazenamento de dados. A tecnologia de contêineres utilizada nos experimentos aqui apresentados foi o *Singularity* pois esta é a única tecnologia de contêiner disponível para uso no LoboC. Dentro da abstração do modelo de custos dos estudos de caso aqui apresentados, não foram consideradas dependências já existentes no ambiente uma vez que o custo de

execução destes passos seria igual a zero. Nos estudos de caso apresentados neste capítulo, a LEA foi utilizada de maneira bastante similar em todos os casos, uma vez que poucas variações foram exploradas. Para cumprir as diferentes pilhas de *software*, no lugar da configuração com custo mínimo, optou-se pela configuração que seria de melhor manutenção considerando o recomendado pela documentação oficial dos *softwares* instalados. O *cluster* Loboc não permite a compilação de imagens em seu ambiente, apenas podem ser realizadas atividades relativas a execução dessas imagens (iniciar, parar, baixar imagem, etc), por isto, as imagens foram compiladas em um outro computador e transferidas para o Loboc já prontas para uso. As imagens criadas possuem os mesmos *softwares* que foram instalados no ambiente da máquina física, sua diferença concentra-se no método de instalação. Na máquina física, alguns *softwares* da pilha tiveram de ser instalados a partir de códigos-fonte, para os contêineres foi possível o uso de gerenciadores de pacote como *apt-get* e *yum*. A inicialização do coletor de proveniência foi feita manualmente nos casos sem a *ProvDeploy* e automaticamente com a *ProvDeploy*. As atividades da LEA são executadas apenas uma vez exceto a atividade de execução.

4.1 Estudo de caso: WordCount

A primeira aplicação utilizada foi um *script* para contagem de palavras, o *WordCount*. Esta aplicação se caracteriza por realizar muitas operações de leitura. Para usar a *ProvDeploy* o *script* foi manualmente instrumentado e executado em apenas dois nós computacionais com todos os processadores, um destes nós computacionais foi dedicado ao coletor de proveniência e por ser uma aplicação bastante simples e rápida, não houve a necessidade de alocação de um número maior de nós. O código-fonte foi instrumentado de modo a realizar chamadas ao coletor de proveniência por linha lida e por palavra contabilizada, conforme a abstração apresentada na Figura 4.1, gerando muitas chamadas seguidas ao coletor de dados de proveniência, e por isso esperava-se uma sobrecarga alta. Para criar o ambiente desta aplicação foi utilizado Python 3.7 e um ambiente com os mesmos componentes foi criado para execução em uma imagem de contêiner. Uma vez criada, esta pode ser executada em múltiplos ambientes. O arquivo de entrada possui 8 MB, e contém palavras geradas aleatoriamente. Foram realizadas dez execuções do *Wordcount* com e sem a *ProvDeploy* e o tempo médio de execução sem a *ProvDeploy* foi de 57,05 segundos, com desvio padrão igual a 0,81, e com a *ProvDeploy* foi de 68,59, com desvio padrão igual a 2,5, representando um aumento de 20% no tempo total.

As execuções foram planejadas utilizando um dos possíveis cenários do modelo de custos apresentado no Capítulo 3. Assim, temos que os *passos* para executar esta aplicação com coleta de proveniência sem uso da *ProvDeploy* é dado pelo seguinte

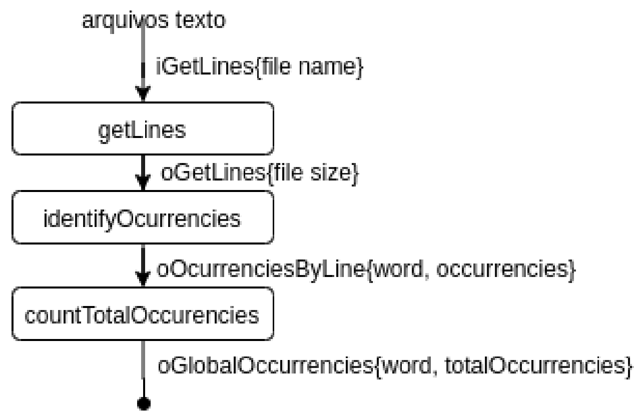

Figura 4.1: Exemplo de abstração dos dados coletados no estudo de caso do *Word-Count*

conjunto de dependências $D = \{PyEnv, Pyenv-Virtualenv, Python3.7, dfa-lib-python, DfAnalyzer, MonetDB, FastBit\}$. Como o ambiente do cluster LoboC é homogêneo, não há diferença de *slowdown index* em relação a essas máquinas. Logo, todos os *csi* envolvidos na execução são os mesmos e iguais a 1. Dessa forma, os tempos de execução normalizados são exatamente os mesmos tempos de execução coletados nessas máquinas, uma vez que o *csi* é igual a 1. O valor do tempo normalizado só seria diferente caso uma das máquinas (ao menos) possuísse *csi* diferente das demais. O PyEnv com PyEnv-virtualenv foi escolhido por permitir diferentes versões de Python e diferentes ambientes Python na mesma máquina permitindo a mudança de ambiente com apenas um comando simples. Apesar de haver Anaconda entre os módulos disponíveis no LoboC este vai muito além do desejado e as dependências baixadas e instaladas com *conda* vem dos servidores do Anaconda e não do servidor oficial PyPi. Por isto, os pacotes instalados com *conda* podem não estar atualizados ou completos. Cada aplicação listada representa um passo no modelo de custos da *ProvDeploy*. O conjunto de ações associado a cada passo com seus tempos de instalação estimados em segundos entre parênteses é listado abaixo:

- PyEnv :
  - clonar o repositório ($\approx$ 150s)
  - definir (2) variáveis de ambiente ($\approx$ 10s)
  - adicionar os comandos de inicialização ao *shell script*($\approx$ 30s)
  - reiniciar o *shell* ($\approx$ 5s)
- PyEnv-virtualenv :
  - clonar o repositório ($\approx$ 150s)
  - adicionar os comandos de inicialização ao *shell script* ($\approx$ 30s)

- reiniciar o *shell* (≈ 5s)

- Python 3.7 :
  - baixar a versão Python com Pyenv (≈ 180s)
  - criar o ambiente Python com virtualenv (≈ 5s)

- dfa-lib-py:
  - carregar o ambiente Python criado no passo anterior (≈ 5s)
  - instalar coverage(≈ 30s),
  - instalar pytest (≈ 30s)
  - executar script de instalação da dfa-lib-py(≈ 60s)

- DfAnalyzer:
  - baixar (≈ 30s);
  - ativar módulo Java (≈ 5s);
  - modificar dfa.properties (≈ 30s).

- MonetDB:
  - definir (6) variáveis de ambiente (≈ 30s);
  - criar o diretório do MonetDB (≈ 5s);
  - baixar com *wget*(≈ 300s);
  - descompactar (≈ 120s);
  - criar um diretório para compilação (≈ 5s);
  - ativar modulo gcc(≈ 5s);
  - executar o *./configure* (≈ 120s);
  - executar *make*(≈ 300s);
  - verificar erros (≈ 10s);
  - executar *make install* (≈ 300s);
  - adicionar os caminhos do MonetDB às variáveis de ambiente no arquivo de bash do usuário (≈ 30s).

- fastbit:
  - wget no arquivo (≈ 180s)
  - executar *./configure* (≈ 600s)

- executar *make* ($\approx$ 600s)
- verificar a instalação com *make test* ($\approx$ 180s)

O tempo gasto em cada passo foi determinado a partir dos tempos estimados na documentação dos respectivos *softwares* e em estimativas feitas a partir de experiência prévia, o tempo em segundos de execução de cada passo é dado pelo conjunto $C(D) = \{195, 185, 185, 125, 65, 895, 1560\}$, o custo $C(D)$ de cumprir essas dependências convertido em minutos é 53,5 minutos.

Para executar esta mesma aplicação com coleta de proveniência com uso da *ProvDeploy* o conjunto de *passos* para execução é dado pelo seguinte conjunto de dependências $D_{ProvDeploy}$={ *PyEnv, Pyenv-Virtualenv, Python 2.7+, Paramiko, Singularity, Contêiner de execução*}. O Singularity é uma dependência já instalada no Loboc e por isso, esta dependência não é contabilizada no custo total. O Paramiko é uma biblioteca Python, utilizada pela *ProvDeploy* para realizar conexões remotas e assim realizar transferência de arquivos caso o usuário queira realizar uma execução remota. Nesse conjunto de passos o tempo de compilação do arquivo de definição das imagens de contêineres para execução da ferramenta de coleta de proveniência é desconsiderado, por que esses já são disponibilizados pela *ProvDeploy*, bastando a imagem de contêiner para execução do *Wordcount*. Para isso é possível apenas reutilizar alguma imagem Python encontrada em registros públicos oficiais e adicionar via arquivo de definição a *dfa-lib-py*. O conjunto de ações associado de cada passo é abaixo descrito por:

- PyEnv ($\approx$ 195s)
- PyEnv-virtualenv ($\approx$ 185s)
- Python 2.7+:
  - baixar a versão python com Pyenv ($\approx$ 180s);
  - criar o ambiente Python com virtualenv ($\approx$ 5s);
  - instalar Paramiko($\approx$ 30s).
- Contêiner de execução:
  - selecionar imagem de registro público ($\approx$ 60s);
  - escrever arquivo de definição ($\approx$ 120s);
  - compilar arquivo de definição($\approx$ 310s)
  - adicionar ao catálogo da *ProvDeploy*($\approx$ 5s)

O custo estimado para compilar o arquivo de definição da imagem do contêiner para a aplicação do usuário, foi considerado como sendo igual ao acumulado dos tempos de cada dependência presente no contêiner. Nesse caso, por exemplo, na compilação do arquivo de definição são considerados cumulativamente os custos de instalar a versão do Python e a *dfa-lib-py*. O tempo em segundos de execução de cada passo é dado pelo conjunto $C(D_{ProvDeploy}) = \{195, 185, 215, 495\}$ e o custo $C(D_{ProvDeploy})$ de cumprir essas dependências convertido em minutos é 18,1 minutos. Observa-se uma redução significativa do tempo necessário para a preparação e configuração do ambiente de execução, caindo de 53,5 minutos para 18,1 minutos.

As etapas de instalação a partir da compilação de códigos fonte e de arquivos de configuração foram considerados como atividades atômicas no contexto desta dissertação.

4.2 Estudo de caso: AlexNet

A Alexnet [78] é uma rede neural convolucional voltada para reconhecimento de imagens. Projetada por Alex Krizhevsky, a AlexNet ganhou o desafio de reconhecimento visual em grande escala do ImageNet 2012. Com alto custo computacional, esta rede foi inicialmente viabilizada pelo uso de GPUs. A instrumentação desta aplicação coleta dados relativos ao treinamento da rede, a cada época são coletados valores relativos a acurácia e perda. A taxa de aprendizado utilizada por época e a alteração deste valor ao longo da execução, também são monitorados, a Figura 4.2 apresenta uma abstração da instrumentação desta rede. O total de épocas de cada execução foi igual a 250, pois é um valor com que esta rede retorna resultados satisfatórios. Foram utilizados quatro nós computacionais com todos os processadores, este valor foi definido a partir de experiências anteriores. O conjunto de dados de entrada utilizado foi o OxfordFlowers [79], criado a partir da coleta de imagens de diversos *sites*, consistindo de 17 categorias diferentes de flores. Foram realizadas dez execuções da Alexnet com e sem a *ProvDeploy*, nesse caso podemos observar que não houve sobrecarga adicionada à aplicação, o tempo médio de execução sem a *ProvDeploy* foi de 42,73 minutos com desvio padrão de 0,89 e com a *ProvDeploy* este tempo foi de 40,60 minutos e desvio padrão de 0,84. Nesse caso, o comportamento esperado era de uma sobrecarga significativamente menor que no caso apresentado na Seção 4.1, pois a quantidade de chamadas simultâneas ao contêiner responsável por executar a aplicação de coleta de proveniência era muito inferior, entretanto, houve queda no tempo de execução. Atribui-se essa queda no tempo de execução à instanciação prévia do *schema* do banco de dados e não haver necessidade de compilação do mesmo em contêineres [80, 81]. Além disso, as operações de leitura e escrita em contêineres Singularity são favorecidas no sistema de arquivos Lustre [71]. Adicio-

nado a esses fatores a divisão dos contêineres utilizada na *ProvDeploy* se aproxima do melhor cenário de integração de múltiplos contêineres apresentado em [63].

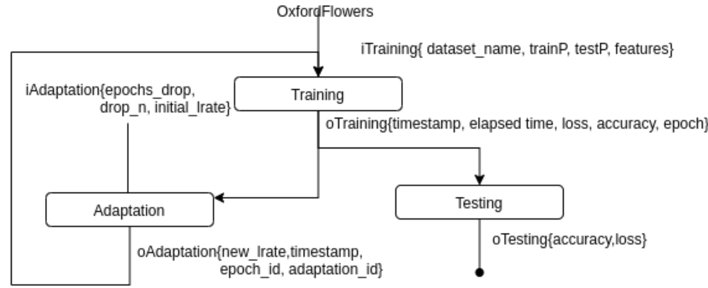

Figura 4.2: Exemplo de abstração dos dados coletados no estudo de caso da *AlexNet*

Esta versão da AlexNet foi implementada utilizando Tensorflow 1.14, o Tensorflow [82] é uma biblioteca de código aberto para aprendizado de máquina aplicável a uma ampla variedade de tarefas. É um sistema para criação e treinamento de redes neurais em larga escala. Tensorflow é um típico exemplo de conjunto de ferramentas que é difícil de ser configurado e instalado em ambientes PAD. Implementado em Python, o Tensorflow requer um grupo de bibliotecas Python muito atual. Não é de fácil suporte e manutenção, por isto muitos ambientes PAD tem disponibilizado imagens Singularity de ambientes que contêm toda a pilha de *software* do Tensorflow [2]. No caso do Tensorflow, o uso da ferramenta VirtualEnv é recomendado pela documentação oficial com objetivo de isolar os pacotes instalados pelo Tensorflow do Python do SO.

Além do Tensorflow, a implementação utilizada faz uso do módulo *numpy*, do *scikit-learn* para divisão dos conjuntos de treino e teste e da biblioteca TfLearn para otimização de desempenho. A imagem do catálogo da *ProvDeploy* que foi utilizada para execução não contêm apenas as bibliotecas do Tensorflow, mas também bibliotecas utilizadas em aprendizado de máquina para manipular, analisar e visualizar dados como Pandas e MatPlotLib, bibliotecas científicas e matemáticas como *Scipy* e *Numpy* e outras bibliotecas muito utilizadas em aprendizado de máquina como *scikit-learn*. Deste modo, caso o usuário faça modificações ao seu *script*, a mesma imagem pode ser utilizada.

Utilizando o modelo de custos, temos que o conjunto de *passos* para executar esta aplicação com coleta de proveniência é dado pelo conjunto de dependências *D={Pyenv, pyenv-virtualenv, Python 3.6, Tensorflow, scikit-learn, Numpy, dfa-lib-python, DfAnalyzer, MonetDB, FastBit}*. Utilizando Pyenv, os passos para instalação do Tensorflow, scikit-learn são agrupados em um único passo, onde a instalação do scikit-learn já engloba a Numpy. As ações de instalação do Tensorflow e do scikit-learn são demorados devido ao grande número de bibliotecas que trazem consigo. Cada aplicação listada representa um passo no modelo de custos da

ProvDeploy. O conjunto de ações e os tempos estimados associados a cada passo é abaixo descrito por:

- PyEnv ($\approx$ 195)
- PyEnv-virtualenv($\approx$ 185)
- Python 3.6 ($\approx$ 185)
- dfa-lib-py ($\approx$ 125)
- DfAnalyzer ($\approx$ 65)
- MonetDB ($\approx$ 915)
- fastbit ($\approx$ 1560s)
- demais bibliotecas :
  - instalar Tensorflow ($\approx$ 300s)
  - instalar scikit-learn ($\approx$ 240s)
  - instalar TfLearn ($\approx$ 120s)

Observando que o usuário executa os todos passos de instalação em um mesmo período de tempo, passos como instalação do Tensorflow, Numpy e scikit-learn são considerados ações atômicas pois o ambiente Python já se encontra ativado e atualizado. O tempo em segundos para execução de cada passo é dado pelo conjunto $C(D) = \{195, 185, 185, 125, 65, 915, 1560, 660\}$ e o custo $C(D)$ de cumprir essas dependências convertido em minutos é 64,8 minutos.

Para executar esta mesma aplicação com coleta de proveniência com uso da *ProvDeploy* o conjunto de *passos* e as ações associadas é igual ao apresentando na Seção 4.1, porém, o contêiner para execução da aplicação do usuário é diferente. Nesse caso, o indicado é que seja utilizada a imagem oficial do Tensorflow como imagem base, pois isto é indicado pela documentação[1]. Estimando o custo da ação de compilação das imagens contêineres como o custo acumulado das diferentes dependências instaladas no contêiner, temos que o tempo em segundos de execução de cada passo com a *ProvDeploy* é dado pelo conjunto $C(D_{ProvDeploy}) = \{195, 185, 215, 1155\}$ e o custo $C(D_{ProvDeploy})$ de cumprir essas dependências convertido em minutos é 29,1 minutos.

[1]https://www.tensorflow.org

4.3 Estudo de caso: Modelo substitutivo para quantificação de incertezas - DenseED

A DenseED é uma rede neural profunda proposta por ZHU e ZABARAS [83] como um modelo substitutivo para quantificação de incertezas e propagação em problemas de equações diferenciais parciais estocásticas. A Migração Reversa no Tempo (MRT) é um dos métodos de imageamento mais precisos na migração de dados sísmicos de modelos geológicos de grande complexidade, como variações laterais de velocidade, falhas, dobras e diapirismo [84]. Para quantificar a incerteza gerada no imageamento sísmico é necessário que várias MRTs sejam realizadas. Para evitar o custo computacional desses cálculos, a DenseED adota um modelo substitutivo em dados sísmicos [85] por meio do Tensorflow. No entanto, para auxiliar na busca pela configuração da rede mais adequada tornou necessário o uso de serviços de coleta de dados de proveniência. A instrumentação desta rede neural, por meio da *ProvDeploy*, faz a coleta de dados da estrutura da rede neural, assim como dados relativos ao treinamento por época, o fluxo de transformações de dados da DenseED pode ser visualizado na Figura 4.3. As execuções foram feitas sobre oito nós computacionais com todos os processadores, e o treinamento foi feito com 200 épocas. Esta rede neural profunda foi implementada no Tensorflow 2, versão mais nova desta biblioteca, exige a versão mais nova do pip, o instalador de pacotes do Python, e executa sobre o Python 3.7. Assim como a imagem citada na seção anterior, a imagem utilizada pela *ProvDeploy* possui bibliotecas de manipulação de dados e aprendizado de máquina visando maior reutilização. A imagem contendo Tensorflow 2 não foi utilizada no estudo da caso da Seção 4.2 por que a implementação da Alexnet utilizada faz uso da biblioteca TfLearn, que ainda não possui uma versão compatível com Tensorflow 2.+. Foram realizadas dez execuções da DenseED com e sem a *ProvDeploy*, a média dos resultados com a *ProvDeploy* foi de 129,17 minutos e o desvio padrão de 1,35, enquanto sem a *ProvDeploy* a média foi de 130,35 minutos e o desvio padrão de 0,74. Esperava-se uma sobrecarga menor que no caso apresentado na Seção 4.1, pois apesar de possuir uma quantidade maior de dados sendo coletados, o total de chamadas simultâneas ao contêiner responsável por executar a aplicação de coleta de proveniência era muito inferior, mas, assim como na Seção 4.2, não houve sobrecarga e as razões para tal são iguais às apresentadas na Seção 4.2.

Utilizando o modelo de custos, temos que os *passos* para executar esta aplicação com coleta de proveniência é dado pelo conjunto de dependências *D*={ *Pyenv, pyenv-virtualenv, Python 3.7, Numpy, Tensorflow 2, scikit-learn, Pandas, dfa-lib-python, DfAnalyzer, MonetDB, FastBit*}. Cada aplicação listada representa um passo no modelo de custos da *ProvDeploy*. O conjunto de ações associado a cada passo é

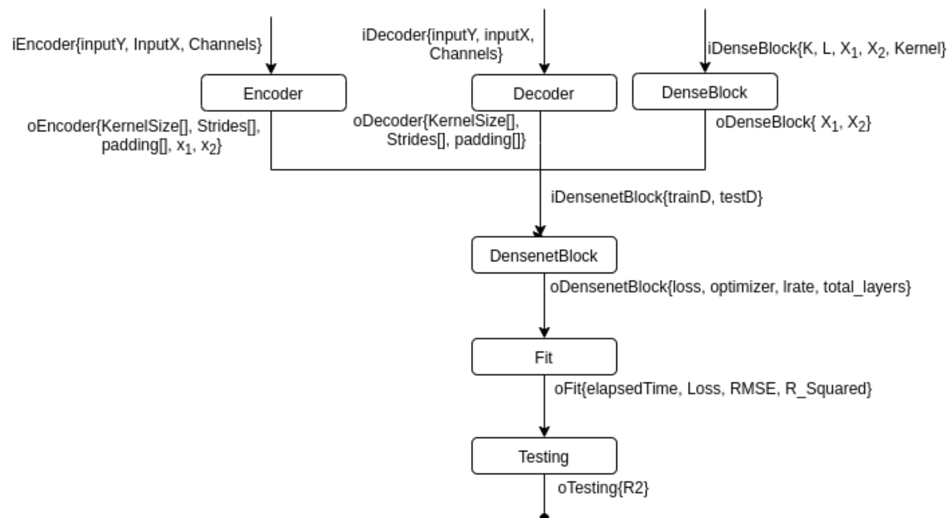

Figura 4.3: Exemplo de abstração dos dados coletados no estudo de caso da *DenseED*

abaixo descrito por:

- PyEnv ($\approx$ 195)
- PyEnv-virtualenv ($\approx$ 185)
- Python 3.7 ($\approx$ 185)
- Tensorflow 2 :
  - carregar o ambiente python criado no passo anterior ($\approx$ 5s)
  - atualizar o pip ($\approx$ 30s)
  - instalar Tensorflow 2 ($\approx$ 300s)
- dfa-lib-py ($\approx$ 120)
- DfAnalyzer ($\approx$ 65)
- MonetDB ($\approx$ 915)
- fastbit ($\approx$ 1560)
- demais bibliotecas:
  - instalar scikit-learn ($\approx$ 240s)
  - instalar Pandas ($\approx$ 120s)

O tempo em segundos para execução de cada passo é dado pelo conjunto $C(D) = \{195, 185, 185, 335, 120, 65, 915, 1560, 360\}$ e o custo $C(D)$ de cumprir essas dependências convertido em minutos é 65,7 minutos.

Assim como nas Seções 4.1 e 4.2, o conjunto de passos e ações utilizando a *ProvDeploy* é igual, diferenciando-se apenas o tempo de compilação do contêiner de execução da aplicação do usuário, que neste caso foi considerado como o tempo acumulado de instalação do Python, Tensorflow 2, dfa-lib-py, *scikit-learn* e Pandas. A estimativa de tempo gasto em cada passo é dada por $C(D_{ProvDeploy}) = \{195, 185, 215, 1090\}$ e o custo $C(D_{ProvDeploy})$ de cumprir essas dependências convertido em minutos é 33,6 minutos. Observa-se também neste terceiro estudo de caso, uma redução significativa do tempo necessário para a preparação e configuração do ambiente de execução, caindo de 65,7 minutos para 33,6 minutos.

É possível observar que o conjunto de dependências da *ProvDeploy* não varia de acordo com a pilha de *software* da aplicação e isso se deve ao uso de contêineres. Ainda, há custos que em muitos casos serão desconsiderados, a instalação do ambiente Python, por exemplo. Outra vantagem, proporcionada pelo uso de contêineres, é a possibilidade de utilizar gerenciadores de pacotes que simplificam processo de instalação de dependências. A *ProvDeploy* também diminui o total de passos para execução ao executar automaticamente a ferramenta de coleta de proveniência e verificar se todos os componentes estão funcionando conforme o esperado antes de começar a execução da aplicação do usuário.

Capítulo 5

Conclusões

Aplicações científicas se caracterizam pelo grande volume de dados gerados e por invocarem muitos componentes externos, que integram sua pilha de *software*. Para auxiliar a análise dos dados gerados, coletores de dados de proveniência são ferramentas promissoras. No entanto, ferramentas de proveniência fazem uso de muitos componentes de *software* que, ao gerar problemas na instalação e configuração, podem levar o usuário a desistir de sua adoção. Com isto, os benefícios proporcionados pela coleta de dados de proveniência como análise, reprodutibilidade e monitoramento ficam comprometidos.

Como parte da solução para incentivar a adoção de serviços de coleta de dados de proveniência foi proposta a *ProvDeploy*, para direcionar a composição da virtualização da aplicação para execução em PAD com contêineres de modo a já incorporar a ferramenta de coleta de dados de proveniência. Foi definida uma sistematização de atividades, através da LEA, a serem realizadas até a execução de experimentos através de imagens de contêineres. A *ProvDeploy* foi utilizada para apoiar estas atividades, automatizando a inicialização do serviço de coleta de dados de proveniência e auxiliando a execução de aplicações pode meio de contêineres em PAD. Finalmente, um modelo de custos foi apresentado para analisar as vantagens e desvantagens do ponto de vista de ganho de tempo e diminuição de passos para o usuário cumprir uma configuração. Foram realizados experimentos sobre diferentes estudos de caso com a *ProvDeploy* para verificar a sobrecarga adicionada pela solução. Estes evidenciaram a redução do esforço necessário para adoção da coleta de proveniência em ambientes PAD.

Como trabalhos futuros podemos citar a criação e adição de novos modelos de custo de maneira que o usuário possa escolher a partir de suas preferências. Para diminuir ainda mais o tempo de configuração, verificar o acoplamento de ferramentas de geração de arquivos de configuração de imagens, como o HPCCM [27], ao *Inicializador* da *ProvDeploy* para criar novas imagens, diminuindo o esforço do usuário na criação de imagens para a *ProvDeploy*. Além disso, seria proveitoso

alterar a *ProvDeploy* para permitir a adoção de outros serviços além de coleta de proveniência.

Referências Bibliográficas

[1] RUDE, U., WILLCOX, K., MCINNES, L. C., STERCK, H. D. "Research and education in computational science and engineering", *Siam Review*, v. 60, n. 3, pp. 707–754, 2018.

[2] VAHI, K., RYNGE, M., PAPADIMITRIOU, G., BROWN, D. A., MAYANI, R., DA SILVA, R. F., DEELMAN, E., MANDAL, A., LYONS, E., ZINK, M. "Custom execution environments with containers in pegasus-enabled scientific workflows", *arXiv preprint arXiv:1905.08204*, 2019.

[3] SILVA, V., CAMATA, J., DE OLIVEIRA, D., COUTINHO, A., VALDURIEZ, P., MATTOSO, M. "In situ data steering on sedimentation simulation with provenance data", 2016.

[4] KIRK, B. S., PETERSON, J. W., STOGNER, R. H., CAREY, G. F. "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations", *Engineering with Computers*, v. 22, n. 3-4, pp. 237–254, 2006.

[5] AYACHIT, U., BAUER, A., GEVECI, B., O'LEARY, P., MORELAND, K., FABIAN, N., MAULDIN, J. "Paraview catalyst: Enabling in situ data analysis and visualization". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pp. 25–29, 2015.

[6] FILGUEIRA, R., DA SILVA, R. F., KRAUSE, A., DEELMAN, E., ATKINSON, M. "Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science". In: *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud)*, pp. 1–8. IEEE, 2016.

[7] GERLACH, W., TANG, W., KEEGAN, K., HARRISON, T., WILKE, A., BISCHOF, J., DSOUZA, M., DEVOID, S., MURPHY-OLSON, D., DESAI, N., MEYER, F. "Skyport-container-based execution environment management for multi-cloud scientific workflows". In: *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, pp. 25–32. IEEE, 2014.

[8] BUNEMAN, P., KHANNA, S., WANG-CHIEW, T. “Why and where: A characterization of data provenance”. In: *International conference on database theory*, pp. 316–330. Springer, 2001.

[9] DAVIDSON, S. B., FREIRE, J. “Provenance and scientific workflows: challenges and opportunities”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1345–1350, 2008.

[10] SIMMHAN, Y. L., PLALE, B., GANNON, D. “A survey of data provenance in e-science”, *ACM Sigmod Record*, v. 34, n. 3, pp. 31–36, 2005.

[11] DE OLIVEIRA, D., OCAÑA, K. A., BAIÃO, F., MATTOSO, M. “A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds”, *Journal of grid Computing*, v. 10, n. 3, pp. 521–552, 2012.

[12] GUEDES, T., DE JESUS, L. A., OCAÑA, K. A. C. S., DE A. DRUMMOND, L. M., DE OLIVEIRA, D. “Provenance-based fault tolerance technique recommendation for cloud-based scientific workflows: a practical approach”, *Cluster Computing*, v. 23, n. 1, pp. 123–148, 2020. doi: 10.1007/s10586-019-02920-6. Disponível em: <https://doi.org/10.1007/s10586-019-02920-6>.

[13] DE OLIVEIRA, D., VIANA, V., OGASAWARA, E. S., OCAÑA, K. A. C. S., MATTOSO, M. “Dimensioning the virtual cluster for parallel scientific workflows in clouds”. In: Chard, K. (Ed.), *ScienceCloud’13, Proceedings of the 4th ACM HPDC Workshop on Scientific Cloud Computing, New York, NY, USA, June 17, 2013*, pp. 5–12. ACM, 2013. doi: 10.1145/2465848.2465852. Disponível em: <https://doi.org/10.1145/2465848.2465852>.

[14] DE C. COUTINHO, R., FROTA, Y., OCAÑA, K. A. C. S., DE OLIVEIRA, D., DE A. DRUMMOND, L. M. “Mirror Mirror on the Wall, How Do I Dimension My Cloud After All?” In: Antonopoulos, N., Gillam, L. (Eds.), *Cloud Computing - Principles, Systems and Applications, Second Edition*, Computer Communications and Networks, Springer, pp. 27–58, 2017. doi: 10.1007/978-3-319-54645-2_2. Disponível em: <https://doi.org/10.1007/978-3-319-54645-2_2>.

[15] GUERINE, M., STOCKINGER, M. B., ROSSETI, I., SIMONETTI, L. G., OCAÑA, K. A. C. S., PLASTINO, A., DE OLIVEIRA, D. “A provenance-based heuristic for preserving results confidentiality in cloud-based scientific workflows”, *Future Gener. Comput. Syst.*, v. 97, pp. 697–713, 2019.

doi: 10.1016/j.future.2019.01.051. Disponível em: <https://doi.org/10.1016/j.future.2019.01.051>.

[16] SILVA, V., SOUZA, R., CAMATA, J., DE OLIVEIRA, D., VALDURIEZ, P., COUTINHO, A. L., MATTOSO, M. "Capturing provenance for runtime data analysis in computational science and engineering applications". In: *International Provenance and Annotation Workshop*, pp. 183–187. Springer, 2018.

[17] CAMATA, J. J., SILVA, V., VALDURIEZ, P., MATTOSO, M., COUTINHO, A. L. "In situ visualization and data analysis for turbidity currents simulation", *Computers & Geosciences*, v. 110, pp. 23–31, 2018.

[18] PINA, D. B., NEVES, L., PAES, A., DE OLIVEIRA, D., MATTOSO, M. "Análise de Hiperparâmetros em Aplicações de Aprendizado Profundo por meio de Dados de Proveniência". In: *Anais do XXXIV Simpósio Brasileiro de Banco de Dados*, pp. 223–228. SBC, 2019.

[19] SURIARACHCHI, I., ZHOU, Q., PLALE, B. "Komadu: A capture and visualization system for scientific data provenance", *Journal of Open Research Software*, v. 3, n. 1, 2015.

[20] HUQ, M. R., APERS, P. M., WOMBACHER, A. "ProvenanceCurious: a tool to infer data provenance from scripts". In: *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 765–768, 2013.

[21] KURTZER, G. M., SOCHAT, V., BAUER, M. W. "Singularity: Scientific containers for mobility of compute", *PloS one*, v. 12, n. 5, 2017.

[22] BARBOSA, C. H., KUNSTMANN, L. N., SILVA, R. M., ALVES, C. D., SILVA, B. S., SOARES FILHO, D. M., MATTOSO, M., ROCHINHA, F. A., COUTINHO, A. L. "A workflow for seismic imaging with quantified uncertainty", *arXiv preprint arXiv:2001.06444*, 2020.

[23] MOREAU, L. "ProvToolbox—Java library to create and convert W3C PROV data model representations". 2016.

[24] MISSIER, P., BELHAJJAME, K., CHENEY, J. "The W3C PROV family of specifications for modelling provenance metadata". In: *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 773–776, 2013.

[25] PREETH, E., MULERICKAL, F. J. P., PAUL, B., SASTRI, Y. "Evaluation of Docker containers based on hardware utilization". In: *2015 International Conference on Control Communication & Computing India (ICCC)*, pp. 697–700. IEEE, 2015.

[26] MERKEL, D. "Docker: lightweight linux containers for consistent development and deployment", *Linux journal*, v. 2014, n. 239, pp. 2, 2014.

[27] MCMILLAN, S. "Making Containers Easier with HPC Container Maker". In: *In HPCSYSPROS18: HPC System Professionals Workshop, Dallas, TX*, 2018.

[28] THÖNES, J. "Microservices", *IEEE software*, v. 32, n. 1, pp. 116–116, 2015.

[29] YOUNGE, A. J., PEDRETTI, K., GRANT, R. E., BRIGHTWELL, R. "A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 74–81. IEEE, 2017.

[30] GERHARDT, L., BHIMJI, W., FASEL, M., PORTER, J., MUSTAFA, M., JACOBSEN, D., TSULAIA, V., CANON, S. "Shifter: Containers for hpc". In: *J. Phys. Conf. Ser.*, v. 898, p. 082021, 2017.

[31] PRIEDHORSKY, R., RANDLES, T. "Charliecloud: Unprivileged containers for user-defined software stacks in hpc". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, 2017.

[32] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ZAHARIA, M. "A view of cloud computing", *Communications of the ACM*, v. 53, n. 4, pp. 50–58, 2010.

[33] SABAHI, F. "Secure virtualization for cloud environment using hypervisor-based technology", *International Journal of Machine Learning and Computing*, v. 2, n. 1, pp. 39, 2012.

[34] FELTER, W., FERREIRA, A., RAJAMONY, R., RUBIO, J. "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 171–172. IEEE, 2015.

[35] REUTHER, A., MICHALEAS, P., PROUT, A., KEPNER, J. "HPC-VMs: Virtual machines in high performance computing systems". In: *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–6. IEEE, 2012.

[36] FENN, M., MURPHY, M. A., GOASGUEN, S. "A study of a KVM-based cluster for grid computing". In: *Proceedings of the 47th Annual Southeast Regional Conference*, pp. 1–6, 2009.

[37] HUANG, W., LIU, J., ABALI, B., PANDA, D. K. "A case for high performance computing with virtual machines". In: *Proceedings of the 20th annual international conference on Supercomputing*, pp. 125–134, 2006.

[38] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., DE ROSE, C. A. "Performance evaluation of container-based virtualization for high performance computing environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240. IEEE, 2013.

[39] MARTIN, J. P., KANDASAMY, A., CHANDRASEKARAN, K. "Exploring the support for high performance applications in the container runtime environment", *Human-centric Computing and Information Sciences*, v. 8, n. 1, pp. 1, 2018.

[40] KOVÁCS, Á. "Comparison of different Linux containers". In: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, pp. 47–51. IEEE, 2017.

[41] GOLDBERG, R. P. "Architecture of virtual machines". In: *Proceedings of the workshop on virtual computer systems*, pp. 74–112, 1973.

[42] MAGALHÃES, D. V., SOARES, J. M., GOMES, D. G. "Análise do impacto de migração de máquinas virtuais em ambiente computacional virtualizado", *SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS*, v. 29, pp. 235–248, 2011.

[43] JOY, A. M. "Performance comparison between linux containers and virtual machines". In: *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346. IEEE, 2015.

[44] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., HUICI, F. "My VM is Lighter (and Safer) than your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 218–233, 2017.

[45] FOX, G. C., VON LASZEWSKI, G., DIAZ, J., KEAHEY, K., FORTES, J., FIGUEIREDO, R., SMALLEN, S., SMITH, W., GRIMSHAW, A. "Futuregrid: a reconfigurable testbed for cloud, hpc, and grid computing". In: *Contemporary High Performance Computing*, Chapman and Hall/CRC, pp. 603–635, 2017.

[46] RAMAKRISHNAN, L., ZBIEGEL, P. T., CAMPBELL, S., BRADSHAW, R., CANON, R. S., COGHLAN, S., SAKREJDA, I., DESAI, N., DECLERCK, T., LIU, A. "Magellan: experiences from a science cloud". In: *Proceedings of the 2nd international workshop on Scientific cloud computing*, pp. 49–58, 2011.

[47] MAMBRETTI, J., CHEN, J., YEH, F. "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)". In: *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pp. 73–79. IEEE, 2015.

[48] BRIGHTWELL, R., OLDFIELD, R., MACCABE, A. B., BERNHOLDT, D. E. "Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R". In: *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, pp. 1–8, 2013.

[49] "Linux Containers". Disponível em: <`https://linuxcontainers.org`>.

[50] BARIK, R. K., LENKA, R. K., RAO, K. R., GHOSE, D. "Performance analysis of virtual machines and containers in cloud computing". In: *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 1204–1210. IEEE, 2016.

[51] ALSHUQAYRAN, N., ALI, N., EVANS, R. "A systematic mapping study in microservice architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51. IEEE, 2016.

[52] JAMSHIDI, P., PAHL, C., MENDONÇA, N. C., LEWIS, J., TILKOV, S. "Microservices: The journey so far and challenges ahead", *IEEE Software*, v. 35, n. 3, pp. 24–35, 2018.

[53] BERNSTEIN, D. "Containers and cloud: From lxc to docker to kubernetes", *IEEE Cloud Computing*, v. 1, n. 3, pp. 81–84, 2014.

[54] QIU, Y., LUNG, C.-H., AJILA, S., SRIVASTAVA, P. "LXC container migration in cloudlets under multipath TCP". In: *2017 IEEE 41st Annual*

Computer Software and Applications Conference (COMPSAC), v. 2, pp. 31–36. IEEE, 2017.

[55] MARTIN, A., RAPONI, S., COMBE, T., DI PIETRO, R. "Docker ecosystem: vulnerability analysis", *Computer Communications*, v. 122, pp. 30–43, 2018.

[56] "Test an insecure registry — Docker Documentation". Disponível em: <`https://docs.docker.com/registry/insecure/`>.

[57] SHU, R., GU, X., ENCK, W. "A study of security vulnerabilities on docker hub". In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 269–280, 2017.

[58] XIE, X.-L., WANG, P., WANG, Q. "The performance analysis of Docker and rkt based on Kubernetes". In: *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pp. 2137–2141. IEEE, 2017.

[59] XAVIER, M. G., DE OLIVEIRA, I. C., ROSSI, F. D., DOS PASSOS, R. D., MATTEUSSI, K. J., DE ROSE, C. A. "A performance isolation analysis of disk-intensive workloads on container-based clouds". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 253–260. IEEE, 2015.

[60] STAMATOGIANNAKIS, M., KAZMI, H., SHARIF, H., VERMEULEN, R., GEHANI, A., BOS, H., GROTH, P. "Trade-offs in automatic provenance capture". In: *International Provenance and Annotation Workshop*, pp. 29–41. Springer, 2016.

[61] SILVA, V., DE OLIVEIRA, D., VALDURIEZ, P., MATTOSO, M. "DfAnalyzer: runtime dataflow analysis of scientific applications using provenance", *Proceedings of the VLDB Endowment*, v. 11, n. 12, pp. 2082–2085, 2018.

[62] MCPHILLIPS, T., SONG, T., KOLISNIK, T., AULENBACH, S., BELHAJJAME, K., BOCINSKY, K., CAO, Y., CHIRIGATI, F., DEY, S., FREIRE, J., HUNTZINGER, D., JONES, C., DAVID, K., MISSIER, P., SCHILDHAUER, M., SCHWALM, C., WEI, Y., CHENEY, J., BIEDA, M., LUDÄSCHER, B. "YesWorkflow: a user-oriented, language-independent tool for recovering workflow information from scripts", *arXiv preprint arXiv:1502.02403*, 2015.

[63] ZHENG, C., THAIN, D. “Integrating containers into workflows: A case study using makeflow, work queue, and docker”. In: *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, pp. 31–38, 2015.

[64] DEELMAN, E., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., PATIL, S., SU, M.-H., VAHI, K., LIVNY, M. “Pegasus: Mapping scientific workflows onto the grid”. In: *European Across Grids Conference*, pp. 11–20. Springer, 2004.

[65] FILGUIERA, R., KRAUSE, A., ATKINSON, M., KLAMPANOS, I., MORENO, A. “dispel4py: A python framework for data-intensive scientific computing”, *The International Journal of High Performance Computing Applications*, v. 31, n. 4, pp. 316–334, 2017.

[66] TANG, W., WILKENING, J., DESAI, N., GERLACH, W., WILKE, A., MEYER, F. “A scalable data analysis platform for metagenomics”. In: *2013 IEEE International Conference on Big Data*, pp. 21–26. IEEE, 2013.

[67] RODRIGUEZ, M. A., BUYYA, R. “Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms”, *Future Generation Computer Systems*, v. 79, pp. 739–750, 2018.

[68] ALBRECHT, M., DONNELLY, P., BUI, P., THAIN, D. “Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids”. In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, pp. 1–13, 2012.

[69] BUI, P., RAJAN, D., ABDUL-WAHID, B., IZAGUIRRE, J., THAIN, D. “Work queue+ python: A framework for scalable scientific ensemble applications”. In: *Workshop on python for high performance and scientific computing at sc11*, 2011.

[70] GLATARD, T., DA SILVA, R. F., BOUJELBEN, N., ADALAT, R., BECK, N., RIOUX, P., ROUSSEAU, M.-E., DEELMAN, E., EVANS, A. “Boutiques: an application-sharing system based on Linux containers”, *Neuroinformatics*, 2015.

[71] “Singularity Documentation— Singularity”. Disponível em: <`https://singularity.lbl.gov/docs-flow`>.

[72] MARINHO, A., DE OLIVEIRA, D., OGASAWARA, E., SILVA, V., OCAÑA, K., MURTA, L., BRAGANHOLO, V., MATTOSO, M. “Deriving scien-

tific workflows from algebraic experiment lines: A practical approach", *Future Generation Computer Systems*, v. 68, pp. 111–127, 2017.

[73] MARINHO, A. S. *ALGEBRAIC EXPERIMENT LINE: AN APPROACH TO REPRESENT SCIENTIFIC EXPERIMENTS BASED ON WORKFLOWS*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2015.

[74] NASCIMENTO, A. P., SENA, A., BOERES, C., REBELLO, V. E. F. "On the Feasibility of Dynamically Scheduling DAG Applications on Shared Heterogeneous Systems". In: Sips, H., Epema, D., Lin, H.-X. (Eds.), *Euro-Par 2009 Parallel Processing*, pp. 191–202, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN: 978-3-642-03869-3.

[75] WU, K., AHERN, S., BETHEL, E. W., CHEN, J., CHILDS, H., CORMIER-MICHEL, E., GEDDES, C., GU, J., HAGEN, H., HAMANN, B., KOEGLER, W., LAURET, J., MEREDITH, J., MESSMER, P., OTOO, E., PEREVOZTCHIKOV, V., POSKANZER, A., PRABHAT, RÜBEL, O., SHOSHANI, A., SIM, A., STOCKINGER, K., WEBER, G., ZHANG, W.-M. "FastBit: interactively searching massive data". In: *Journal of Physics: Conference Series*, v. 180, p. 012053. IOP Publishing, 2009.

[76] PINA, D. B., NEVES, L., PAES, A., DE OLIVEIRA, D., MATTOSO, M. "Análise de Hiperparâmetros em Aplicações de Aprendizado Profundo por meio de Dados de Proveniência". In: *XXXIV Simpósio Brasileiro de Banco de Dados, SBBD 2019, Fortaleza, CE, Brazil, October 7-10, 2019*, pp. 223–228. SBC, 2019. doi: 10.5753/sbbd.2019.8827. Disponível em: <https://doi.org/10.5753/sbbd.2019.8827>.

[77] GUEDES, T., SILVA, V., MATTOSO, M., V. N. BEDO, M., DE OLIVEIRA, D. "A Practical Roadmap for Provenance Capture and Data Analysis in Spark-Based Scientific Workflows". In: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pp. 31–41, Nov 2018. doi: 10.1109/WORKS.2018.00009.

[78] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[79] NILSBACK, M.-E., ZISSERMAN, A. "A visual vocabulary for flower classification". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, v. 2, pp. 1447–1454. IEEE, 2006.

[80] SUPALOV, V. "Should You Run Your Database in Docker?" Disponível em: <https://vsupalov.com/database-in-docker/>.

[81] GADGE, A. "Is it worth deploying Database or Data Store on Containers?" Disponível em: <https://www.ashnik.com/is-it-worth-deploying-database-or-data-store-on-containers/>.

[82] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., ZHENG, X. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[83] ZHU, Y., ZABARAS, N. "Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification", *Journal of Computational Physics*, v. 366, pp. 415–447, 2018.

[84] BAYSAL, E., KOSLOFF, D. D., SHERWOOD, J. W. "Reverse time migration", *Geophysics*, v. 48, n. 11, pp. 1514–1524, 1983.

[85] FREITAS, R. S., BARBOSA, C. H., GUERRA, G. M., COUTINHO, A. L., ROCHINHA, F. A. "An encoder-decoder deep surrogate for reverse time migration in seismic imaging under uncertainty", *arXiv preprint arXiv:2006.09550*, 2020.