COPPE UFRJ
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

APOIO À IMPLANTAÇÃO DE *WORKFLOWS* CONTEINERIZADOS

Liliane Neves de Oliveira Kunstmann

Tese de Doutorado 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 Doutor em Engenharia de Sistemas e Computação.

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

Rio de Janeiro
Dezembro de 2024

APOIO À IMPLANTAÇÃO DE *WORKFLOWS* CONTEINERIZADOS

Liliane Neves de Oliveira Kunstmann

TESE 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 DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

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

Aprovada por: Prof. Marta Lima de Queirós Mattoso
Prof. Daniel Cardoso Moraes de Oliveira
Prof. Adriano Mauricio de Almeida Côrtes
Prof. Carla Osthoff Ferreira de Barros
Prof. José Maria da Silva Monteiro Filho

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2024

Kunstmann, Liliane Neves de Oliveira

Apoio à implantação de *Workflows* conteinerizados/Liliane Neves de Oliveira Kunstmann. – Rio de Janeiro: UFRJ/COPPE, 2024.

XIV, 94 p.: il.; 29, 7cm.

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

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2024.

Referências Bibliográficas: p. 83 – 94.

1. containers. 2. scientific workflows. 3. deployment. 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 sobrinhos Helena,
João(in memoriam) e Benício.

Agradecimentos

Gostaria de expressar minha profunda gratidão a Deus, autor da vida e fonte de todas as coisas.

Durante o doutorado, tive a sorte de contar com orientadores que foram verdadeiras manifestações da graça de Deus, e sou imensamente grata por eles terem se mostrado pilares nos momentos mais difíceis.

Agradeço especialmente à minha orientadora, professora Marta Mattoso, cuja dedicação ao meu crescimento acadêmico e pessoal foi fundamental. Sua orientação cuidadosa e experiência enriqueceram minha formação de formas que jamais poderia imaginar. Ao ingressar no mestrado, enfrentei desafios pela falta de experiência acadêmica, mas a professora Marta nunca hesitou em me preparar da melhor maneira possível, sempre acreditando em meu potencial e incentivando-me a ir além. Escolhi seguir para o doutorado com a certeza de que dificilmente encontraria outra pessoa com suas habilidades e comprometimento. Foram muitas prévias, revisões, correções e repetições de orientações e conselhos, e sua paciência foi interminável. Todo esse processo foi conduzido de maneira humana e responsável, qualidades que me ensinaram mais que qualquer livro ou artigo. Sinto-me muito afortunada por ter sido orientada por alguém que sempre admirei profundamente.

Estendo meus agradecimentos ao meu coorientador, professor Daniel de Oliveira, por suas valiosas contribuições, ideias e pelo bom humor que sempre trouxe às nossas interações. Obrigada por "não deixar a peteca cair"(risos). Mesmo com muitos alunos e uma rotina ocupada, você sempre encontrou tempo para ajudar. Sua amizade e apoio tornaram essa jornada muito mais leve e enriquecedora. Sou especialmente grata também pelas oportunidades que tive de colaborar com outros alunos, revisar artigos e participar de bancas. Essas experiências mostraram-me o quanto essas atividades podem ser prazerosas e recompensadoras.

Agradeço ao meu marido, Ronaldo, o amor da minha vida, meu maior apoiador e aliado nas lutas diárias. Reconheço que acompanhar meu percurso não foi fácil, mas sou grata pela sua disposição em facilitar meu caminho de todas as formas possíveis e pela paciência nos momentos de estresse. Você sempre foi meu porto seguro, cuidando de mim quando mais precisei.

Sou grata à Débora Pina pela amizade e parceria ao longo dos anos, pelas revisões

de textos, por sua disposição em ajudar, por sempre me incentivar a me aprimorar e pelas valiosas trocas de ideias.

Agradeço à família Kunstmann, especialmente aos meus sogros, Ronaldo e Lídia, por me adotarem como filha e por estarem sempre prontos a me apoiar e oferecer um lugar de descanso.

Sou grata aos meus pais, Eliane e Aelson, por todo o incentivo e apoio ao longo da vida e na minha trajetória até a graduação.

Aos meus tios Renato, Izaias, Marceli, Ageu e Lília, e aos meus primos Victória, João e Antônio, agradeço o carinho constante.

Agradeço também aos meus amigos e irmãos em Cristo: Sandra, Francisco, Eniliane, Délis, Fernanda, Fábio e aos reverendos Vanderlei e João, por acreditarem, orarem e torcerem por mim mesmo quando eu ainda tinha dúvidas.

Um agradecimento especial aos meus amigos da UFRRJ, os Morcegos, que trazem leveza à minha jornada com conversas divertidas e sempre cheias de bom humor. Sinto que o riso é mais longo ao dividir com vocês. Em especial, agradeço ao Ricardo Luiz, que me estendeu a mão quando pensei em desistir do mestrado. Sua generosidade ao compartilhar conhecimento me inspirou e me fez acreditar que eu poderia ter êxito no mundo acadêmico.

Agradeço ao Lyncoln Sousa pela parceria, que se transformou em uma amizade forte, pela paciência ao me ensinar estatística e por estar sempre disposto a escrever um código para manipular as *strings* mais inusitadas (risos).

Também agradeço ao Wesley Ferreira, que chegou no finalzinho, mas me ajudou de forma sem precedentes ao disponibilizar créditos na GCP.

Agradeço à Sarah Gasparini, Jamile Santos e Luiz Gustavo Dias por serem amigos em quem pude confiar para desabafar e compartilhar alegrias.

Agradeço imensamente à equipe da secretaria do PESC, especialmente ao Gutierrez e Ricardo que sempre me auxiliaram com a burocracia. Agradeço ao PESC, que, durante a pandemia, demonstrou agilidade e equilíbrio nas resoluções, proporcionando segurança aos alunos em tempos tão difíceis e incertos.

Agradeço à equipe do NACAD, especialmente ao professor Álvaro Coutinho e à Mara Prata, por todas as oportunidades e o suporte administrativo que me foram oferecidos.

Meus agradecimentos à banca, composta pelos professores Adriano Cortês, Carla Osthoff e José Maria, por disponibilizarem seu tempo para ler meu trabalho e oferecerem *insights* valiosos.

Agradeço às agências de fomento CAPES, CNPq e Faperj, pelo apoio financeiro em diferentes momentos do doutorado. Também agradeço ao LNCC pela disponibilização e suporte no uso do Santos Dumont, e ao Instituto de Inteligência Artificial pelo auxílio financeiro para a participação na Escola de Verão da ACM.

Por fim, reconheço que esta trajetória não começou em 2020 com o doutorado e que muitas pessoas contribuíram para que eu chegasse até aqui, mesmo que não sejam mencionadas nominalmente. Agradeço a todos que, de alguma forma, não hesitaram em estender a mão.

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

APOIO À IMPLANTAÇÃO DE *WORKFLOWS* CONTEINERIZADOS

Liliane Neves de Oliveira Kunstmann

Dezembro/2024

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

Programa: Engenharia de Sistemas e Computação

Os desafios associados à implantação de um *workflow* estão intrinsecamente ligados à condições específicas dos diversos ambientes de execução. Os contêineres mudaram a implantação de aplicações de software. No entanto, os contêineres foram projetados para implantar e executar aplicações de forma autônoma, isolada e independente, enquanto, em *workflows*, todos os componentes de software do *workflow* são interconectados e devem ser acomodados em uma composição de contêineres. Definir essa composição de contêineres é um desafio, pois são muitas alternativas para os componentes do *workflow*. O desempenho associado à escolha da composição varia conforme o ambiente computacional. Outro desafio está ligado à acomodação dos serviços de dados de proveniência dos *workflows* junto aos dados de proveniência dos contêineres. As soluções existentes ao apoio à implantação de *workflows* conteinerizados não oferecem ajuda para a definição da composição de contêineres e nem proveem rastros de proveniência que integram dados de contêiner e da execução do *workflow*. Esta tese apresenta um estudo aprofundado de composições de contêineres para *workflows*, que aliado a diversas análises de desempenho conduzidas em diferentes ambientes de alto desempenho (PAD), resultaram em auxílio efetivo à implantação dos *workflows*. Foi também desenvolvido um modelo de dados de proveniência para integrar os dados de contêiner e de *workflows* baseado em padrões de proveniência e de contêineres. Esse modelo é utilizado como base para a solução desenvolvida para coletar dados de proveniência e gerar os rastros com dados da execução integrada. Foram realizados experimentos em ambientes PAD com dois *workflows* reais, sendo um de Aprendizado de máquina e outro de Bioinformática, e um terceiro, o Montage, que é um *benchmark* de fato. Os resultados evidenciam

as vantagens de explorar diferentes composições e que o auxílio dos dados de proveniência é essencial para a qualidade e reprodução de resultados dos *workflows* conteinerizados.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

SUPPORTING THE DEPLOYMENT OF CONTAINERIZED WORKFLOWS

Liliane Neves de Oliveira Kunstmann

December/2024

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

Department: Systems Engineering and Computer Science

The challenges associated with deploying a workflow are intrinsically linked to the specific conditions of various execution environments. Containers have revolutionized the deployment of software applications. However, containers were designed to deploy and run applications autonomously, in isolation, and independently, while, in workflows, all software components of the workflow are interconnected and must be accommodated in a container composition. Defining this container composition is challenging, as there are numerous alternatives for the workflow components. The performance associated with the choice of composition varies depending on the computational environment. Another challenge is related to accommodating the provenance data services of workflows alongside the provenance data of containers. Existing solutions to support the deployment of containerized workflows neither assist in defining the container composition nor provide provenance traces that integrate container and workflow execution data. This thesis presents an in-depth study of container compositions for workflows, which, combined with various performance analyses conducted in different high-performance computing (HPC) environments, resulted in effective support for the deployment of workflows. A provenance data model was also developed to integrate container and workflow data, based on provenance and container standards. This model serves as the foundation for the solution developed to collect provenance data and generate traces with integrated execution data. Experiments were conducted in HPC environments with two real workflows: one focused on machine learning, another on bioinformatics, and a third, Montage, which is a *de facto* benchmark. The results highlight the advantages of exploring different compositions and demonstrate that the support provided by provenance data is essential for the quality and reproducibility of results in containerized workflows.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Workflows científicos demandam uma geração intensiva de dados em ambientes de processamento de alto desempenho (PAD) [20]. Esses *workflows* são compostos por atividades que representam processos computacionais, com um fluxo de dados associado. É comum que *workflows* científicos de grande escala utilizem bibliotecas para processamento paralelo, reutilizem código e executem em diferentes plataformas. Essa miríade de soluções gera um ecossistema de software complexo, e as instalações de PAD não conseguem mais acompanhar esse crescimento acelerado de programas de software e bibliotecas [93]. Como resultado, os desenvolvedores precisam executar tarefas adicionais (e muitas vezes complexas) para implantar seus *workflows*, por exemplo, instalar vários softwares a partir da compilação manual do código fonte, alterar variáveis de ambiente, instalar dependências e alterar arquivos de sistema, que são tarefas propensas a erro (devido a incompatibilidades de versão, por exemplo).

É comum que na busca de resultados satisfatórios um *workflow* científico seja executado múltiplas vezes com diferentes parametrizações em um processo de tentativa e erro, onde cada execução que pode se estender por horas ou dias. Uma vez encontrado um resultado satisfatório, é preciso assegurar que seja repetível por outros usuários e/ou em outros ambientes. Por isso, ferramentas de captura de proveniência geralmente são adotadas para auxiliar análises durante a execução desses *workflows*, e assim acelerar a busca por resultados satisfatórios e também sua posterior reprodução.

A implantação é uma fase importante no ciclo de vida de *workflows* em PAD, que inclui softwares de apoio a execução, como serviços de coleta de proveniência. Nesta tese, consideramos a implantação de *workflows* como a preparação de um *workflow* para ser executado, em um ambiente de execução. Os desafios associados à implantação de um *workflow* estão intrinsecamente ligados às condições específicas do ambiente de execução. Um ambiente de execução é uma máquina física ou virtual com hardware e software (por exemplo, SO - sistema operacional), onde o *workflow*

pode ser implantado. Isso inclui computadores pessoais, instâncias na nuvem e máquinas de PAD, como supercomputadores e *clusters*. Nesta tese, o ambiente de execução alvo são máquinas PAD, que tendem a ser heterogêneas, permitindo múltiplas possibilidades de ambientes de execução que são disponibilizados durante a execução de *jobs* gerenciados por um escalonador PAD.

Um exemplo de *workflow* científico que evidencia os desafios de implantação de *workflows* é o SciPhy, que é apresentado na Figura 1.1. O SciPhy [66] é um *workflow* da Bioinformática, que recebe como entrada um grande conjunto de sequências de DNA, RNA e aminoácidos, e produz uma árvore filogenética (ou seja, uma árvore que representa as relações evolutivas entre várias espécies). O *script* do SciPhy é composto por quatro atividades principais, que são: (i) alinhamento de múltiplas sequências (MSA); (ii) conversão de formato de alinhamento; (iii) busca pelo melhor modelo evolutivo; e (iv) construção da árvore filogenética [15, 89]. Essas atividades executam, respectivamente, as seguintes bibliotecas e programas: programas de alinhamento de múltiplas sequências (permitindo ao usuário escolher entre os programas MAFFT, Kalign, ClustalW, Muscle ou ProbCons), ReadSeq, ModelGenerator e programas de geração de árvores filogenéticas (permitindo ao usuário escolha entre RAxML ou MrBayes).

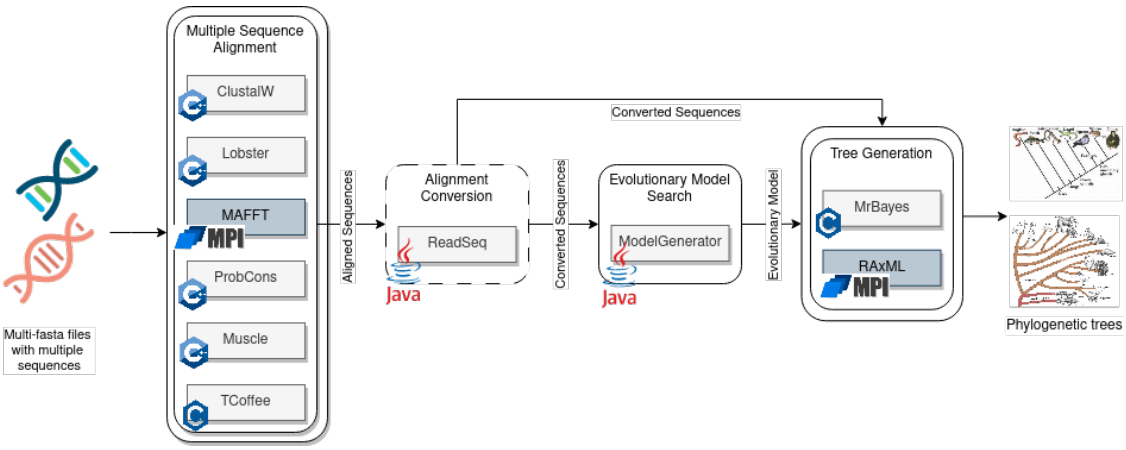


Figura 1.1: *Workflow* Sciphy com atividades e programas

A versão Python do SciPhy é um *script* que chama vários componentes de software de terceiros, como bibliotecas Java, Python 2 e módulos C/C++, integrados com MPI e outras bibliotecas para paralelização. Para implantar o SciPhy em um ambiente PAD, o usuário terá que instalar módulos C/C++ que dependem de MPI e requerem privilégios específicos. Essa instalação muitas vezes acaba sendo feita através do código-fonte das bibliotecas ou usando gerenciadores de pacotes (quando disponíveis) como o Bioconda, mas esse processo é propenso a erros, pode gerar conflitos de software ou exigir novas dependências que não são fáceis de instalar.

O SciPhy possui dependências legadas, pois usa Python 2 para a maioria de suas

atividades. Python 2 foi oficialmente descontinuado e parou de receber melhorias em 2020. Alguns programas poderiam fazer parte do SciPhy, como os métodos de alinhamento em versões mais recentes do BioPython, mas não são compatíveis com Python 2 e não é possível alternar ambientes Python no SciPhy durante a execução. Essa característica é limitante e tentar resolvê-la pode levar a conflitos de software causados pelo *downgrade* de bibliotecas. Outra característica legada do SciPhy é o uso da biblioteca ReadSeq que foi descontinuada e não possui versões substitutas, mais recentes ou atualizadas, o que tende a criar conflitos de ambiente ao longo do tempo, como a necessidade de módulos e versões de software que não estão mais disponíveis em ambientes PAD. No entanto, ReadSeq e algumas outras ferramentas de Bioinformática já estão disponíveis para serem executadas através de contêineres, ignorando o processo de instalação, mas usar contêineres no SciPhy exige alterar partes do *script* para iniciar e parar as imagens do contêiner em vez de chamar a biblioteca ReadSeq, definir volumes ou *bind paths*, e alteração do *script* do *workflow* e utilização de imagens de acordo com a política das instalações de HPC.

SciPhy é um *workflow* simples do ponto de vista computacional, mas pode ser complexo de executar devido ao processamento intensivo de dados que são consumidos e produzidos. Um experimento típico de filogenia pode analisar centenas ou milhares de arquivos contendo centenas ou milhares de sequências biológicas. Esses experimentos podem durar dias ou horas e capturar sua proveniência pode permitir ações de supervisão do usuário. Porém, o SciPhy não possui recursos nativos para monitoramento ou captura de proveniência, o que exige a adição de proveniência [32, 33, 66]. Adicionar proveniência ao SciPhy pode exigir a alteração do *script* e a adição de mais componentes à pilha de software do SciPhy para usar ferramentas de capturar proveniência como DfAnalyzer [82] e YesWorkflow [56]. Este processo pode ser bastante desafiador, considerando os problemas de implantação que este *workflow* já apresenta sem ferramentas adicionais. No entanto, a supervisão do usuário destaca-se como um exemplo de vantagem que a incorporação da captura de dados de proveniência pode oferecer ao SciPhy. Além de permitir ações de supervisão do usuário, os dados de proveniência são necessários para *workflows* em geral, para ajudar, facilitar, e apoiar a qualidade, reutilização, confiabilidade dos dados [9] e explicabilidade [67].

Os contêineres mudaram a forma como aplicações de software são implantadas e desenvolvidas. Suas principais vantagens para implantação de aplicações estão relacionadas ao isolamento, portabilidade e reutilização em diferentes ambientes computacionais. O uso de contêineres traz benefícios ao desenvolvimento de aplicações, de abordagens monolíticas até microsserviços que podem fornecer maior disponibilidade e consistência. No entanto, os contêineres foram projetados para implantar e executar programas de forma autônoma, isolada e independente, en-

quanto, em *workflows*, todos os componentes de software devem ser acomodados em contêineres para serem implantados no mesmo ambiente. Conforme destacado por KELLER TESSER e BORIN [40], os usuários continuam tendo que equilibrar necessidades de desempenho e portabilidade ao adotar contêineres para PAD. Tal dilema decorre da constatação de que a quantidade de contêineres utilizados pode influenciar o desempenho durante a execução, ao mesmo tempo que a longo prazo é possível observar benefícios no aumento da quantidade de contêineres.

O uso de contêineres para implantação de *workflows*, que chamamos de conteinerização de *workflow* para implantação, requer trabalho adicional para definir uma composição de contêiner. Nesta tese, uma composição de contêineres é definida como o mapeamento de atividades do *workflow*, softwares de apoio à execução e dados em um ou mais contêineres. Acomodar os requisitos de execução de todos os componentes do *workflow* em um único contêiner, denominado composição *coarse-grained*, não é simples e essa abordagem é de difícil manutenção e depuração. O uso de contêineres específicos para cada componente do *workflow*, denominado composição *fine-grained*, oferece flexibilidade, mas requer o gerenciamento de contêineres diferentes para a execução do *workflow*. A conteinerização de *workflows* representa um desafio significativo, especialmente ao tentar transferí-los entre diversos ambientes de PAD [13, 49, 60, 63, 96], que é essencial ao ciclo de vida de *workflows*.

Dados de proveniência têm se mostrado eficientes para aumentar a confiabilidade e interpretabilidade de resultados através de rastreabilidade [61]. No entanto, a coleta e análise de dados de proveniência do *workflow* pode sofrer alterações causadas pela composição de contêiner que é adotada. Observamos que com os atuais serviços de coleta de proveniência, a não representação dos contêineres tende a gerar traços de dados que levam o usuário a confundir a máquina hospedeira e os contêineres, tornando as análises bastante complexas a depender da quantidade de imagens de contêiner associadas ao *workflow*.

Contêineres possuem um padrão de implementação que é disponibilizado pela Open Container Initiative[1](OCI) que é uma iniciativa para padronização de formatos de *runtime* de contêiner. Diferentes tecnologias de contêineres, como Docker, Singularity e LXC, são compatíveis com o formato da OCI para implementação de *runtimes*, formato de imagem e metadados de execução. Esses metadados dos contêineres ajudam a analisar o comportamento dos contêineres; no entanto, não oferecem suporte à rastreabilidade de *workflows*, pois não são "*workflow aware*", ou seja, não são capazes de representar as atividades do *workflow*.

Uma vez que um *workflow* é conteinerizado, a proveniência de contêiner se torna necessária às análises desse *workflow* junto também ao ambiente de execução. A proveniência de contêiner tem papel de complementar a proveniência do *workflow*

[1]https://opencontainers.org/

fornecendo informações sobre os contêineres disparados, as imagens utilizadas, as atividades do *workflow* associadas às imagens, as origens das imagens de contêiner e o ambiente de execução onde o *workflow* foi executado. Além disso, em PAD é comum ter que gerar uma mesma imagem de contêiner para diferentes ambientes de execução ou *engines* [10, 92] e a proveniência dos contêineres pode prover informações necessárias a essa tarefa.

1.1 Definição do problema

Dito isso, observamos que para alcançar interpretabilidade e rastreabilidade de um *workflow* conteinerizado, é essencial integrar dados de proveniência do *workflow* e dos contêineres associados. Dados de proveniência se mostram eficazes em várias áreas, como em *workflows* para reprodutibilidade [59], *blockchain* para atribuição e rastreamento [74], e ataques cibernéticos para prevenção de acessos maliciosos [1, 98]. A integração dos dados de proveniência de *workflows* à proveniência de contêineres é importante porque há características associadas aos contêineres, como *drivers* otimizados e controle de recursos, que impactam a execução do *workflow* e sem o registro desses dados, as análises posteriores dos resultados do *workflow* podem ser impossibilitadas.

Trabalhos recentes têm abordado o suporte à proveniência para contêineres, mas a maioria deles é focada em aplicações isoladas [12, 52, 53] ou baseadas em microsserviços [1, 79]. Ao assumir uma composição padrão, eles geram rastros de dados desconectados que não consideram os *workflows*.

Uma exceção é o trabalho de OLAYA *et al.* [67], que auxilia na implantação de *workflows* conteinerizados e gera metadados em contêineres Singularity para permitir a análise de *workflows* por meio de uma interface gráfica do Jupyter. Para fornecer rastreabilidade, o trabalho de OLAYA *et al.* [67] captura metadados utilizando a proveniência de cada contêiner Singularity na execução do *workflow*. No entanto, a abordagem de OLAYA *et al.* [67] é fortemente acoplada ao Singularity e, sem uma composição *fine-grained* de *workflows*, não reconhece os metadados do *workflow*. O vínculo entre a proveniência do contêiner e do *workflow* a uma composição específica pode resultar em rastros de dados que variam caso uma composição diferente seja adotada, dificultando a análise. Desta forma, o principal problema que esta tese aborda é:

- Como podemos prover suporte à captura de proveniência em diferentes composições para a implantação de *workflows* conteinerizados em larga escala?

A partir deste problema, podemos listar algumas questões específicas, que são:

- Como auxiliar a definição de composição de contêineres para *workflows*?
- Como capturar dados de proveniência em diferentes composições?
- Como consultar dados de proveniência do *workflow* junto aos dados dos contêineres?

Argumentamos que a captura de proveniência do *workflow* não deve estar vinculada ou variar de acordo com a composição de contêiner, mas sim ser capaz de representar os mesmos dados em diferentes composições e integrá-las à proveniência do *workflow*, permitindo que o usuário obtenha a mesma análise do *workflow*. Para isso, realizamos um estudo aprofundado de composições de contêineres para *workflows* com analise qualitativa da composições de contêiner. A `ProvDeploy` foi proposta na dissertação de mestrado em KUNSTMANN [46]. Esse protótipo teve como objetivo o suporte à adoção de serviços de captura proveniência em aplicações científicas. Nesta tese, a `ProvDeploy` foi remodelada e acrescida de diversos componentes para dar suporte à implantação *workflows* com múltiplas composições. A `ProvDeploy` foi utilizada para realizar os experimentos que avaliam as composições de contêiner e o modelo de proveniência de contêiner.

Desenvolvemos modelo de dados de proveniência para integrar os dados de contêiner e de *workflows* em conformidade com o W3C-PROV a partir das anotações da OCI, apresentando uma solução para gerenciar dados de proveniência de *workflows* conteinerizados de forma padronizada. Ao adotar o W3C-PROV, objetivamos prover interoperabilidade e extensibilidade à proveniência coletada. Além disso, ao basearmos nosso modelo nas informações fornecidas através das anotações da OCI, somos capazes de representar dados comuns e disponíveis em diferentes *engines* de contêiner, não ficando presos a tecnologias específicas. Além disso, o modelo permite a extração de grafos de proveniência do *workflow* associado aos contêineres e atende a consultas de proveniência. Este modelo foi implementado na `ProvDeploy` e é integrado ao banco de dados de proveniência do *workflow*. Através dessa integração, podemos avaliar aspectos qualitativos e quantitativos das diferentes composições. Para essas avaliações, usamos diferentes *workflows* científicos, um da Bioinformática, um de aprendizado de máquina científico e o *benchmark* do *workflow* Montage. Nossos experimentos com a execução desses *workflows* na Nuvem e no supercomputador Santos Dumont com diferentes configurações evidenciaram a importância da liberdade de escolha de composições e como o modelo de proveniência para contêineres enriquece as análises.

1.2 Organização da tese

Esta tese está estruturada em quatro capítulos, além desta introdução. O Capítulo 2 aborda os conceitos fundamentais necessários para o desenvolvimento deste trabalho. Apresentamos as bases teóricas, a terminologia associada à tecnologia de conteinerização e os principais desafios relacionados à implantação de *workflows* científicos. Além disso, discutimos os trabalhos relacionados, destacando como cada abordagem trata os desafios de conteinerização identificados.

No Capítulo 3, realizamos uma análise qualitativa das composições de contêiner, destacando as vantagens associadas a cada tipo de composição. Também introduzimos a ferramenta `ProvDeploy`, que possibilita a execução dos *workflows* com diferentes composições, e apresentamos o modelo de dados de proveniência de *workflows* para contêineres. Esse modelo substitui o antigo modelo de dados da `ProvDeploy` com o objetivo de fornecer uma coleta consistente de proveniência de *workflows* para diferentes composições de contêiner.

O Capítulo 4 apresenta os experimentos realizados com *workflows* reais, evidenciando a relevância de permitir e avaliar diferentes composições de contêiner. Além disso, apresentamos o modelo de dados em ação, destacando como a proveniência de contêineres enriquece as análises e facilita a interpretação dos resultados.

Por fim, o Capítulo 5 traz as conclusões do trabalho, apontando as principais contribuições desta tese e sugerindo direções para trabalhos futuros.

Capítulo 2

Referencial teórico e trabalhos relacionados

Neste capítulo, introduzimos conceitos fundamentais a contêineres, proveniência e implantação de *workflows* científicos e seus desafios.

2.1 A implantação de *workflows* científicos

Workflows científicos geralmente são escritos na forma de *scripts* fazendo chamadas a atividades que são compostas de múltiplos programas e dados com uma configuração que objetiva solucionar problemas complexos. Cada um dos programas que é utilizado tem seus próprios requisitos para implantação e execução [73] que precisam ser atendidos para que todas as atividades possam ser executadas no mesmo ambiente. Devido a esses requisitos, um *workflow* pode ser executado com êxito no ambiente de execução sobre o qual foi projetado e falhar quando transferido para outro ambiente. Isso pode se causado por privilégios do usuário durante a execução ou pelos programas envolvidos serem limitados a um grupo de sistemas operacionais (SO) nativos [29], que chamamos de problemas de implantação. Abordar problemas de implantação é essencial para a implantação de *workflows* científicos. Uma das principais preocupações é que durante a implantação, as dependências de diferentes programas possam afetar ou conflitar entre si. Para representar os principais problemas de implantação, esta tese propõe duas categorias se são: *conflitos de ambientes* e *gerenciamento de dados*. Tais categorias são descritas abaixo.

2.1.1 Conflitos de ambiente

Conflitos de ambiente ocorrem quando os requisitos para execução de um *workflow* conflitam com características do ambiente de execução ou do *workflow*. Dividimos os conflitos de ambiente em três categorias *conflitos de atividade*, *conflitos de workflow*

e *conflitos de software legado*, que são detalhados a seguir.

- *Conflitos de atividade* são conflitos de ambiente que ocorrem quando o ambiente de execução não atende os requisitos das atividades do *workflow*. Um exemplo de conflito de atividade é quando uma das atividades do *workflow* requer nativamente uma versão ou distribuição específica de um sistema operacional (SO) que o ambiente de execução não possui ou permite [25, 29], ou privilégios que o usuário não possui. Isto impõe limitações a implantação da pilha de software do *workflow*.

- *Conflitos de workflow* estão relacionados às pilhas de software das diferentes atividades apresentarem requisitos conflituosos entre si, uma vez que as dependências de *workflows* científicos são inerentes às suas atividades encadeadas. Tomemos por exemplo atividades de *workflow* que demandam o uso de diferentes versões da biblioteca Numpy ou do Java. Conflitos de *workflow* podem ocorrer devido a adaptações ou alterações realizadas na pilha de *software* do *workflow* no processo implantação em um novo ambiente de execução. A maioria dos conflitos de ambiente, em instalações PAD, são resolvidos através de adaptações a pilha de software para adequação as limitações impostas pelo ambiente, utilizando versões ou distribuições de software alternativas ou compatíveis com os requisitos de software originais. Tais adaptações podem alterar os resultados. Em contextos, em que alterar a pilha de software original não é suficiente para resolver conflitos de ambientes, outras alternativas incluem instalar bibliotecas a partir de código fonte, alternativa que aumenta instabilidade [78], é propenso a erro e demorado; outra opção e o uso de múltiplos ambientes virtuais de execução, porém esta opção é limitada à algumas linguagens de programação.

- *Conflitos legados* são caracterizados por requerer o uso de dependências depreciadas. *Workflows* científicos são comumente compostos de softwares legados ou bibliotecas de software de domínio científico. Conflitos de software legado podem ter as mesmas características de conflitos de *workflow* ou conflitos de atividade. A distinção feita para conflitos de software legado enquanto categoria, se dá por que o depreciamento geralmente é resolvido através do *downgrade* de bibliotecas e assim, uso do software depreciado. Nesta tese, consideramos depreciados softwares que se encontram oficialmente em modo de manutenção, descontinuados, depreciados, abandonados ou obsoletos, de modo que não há previsão ou expectativa de receberem melhorias ou correções. Depreciação é uma característica comum em *workflows* científicos por conta da necessidade de reprodutibilidade e verificação de resultados durante o tempo. O rebaixamento de bibliotecas também pode gerar conflitos de *workflow* com atividades

restritas a versões específicas de bibliotecas que exigem um conjunto diferente de dependências. Isso também limita as opções para desenvolvimento e, ao longo do tempo, tende a gerar conflitos de atividades e de *workflow*, ou comprometer a implantação do *workflow* se o software legado não estiver disponível.

2.1.2 Gerenciamento de dados

As atividades de um *workflow* podem consumir, produzir e compartilhar, entre atividades, grandes quantidades de dados, que exigem persistência e gerenciamento de dados. O gerenciamento de dados é um problema de implantação porque mecanismos de armazenamento de dados podem ter que ser configurados para cada ambiente de execução. Isso pode causar problemas de geração de dados que podem comprometer toda a execução. O gerenciamento de dados em *workflows* pode ser categorizado em *operações de E/S* e *adição de proveniência*.

- *Operações de E/S* são caracterizadas por exigir persistência e enviar chamadas de rede que podem ser feitas por meio de *callbacks*, *logs*, arquivos ou SGBDs. Como essas aplicações podem ser executadas em vários ambientes onde o usuário tem um conjunto muito restrito de permissões, o usuário pode ter que adaptar o código para gerar arquivos ou executar experimentos.

- *Adição de proveniência* é uma categoria de gerenciamento de dados em *workflows* porque os requisitos de proveniência adicionam complexidade à implantação do *workflow*, tendo que ser cuidadosamente configurados, já que adicionar captura de dados de proveniência não é um processo semelhante a adicionar outras ferramentas a *workflows*. *Workflows* estão crescendo em complexidade usando múltiplas ferramentas de software, sistemas cada vez mais heterogêneos e métodos com transparência limitada. Métodos confiáveis para explicabilidade, monitoramento e análise nesses *workflows* são linhagem e rastreamento de dados [67]. Portanto, a captura de dados de proveniência é essencial na implantação de *workflows*, e a maioria dos *workflows* demanda ferramentas de captura que são externas ao *workflow*. Para capturar dados de proveniência, o usuário pode ser obrigado a mover o *workflow* para um sistema de proveniência/*workflow*, usar certas linguagens de programação ou integrá-lo à pilha de software. Em todos os casos, adicionar proveniência inclui abordar operações de E/S, conflitos de ambiente e requisitos de execução paralela. A captura de dados de proveniência também pode exigir a alteração do *script* do *workflow* por meio de instrumentação e a adição de bibliotecas para configurar chamadas de proveniência. Ele também requer execução paralela a todo o *workflow*, então possíveis conflitos de ambiente que componentes de proveniência podem

adicionar devem ser abordados com cada atividade do *workflow*. Serviços de proveniência também requerem enviar e receber múltiplas chamadas de múltiplas atividades do *workflow*, e persistência de dados por meio de *logs* ou SGBDs. Assim, a adição de proveniência aos *workflows* pode levar a problemas de implantação em uma escala similar à implantação de um *workflow* inteiro.

Conforme detalhado através SciPhy no Capítulo 1, implantar *workflows* científicos manualmente pode ser uma tarefa cansativa devido aos problemas de implantação mencionados anteriormente. Além disso, esses *workflows* geralmente não são portáveis, o que significa que o usuário terá que lidar com problemas de implantação para cada ambiente de execução que ele precisa para executar o *workflow*. Existem abordagens como contêineres que podem ser úteis para auxiliar na implantação de *workflows*. Por isso, conceitos associados a conteinerização e sua aplicação em *workflows* são discutidos na Seção 2.2.

2.2 Conteinerização

A tecnologia de conteinerização ganhou ampla popularidade com o lançamento do Docker em 2013. Contudo, o Docker é baseado no *Linux Containers* (LXC)[1], lançado em 2008, que foi o primeiro método de virtualização em nível de sistema operacional capaz de executar múltiplos sistemas Linux isolados em um único hospedeiro, utilizando um único *kernel* Linux. Segundo o LXC, os contêineres podem ser descritos como "um ambiente o mais próximo possível de uma instalação Linux padrão, mas sem a necessidade de um *kernel* separado".

A adoção generalizada dos contêineres revolucionou a computação em nuvem, promovendo um desenvolvimento de software mais limpo, portável e eficiente. Eles permitem a implantação rápida de aplicações e a customização de ambientes, tornando-se uma peça-chave na modernização de práticas de desenvolvimento e operações.

Um contêiner de navio é visto como um repositório para armazenamento, o que facilita o isolamento de diferentes objetos e seu transporte. Os contêineres de transporte podem parecer semelhantes, mas cada um tem suas características com base em seu conteúdo. Uma solução de contêiner de software é mais do que apenas um repositório de armazenamento. Ela ajuda a empacotar uma pilha de software para que esta se torne executável em um ambiente de execução diferente. Uma melhor analogia pode ser feita entre os serviços de uma empresa de mudanças e a tecnologia de contêineres. As empresas de mudanças fornecem serviços como fornecimento

[1] https://linuxcontainers.org/

de materiais de embalagem, serviços de embalagem de pertences, desmontagem de móveis, carga/descarga de caminhões, desembalagem e remontagem de móveis.

Intuitivamente, a conteinerização permite mover a execução computacional de um ambiente para outro. A *engine* de contêiner precisa de uma descrição da pilha de software a ser empacotada, suas dependências e os dados necessários para sua execução. Com base nessa descrição, a *engine* de contêiner empacota esses elementos. Este usa técnicas de virtualização para compor uma imagem da pilha de software para que ela possa ser desempacotada e executada no ambiente de destino. Essa imagem pode ser usada para criar novas imagens ou para executar processos isolados (conteinerizados). Uma solução de contêiner tem uma *engine* com quatro atividades básicas: empacotamento, geração de imagem, desempacotamento e montagem.

A conteinerização é baseada em três recursos principais do LXC:

Namespeces* de *kernel - quando falamos de contêineres, os *namespaces* de *kernel* são talvez a estrutura de dados mais importante, porque eles permitem contêineres como os conhecemos hoje. Os *namespaces* do *kernel* permitem que cada contêiner tenha seus pontos de montagem, interfaces de rede, identificadores de usuário, identificadores de processo, etc. [55].

chroot (*change root*) - é um mecanismo que permite a alteração do diretório raiz de um processo e todos os seus subprocessos. O *chroot* é usado para restringir o acesso do sistema de arquivos a uma única pasta que é tratada como a pasta raiz (/) pelo processo de destino e seus subprocessos. O *chroot* pode impedir que softwares não autorizados acessem arquivos fora do *chroot* e — mesmo que seja possível escapar do *chroot* — torna mais difícil para invasores obterem acesso ao sistema de arquivos completo. O *chroot* é frequentemente usado para testar software em ambientes isolados e para instalar ou reparar o sistema. Isso significa que o *chroot* não fornece nenhum isolamento de processo adicional além de alterar o diretório raiz de um processo para um diretório diferente em algum lugar no sistema de arquivos [18].

cgroups - responsável por agrupar processos em execução no contêiner. *Cgroups* permitem o gerenciamento dos recursos de um contêiner, como CPU, memória e rede. *Cgroups* não apenas rastreiam e gerenciam grupos de processos, mas também expõem métricas sobre CPU, memória e uso de bloco de E/S. *Cgroups* ou subsistemas são expostos por meio de pseudo-sistemas de arquivos. O sistema de arquivos pode ser encontrado em */sys/fs/cgroups* em distribuições Linux recentes. Nesse diretório, podemos acessar vários subsistemas nos quais podemos controlar e monitorar memória, núcleos de CPU, tempo de CPU, E/S, etc. Nesses arquivos, os recursos da *engine* de contêiner podem ser configurados para ter limites rígidos ou flexíveis. Quando limites flexíveis são configurados, o contêiner pode usar todos os recursos na máquina hospedeira. No entanto, outros parâmetros podem ser controlados, como compartilhamento de CPU que determina um peso proporcional relativo que o con-

têiner pode acessar a CPU. Limites rígidos são definidos para dar ao contêiner uma quantidade específica de recursos [3].

Os contêineres isolam componentes e têm sido bem-sucedidos na solução de problemas de implantação conectados à complexidade dos componentes envolvidos ("inferno de dependências") na execução de programas que exigem pilhas de software distintas e conflitantes. Além disso, os contêineres servem como elementos fundamentais na arquitetura de microsserviços, em que a aplicações são fragmentadas em serviços menores para facilitar e acelerar o desenvolvimento e a entrega de software.

Os contêineres ganharam popularidade significativa, especialmente para *software* na nuvem. A implantação simplificada facilitada por contêineres é igualmente atraente para *software* científico executado em instalações de PAD. As instalações de PAD fornecem um ambiente com recursos que geralmente são compartilhados por muitos usuários e devem fornecer alta vazão e estabilidade. Para manter o ambiente seguro e disponível para todos os usuários, as políticas de uso desses ambientes tendem a ser mais rigorosas e restritivas, tornando a personalização do ambiente e a configuração da pilha de software mais difíceis de serem alcançadas quando comparadas a ambientes onde o usuário possui maiores privilégios. Soluções como CharlieCloud [71], Shifter [28] e Singularity [47] fornecem contêineres PAD que oferecem portabilidade sem isolamento total, permitindo o mapeamento de bibliotecas do ambiente hospedeiro. No entanto, Docker [57], a solução de contêiner convencional, quando adotada, não é de maneira irrestrita e, embora existam soluções específicas para PAD, algumas funcionalidades que exigem privilégios maiores no sistema operacional são restritas, como a criação de imagens de contêineres.

Em PAD, os contêineres têm sido usados para vários propósitos: fornecer ambientes de software altamente customizáveis dentro das instalações de PAD [71], auxiliar na implantação de pilhas de software complexas, compartilhar resultados e software para verificação, reprodução e repetição de resultados. A sobrecarga computacional introduzida por soluções de contêiner, como o Docker, é geralmente mínima, conforme destacado por FELTER *et al.* [23]. Dentre as soluções de PAD, ferramentas como Singularity, Shifter e CharlieCloud demonstram impacto mínimo [90] ou, em alguns casos, impacto imperceptível [36] na sobrecarga adicionada. Usar contêineres como uma solução para executar *workflows* de PAD pode ser de grande ajuda, mas pode trazer alguns desafios que são apresentados na Seção 2.2.2.

2.2.1 Conceitos de conteinerização

Antes de discutir os desafios de implantar *workflows* por meio de contêineres, precisamos apresentar alguns conceitos de contêiner que são necessários para esta tese. Eles são os seguintes:

***Engine* de contêiner** - responsável por desempacotar uma imagem de contêiner para execução pelo *kernel* na forma de um processo isolado. A *engine* é a interface com a qual o usuário interage para enviar solicitações, enviar e receber imagens e, da perspectiva do usuário, é responsável por executar uma imagem de contêiner [55]. A *engine* de contêiner mais popular é o Docker. A *engine* é responsável por se comunicar com o *kernel* para iniciar a execução de processos conteinerizados, consumindo os metadados fornecidos pela imagem.

Imagem de contêiner - é um pacote de *software* compactado, autônomo e executável que inclui tudo o que é necessário para executar uma aplicação: código, executáveis, ferramentas do sistema, bibliotecas de sistema e configurações [57]. Essas características tornam as imagens de contêiner uma opção muito mais rápida de usar do que os processos estruturados de desenvolvimento e implantação, que dependem da replicação de ambientes de teste tradicionais.

Repositório de contêiner - esse conceito é muito semelhante ao conceito de imagens de contêiner, mas os repositórios agrupam camadas de imagens de contêiner e metadados em um arquivo chamado *manifest.json* [55]. Quando uma imagem de contêiner é criada, extraída ou enviada, ela é armazenada em um repositório de contêiner que é gerenciado pela *engine* de contêiner. Na prática, os repositórios de contêiner Linux são portáveis e consistentes durante toda a migração entre os ambientes de desenvolvimento, teste e produção. Todos os arquivos necessários para executá-los são disponibilizados por meio de uma imagem de contêiner que está contida em um repositório.

Servidor de registro - um servidor de registro de imagem é responsável por armazenar e distribuir imagens de contêiner e repositórios. Normalmente, o servidor de registro é especificado como um nome DNS normal e, opcionalmente, um número de porta para conectar na *engine* de contêiner. Grande parte do valor do ecossistema Docker e da conteinerização vem da capacidade de enviar e receber repositórios de servidores de registro, permitindo o compartilhamento e a reutilização de imagens de contêiner. Docker Hub, Binder e Nvidia GPU Cloud (NGC) são exemplos de servidores de registro públicos.

***Runtime* de contêiner** - componente de baixo nível usado pela *engine* de contêiner. Uma *runtime* de contêiner é responsável por se comunicar com o *kernel* para iniciar a execução de processos conteinerizados, consumir os metadados fornecidos pela *engine* e configurar os *cgroups*. A implementação de referência da OCI é a *runc*. Esta é a *runtime* de contêiner mais amplamente utilizada, e foi gerada a partir do Docker V2. No entanto, há outras *runtimes* compatíveis com OCI, como *crun*, *railcar* e *katacontainers*. Docker, CRI-O e muitas outras *engines* de contêiner dependem do *runc* [55]. Usar o *runc* permite consistência na execução de contêineres, independentemente da *engine* de contêiner.

Hospedeiro de contêiner / Máquina hospedeira - ambiente que executa o processo conteinerizado, pode ser uma máquina virtual, um *cluster* HPC ou um servidor em um *data center*. Para serem executadas, as imagens (repositórios) são descarregadas do servidor de registro para o hospedeiro de contêiner e são mantidas no *cache* local.

***Workload* de contêineres** - um conjunto de processos e volumes em contêineres agrupados e agendados em conjunto devido ao compartilhamento de recursos, como sistemas de arquivos ou endereços de rede. No Kubernetes, esses grupos são chamados de *pods*. O Kubernetes é a solução mais amplamente utilizada para executar e gerenciar *workloads* de contêineres, oferecendo funcionalidades específicas para aplicações de Inteligência Artificial em larga escala [17]. Além disso, apresenta recursos como *self-healing*, dimensionamento automático e gerenciamento de dados, permitindo que os cientistas de domínio foquem no desenvolvimento de suas soluções.

Gerador de imagem de contêiner - componente da *engine* de contêiner que pode ser usado para automatizar a criação e atualização de imagens. Embora esteja presente na *engine* de contêiner, há geradores de imagens que não dependem de *engines* de contêiner específicos. O uso de geradores de imagens faz parte da orquestração de contêineres e também está presente nas plataformas do provedor de nuvem, como AWS com EC2 Image Builder ou OpenShift com S2I build e Docker build.

Camada de imagem / *Image layer* - uma imagem de contêiner é baseada e composta por uma ou mais camadas, e um conjunto de camadas de imagem conectadas, juntamente com metadados, compõe um repositório. Quando um gerador de imagens gera uma nova imagem, as diferenças são salvas como uma camada. Se o gerador está construindo uma imagem com um Dockerfile, cada diretiva no arquivo cria uma nova camada. As camadas de imagem em um repositório estão conectadas em uma relação de pai-filho. Cada camada de imagem representa mudanças entre ela e a camada pai [55]. Algumas *engines* de contêiner não possuem camadas em suas imagens de contêiner nativas ou não gerenciam camadas de imagem da mesma forma. Por exemplo, o Singularity mescla camadas de imagem para alcançar melhor compressão [13]. No entanto, esse conceito se mantém relevante relevante porque o Docker gera imagens em camadas, e uma vez que o Docker Hub é o maior registro público de imagens, imagens Docker são amplamente usadas em diferentes *engines* de contêiner.

Imagem base - Uma imagem base é uma imagem sem camada pai; essas imagens são geralmente de um sistema operacional que o usuário pode instalar em seu ambiente conteinerizado. Normalmente, uma imagem base contém uma cópia nova de um sistema operacional e pode incluir as ferramentas necessárias para instalar

pacotes/fazer atualizações na imagem ao longo do tempo como yum, rpm, apt-get, dnf, microdnf entre outros gerenciadores de pacote. Na prática, as imagens base são, em geral, produzidas e publicadas por projetos de código aberto (como Debian, Fedora ou CentOS) e empresas de software como a Red Hat [55].

Tag - Quando um gerador de imagens cria um novo repositório, ele geralmente rotula as camadas da imagem para uso. Esses rótulos são chamados de *tags* e constituem uma ferramenta para que os geradores de imagens de contêiner comuniquem às *runtimes* de contêiner quais camadas devem ser executadas. Normalmente, *tags* são usadas para designar versões de software dentro do repositório de contêineres. Isso é uma convenção apenas - na realidade, nem a OCI nem qualquer outro padrão exigem um uso específico para as *tags*, e elas podem ser utilizadas para qualquer propósito que o usuário desejar. Existe uma *tag* especial - *latest* - que geralmente aponta para a camada contendo a versão mais recente do software no repositório. Essa *tag* especial ainda aponta para uma camada de imagem e, assim como qualquer outra *tag*, também pode ser usada de maneira inadequada.

Open Container Initiative - A Open Container Initiative (OCI) é uma estrutura de governança aberta criada com o propósito de estabelecer padrões abertos para formatos e *runtimes* de contêineres na indústria. Estabelecida em junho de 2015 pela Docker e outros líderes do setor de contêineres, a OCI atualmente define três especificações: a Especificação de *Runtime* (*runtime-spec*), a Especificação de Imagem (*image-spec*) e a Especificação de Distribuição (*distribution-spec*). A Especificação de *Runtime* descreve como executar um "pacote de sistema de arquivos" que é descompactado no disco. A especificação de imagem define como criar uma imagem OCI, que geralmente será feita por uma *engine*, gerando um *manifest* da imagem, uma serialização do sistema de arquivos (camada) e uma configuração da imagem. A especificação de distribuição alcançou a versão 1.0 em maio de 2020 e foi introduzida na OCI como um esforço para padronizar a API de distribuição de imagens de contêineres.

Formato de imagem de contêiner - historicamente, cada *engine* de contêiner possuía seu próprio formato para gerar imagens, de modo que o formato da imagem identificava a *engine* à qual a imagem pertencia originalmente, como Docker, RKT e LXC. Atualmente, a maioria das *engines* de contêiner migrou seus formatos para o formato de imagem compatível ao da OCI [55].

2.2.2 Contêineres em *Workflows* científicos

Embora os contêineres tenham muitos recursos que potencialmente auxiliam a implantação de software em geral, implantar *workflows* científicos por meio de contêineres ainda é um desafio. Por exemplo, para usar contêineres em *workflows*, o

usuário precisará adaptar o *workflow* para a execução conteinerizada. Além disso, é importante observar que, embora os contêineres sejam valiosos para implantação, eles não abordam, de forma inerente, os desafios de gerenciamento de dados. Ademais, podem carecer de mecanismos de integração fundamentais para uma implantação fluida dos *workflows*. Nesta tese, quando um *workflow* tem suas atividades implantadas por meio de uma ou mais imagens de contêiner, chamamos isso de *workflow* conteinerizado. A implantação de *workflows* conteinerizados envolve diferentes desafios que categorizamos como *composição de contêineres*, *integração de contêineres*, *awareness de contêiner* e *reuso de contêiner*, que chamamos de desafios de conteinerização de *workflows*. Esses desafios são discutidos a seguir.

Dessa forma, o primeiro desafio para a implantação de *workflows* conteinerizados é a composição de contêineres, conforme definido no Capítulo 1. Um mesmo *workflow* pode ter diferentes composições de contêiner, cada uma influenciando aspectos como integração entre contêineres, *container awareness*, esforço necessário para resolver problemas de implantação e utilização de recursos. A composição pode variar de acordo com o número de atividades no *workflow*, as semelhanças nas pilhas de software das atividades e as imagens de contêiner disponíveis.

Uma prática comum no contexto de *workflows* científicos é executar o *workflow* inteiro em uma única imagem de contêiner, abordagem conhecida como composição *coarse-grained* [67]. Essa estratégia é vantajosa em determinados cenários, pois facilita uma execução única e simplifica a repetição de todo o *workflow*. No entanto, ela pode resultar em imagens de contêiner grandes, aumentando os tempos de transferência entre máquinas. Além disso, adicionar software ou realizar pequenas modificações nessas imagens de contêiner é desafiador, e a composição está sujeita aos mesmos conflitos de *workflow* que ocorreriam sem contêineres, uma vez que apenas isola a pilha de software do *workflow* do sistema operacional da máquina hospedeira.

Por outro lado, utilizar várias imagens para implantar as atividades do *workflow* introduz desafios adicionais na integração dos contêineres, mas simplifica a criação e geração de imagens e facilita alterações no *workflow*, economizando tempo. A adoção de uma composição em que cada atividade do *workflow* utiliza uma imagem distinta, conhecida como composição *fine-grained* [67], facilita a substituição e o reuso de imagens. No entanto, essa abordagem depende fortemente da disponibilidade de imagens.

Entre as composições *coarse* e *fine-grained*, há uma variedade de estratégias intermediárias que combinam características de ambas e exigem avaliação para determinar a melhor composição para a implantação do *workflow*. Por exemplo, uma composição potencialmente vantajosa para certos *workflows* seria utilizar imagens oficiais de registros públicos como base, modificando alguns componentes para criar

imagens personalizadas que possam ser usadas em múltiplas atividades de um mesmo *workflow*. Nesse modelo, regras poderiam ser estabelecidas para padronizar e limitar o uso das imagens, promovendo maior eficiência e consistência.

A adição de proveniência aumenta a complexidade da composição de contêineres, pois aumenta a quantidade de composições de contêiner possíveis. Ela pode ser incorporada em outras atividades ou implantada em uma imagem exclusiva ou múltiplas imagens. Em ambos os cenários, é essencial um gerenciamento eficaz para execução paralela e persistência de dados. Embora diferentes composições possam impactar o desempenho, o uso de recursos e aspectos de usabilidade, não identificamos ferramentas ou métodos para avaliar ou apoiar a composição de contêineres em *workflows*. Por isso, nesta tese reformulamos a `ProvDeploy`, que é apresentada no Capítulo 3, para atender este desafio.

Contêineres são projetados para implantar software como serviços ou programas independentes, isolados e autônomos. Eles não estão conectados entre si nem seguem uma ordem específica de execução de tarefas. Contêineres, por padrão, operam como aplicações *stateless*. Em contraste, *workflows* consistem em atividades interconectadas que seguem uma ordem específica, dependem umas das outras e geram grandes volumes de dados. Portanto, o segundo desafio de conteinerização de *workflows* é a integração de contêineres, que consiste em restringir contêineres às atividades do *workflow*. A integração requer um mecanismo que orquestre os contêineres nas atividades do *workflow*, e por orquestração, queremos dizer o seguimento de uma ordem específica de execução [87]. Para orquestrar contêineres em *workflows*, os usuários precisam iniciar e parar as imagens de contêineres, bem como estabelecer conexões de rede/*endpoints* entre contêineres. Para abordar o desafio de integração, nesta tese, adicionamos à `ProvDeploy` funcionalidades para integração de *workflows*. Para persistir dados em contêineres, também é necessário definir *volumes* ou diretórios vinculados. Realizar verificações de integridade periódicas em diferentes contêineres é necessário, pois se uma atividade falhar durante a execução do *workflow*, isso pode comprometer os resultados, e reiniciar a atividade pode não ser eficiente o suficiente para garantir que o experimento e a captura de dados sejam bem-sucedidos. Quando o *workflow* possui tarefas paralelas, é preciso decidir o papel do contêiner na paralelização [95]. Essas tarefas de integração crescem e mudam conforme o *workflow*, mas a adição de proveniência é uma tarefa extra que possui requisitos diferentes de outras atividades, uma vez que está conectada à execução de todas as atividades, mas é iniciada de forma independente.

Os contêineres não são isolados como máquinas virtuais, eles dependem do sistema operacional da máquina hospedeira para executar seus processos isolados e podem ser afetados por outros contêineres, suas configurações [86] e o ambiente de execução [86, 92]. Assim, o terceiro desafio da conteinerização de *workflows* é a *awa-*

reness de contêineres, que consiste na necessidade de tornar explícito, por meio de registro, que as atividades do *workflow* são implantadas através de contêineres, principalmente para fins de reprodutibilidade e análise. Além disso, as tarefas de PAD são tipicamente executadas em lotes em filas gerenciadas por um escalonador, com o ambiente de execução acessível apenas durante a execução da tarefa. Observamos que certos *drivers* ou arquivos são utilizados exclusivamente durante a execução. No entanto, eles desempenham um papel essencial no desempenho. Portanto, para alcançar a *awareness* de contêineres ao implantar *workflows* conteinerizados, é essencial capturar dados de proveniência durante a execução, sobre o ambiente de execução, incluindo os contêineres. Essas informações também são necessárias para reproduzir *workflows* conteinerizados e explicar os resultados. Nesta tese, a proveniência contêiner *aware* inclui informações sobre as imagens de contêiner usadas, o uso e as configurações dos contêineres, como cada atividade está conectada e faz uma distinção entre os rastros do hospedeiro e do contêiner. Capturar dados de proveniência contêiner *aware* durante a execução é uma abordagem para enfrentar os desafios associados à *awareness* de contêiner e à gestão de dados de *workflows*. Como é necessária uma ferramenta externa ao *workflow* para capturar proveniência, esta tarefa acaba por requerer execução paralela, persistência de dados e baixa sobrecarga.

O quarto desafio da conteinerização de *workflows* que os usuários precisam gerenciar ao usar contêineres para implantar *workflows* é o reuso de contêineres. Em *workflows* e aplicações de PAD, é comum que os usuários criem imagens de contêineres em uma máquina local ou estação de trabalho e depois as transfiram para o ambiente de execução. Essa prática geralmente é causada por restrições na criação de imagens dentro das instalações de PAD, conforme descrito em [72]. No entanto, os contêineres dependem da compatibilidade do *kernel* para serem reutilizados em diferentes ambientes; se a imagem do contêiner tiver uma arquitetura de *kernel* diferente do ambiente de execução de destino, ela se tornará inutilizável. Em outros casos, a imagem pode ter *drivers* de GPU que incompatíveis com as GPUs no ambiente de destino, podendo funcionar, mas não como desejado. Isso é semelhante a um conflito de atividades, pois representa uma limitação para a implantação imposta pelas características do ambiente de execução. Além da compatibilidade de *kernel*, no PAD, a implantação de *workflows* conteinerizados pode ser afetada por outras limitações do ambiente. Por exemplo, as *engines* e registros de contêineres permitidos variam entre as instalações, e uma imagem usada no *workflow* pode não ser permitida em todos os ambientes de execução. Embora a maioria das *engines* de contêineres seja compatível, a conversão de imagem nem sempre é bem-sucedida e pode gerar imagens vazias, corrompidas ou inutilizáveis. Isso pode ocorrer por razões como o uso de privilégios de superusuário durante o processo de geração e/ou

o uso de arquivos estáticos. Ao usar contêineres para implantar *workflows* científicos, os usuários tem que encontrar formas de gerenciar o reuso de contêineres para continuar executando seus *workflows*.

No exemplo apresentado no Capítulo 1, o SciPhy, que possui múltiplos programas entre suas atividades, os contêineres podem resolver requisitos de software e limitações de software legados. No entanto, ele não oferece uma maneira fácil de lidar com problemas de gerenciamento de dados, como a adição de proveniência. Para conteinerizar o SciPhy, os desafios de conteinerização de *workflows* relativos à integração, composição e *awareness* de contêineres terão que ser abordados. Primeiro, o usuário poderá decidir quais imagens de contêiner ele usará para implantar e qual configuração melhor atenderá aos seus propósitos; por exemplo, o ModelGenerator, uma atividade do SciPhy, é baseado em Java e pode compartilhar a mesma imagem que uma ferramenta de proveniência, como DfAnalyzer ou Komadu, e as atividades que exigem MPI, como Mafft e RAxML, poderiam ser executadas pela mesma imagem, aproveitando o cache do contêiner. Outra composição diferente poderia ser encontrar imagens para cada um dos programas SciPhy, mas isso aumentaria as tarefas de integração e seria propenso a problemas de reuso. Em seguida, o usuário terá que lidar com a integração de contêineres, configurando-os para iniciar e parar de acordo com as atividades do *workflow*, seguindo a ordem das atividades do SciPhy. O usuário também terá que configurar formas de permitir a persistência de dados (como volumes ou diretórios vinculados) e a transferência entre contêineres, chamadas de rede para componentes de proveniência e execução paralela. Para reproduzir o experimento em outros ambientes de execução, esse usuário terá que lidar com a *awareness* de contêineres, anotando as imagens usadas, suas características, os contêineres iniciados por essas imagens, os *drivers* utilizados e quais atividades foram executadas por cada contêiner.

2.3 Proveniência

Devido à natureza exploratória dos *softwares* científicos, as configurações de *workflows* geralmente precisam ser ajustadas pelo cientista, e o *workflow* é executado várias vezes para encontrar a melhor configuração para um resultado satisfatório. A execução desses *workflows* pode durar horas ou dias [84], frequentemente exigindo Processamento de Alto Desempenho (PAD). A captura de dados de proveniência pode apoiar a tomada de decisão por meio do monitoramento. Quando os dados de proveniência são capturados durante a execução do *workflow* científico, suas representações como linhagem de metadados podem ajudar a entender a execução do *workflow* e sua comparação com execuções anteriores. Assim, a captura de dados de proveniência pode beneficiar essa execução exploratória do *workflow*, auxiliando

o cientista na análise e direcionamento das atividades e dados do *workflow* por meio de um armazenamento de proveniência [83].

O *World Wide Web Consortium* (W3C) definiu o PROV[2] como uma série de documentos sobre representação, gerenciamento e consultas de dados de proveniência; trata-se de uma recomendação que se tornou um padrão *de facto*. O W3C PROV define proveniência como:

> *"... informação sobre entidades, atividades e pessoas envolvidas na produção de um dado ou coisa, que pode ser usada para formar avaliações sobre sua qualidade, confiabilidade ou confiança."*

A Ontologia PROV do W3C (PROV-O) é uma representação do Modelo de Dados PROV (PROV-DM) [5]. A Figura 2.1 ilustra os elementos centrais do PROV-DM e suas relações, representadas por setas, seguindo as especificações do PROV. O PROV-DM fornece uma representação abstrata de relacionamentos que permite derivações de dados de proveniência. Uma característica importante do PROV é ter esses conceitos centrais simples e suas relações, sendo independente de domínio para que possam ser especializados, por exemplo, para rastrear artefatos de *workflows* conteinerizados. Seguir esses relacionamentos permite rastrear a derivação de entidades como as imagens de contêiner.

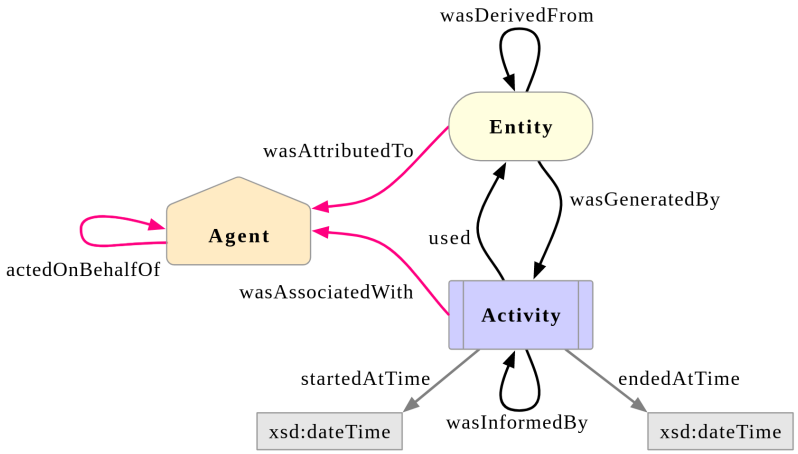


Figura 2.1: PROV-DM: O Modelo de dados do W3C PROV, de BELHAJJAME *et al.* [5].

Os elementos centrais do W3C PROV são: *(i) Agente:* refere-se à responsabilidade, associando humanos ou plataformas responsáveis pela geração de entidades e por atividades realizadas; *(ii) Atividade:* refere-se a uma transformação de dados (*ex.,* ModelGenerator) e ao momento em que foram criadas, usadas ou encerradas; e *(iii) Entidade:* refere-se a objetos de dados (*ex.,* arquivos multi-fasta, árvores filogenéticas, parâmetros, etc.). Para melhorar a conformidade com os padrões, é possível

[2]https://www.w3.org/TR/prov-overview/

especializar o PROV-DM [58] para o domínio da conteinerização, começando pelo uso de projetos públicos como o OCI.

A *Open Container Initiative* (OCI) possui um conjunto de anotações padrão que representam especificações gerais de *runtimes* e imagens de contêineres. No entanto, até onde sabemos, apesar dos dados disponibilizados pela OCI, ainda não existe uma representação de dados de proveniência de *workflows* com *awareness* de contêiner.

2.4 Trabalhos relacionados

Os diferentes desafios de conteinerização de *workflows* podem exigir muito esforço do usuário e, assim como a implantação de *workflows* sem contêineres, esses desafios podem variar de acordo com o ambiente de execução. Nesta seção, discutiremos brevemente as abordagens para os desafios de conteinerização de *workflows* de composição, integração, *awareness* e reuso.

2.4.1 Composição de contêiner

Não encontramos abordagens para implantação de múltiplas composições contêineres de *workflow*, nem que permitissem a implantação de composições híbridas. Portanto, esta seção está organizada em soluções que adotam a composição *coarse-grained* e aquelas que favorecem a composição *fine-grained*.

Abordagens *Coarse-grained*

Em PAD, é comum ter imagens de contêiner que representam todo o *workflow*, como adotado em ELISSEEV *et al.* [21] e ENGLBRECHT *et al.* [22], abordagem que também é chamada de *coarse-grained* ou monolítica. Ter um único contêiner para todo o *workflow* facilita o compartilhamento de resultados, a reexecução e a implantação.

SciUnits [52] é uma ferramenta de conteinerização para repetição e reutilização de *software*. SciUnits oferece uma estrutura leve e fácil de usar para criar e compartilhar execuções de *workflows* científicos baseadas em um único contêiner. SciUnits gera um RO (*Research Object* - Objeto de Pesquisa [4]) reutilizável , que é uma imagem Docker criada a partir das chamadas de função invocadas durante a execução original do *workflow*. Esse RO inclui os dados consumidos e produzidos pela aplicação de software, documentação e dados de proveniência, para que possa ser utilizado na repetição da execução do *workflow* em diferentes ambientes. O SciUnits é utilizado no CHEX [53], uma ferramenta focada na eficiência de reexecução de aplicações, que gerencia o problema de reexecução de múltiplas versões com contêineres.

CHOI *et al.* [13] exploram dez cenários em que contêineres e outras ferramentas de reprodutibilidade (GNU Make, Conda Env, Jupyter Notebooks, Docker, Singularity, SciUnits) são combinados para executar um *workflow* em ambientes locais e remotos. Os cenários avaliados adotam um único contêiner para todo o *workflow*. Em sua avaliação qualitativa, o SciUnits é considerado a melhor opção para portabilidade e reprodutibilidade. Eles também apresentam uma avaliação quantitativa de acordo com competência, tamanho do artefato computacional e tempo de computação. Para ambientes locais, usar o SciUnits foi a abordagem mais simples para os pontos de competência. Em ambientes remotos, utilizar Docker em conjunto com Jupyter Notebooks proporciona soluções simplificadas. SciUnits se destaca como a melhor escolha para eficiência computacional, gerenciando o tamanho da computação ao deduplicar arquivos e encapsular apenas os componentes essenciais do *workflow*. No tempo computacional, o SciUnits é o mais lento. Entre as tecnologias de conteinerização, o Docker apresentou maior peso nos requisitos de armazenamento e foi o mais difícil de usar, enquanto o Singularity se destacou devido a suas capacidades para computação paralela e técnicas de compressão. Apesar das avaliações detalhadas, esse trabalho assume uma única imagem de contêiner e não avalia diferentes composições de contêiner para *workflows*.

PROV-IO$^+$ [34] é um *framework* de rastreamento de proveniência compatível com o W3C PROV para *workflows* de aprendizado de máquina (AM) que suporta a execução de *workflows* em plataformas conteinerizadas e não conteinerizadas (PAD e nuvem). A proveniência coletada não representa dados dos contêineres, mas por seguir o W3C PROV permite extensão para tal. Esta ferramenta foca em atender diferentes necessidades de proveniência para dados científicos em sistemas de PAD, rastreados por meio de operações de E/S. Para execução conteinerizada, ela fornece uma ferramenta chamada *containerizer* que cria uma imagem de contêiner de todo o *workflow* com o PROV-IO$^+$ em uma única imagem de contêiner Docker, que é convertida para Singularity para ser executada com o PROV-IO$^+$. Os resultados apresentados no artigo sugerem que a solução não adiciona sobrecarga.

Abordagens *fine-grained*

Abordagens *fine-grained*, particularmente em *workflows* de PAD, gerenciam múltiplos contêineres, o que favorece a reutilização de componentes, tolerância a falhas, isolamento e autonomia na execução paralela [6].

O *framework* PROOF [51] destaca as vantagens de se ter múltiplos contêineres e apresenta uma abstração chamada "bloco" para modularizar o *workflow*, onde cada bloco possui seu próprio contêiner. Eles expressam preocupações em relação à possível sobrecarga adicional, mas nenhuma avaliação de desempenho foi fornecida. A abordagem Bulker [81] é um gerenciador de ambiente multi-contêiner. Ele cria e dis-

tribui *workflows* completos com componentes modulares reutilizáveis, adicionando uma camada organizacional ao gerenciamento de contêineres que permite o uso de imagens de contêineres para executar *workflows* como nativos, sem a necessidade de tornar cada atividade do *workflow* contêiner *aware*. O Dhmem [35] também adota uma abordagem *fine-grained* para a implantação de *workflows*. O Dhmem visa otimizar o compartilhamento de dados entre tarefas inter-contêiner das atividades existentes do *workflow*, diminuindo a sobrecarga de comunicação causada por múltiplos contêineres.

Para habilitar a reprodutibilidade de *workflows*, KENNEDY *et al.* [41] propõem um *plugin* para o Singularity que transforma um *workflow* monolítico em uma cadeia de contêineres *fine-grained*. O *workflow* é desacoplado em seus componentes (aplicação e dados), e cada componente é encapsulado em seu próprio contêiner com um ID único. Além disso, os componentes do *workflow* são anotados com metadados de execução. De maneira semelhante, o ContainerEnv [67] apresenta um ambiente conteinerizado *fine-grained* usando a tecnologia Singularity/Apptainer. O ContainerEnv possibilita a reprodutibilidade, rastreabilidade e explicabilidade de *workflows* conteinerizados. Eles avaliam o ContainerEnv com um *workflow* científico real, explorando diferentes composições (*coarse* e *fine-grained*) para concluir que o suporte a composição *fine-grained* é o melhor para seus objetivos de coletar metadados e proporcionar confiabilidade.

2.4.2 Integração de contêineres

Os contêineres ganharam ampla adoção em software científico para melhorar a reprodutibilidade de aplicações. Permitindo que os usuários explorem o potencial dos contêineres, alguns SWfMS (Sistemas de Gerenciadores de *Workflows* Científicos) desenvolveram soluções como Skyport [29], Pegasus Containers [91] e Asterism [25] que suportam contêineres e abordam os desafios de integração de contêineres em suas abordagens. Semelhante aos SWfMS não baseados em contêineres, essas abordagens fornecem orquestração de *workflows* por meio de contêineres; essas soluções podem reduzir significativamente a quantidade de trabalho quando comparadas à implantação manual de *workflows* conteinerizados. No entanto, elas degradam seriamente o desempenho dos sistemas de *workflow* baseados em contêineres, resultando em atraso na detecção do estado das tarefas, no acesso ao gargalo de dados compartilhados entre as tarefas, no atraso da limpeza da memória, na detecção ineficiente de recursos, bem como na baixa tolerância a falhas dos contêineres [80, 94].

Também existem soluções de integração de contêineres para a implantação de *workflows* por meio de *contêineres* baseadas em Kubernetes, como Pachyderm [65],

SciPipe [48], Argo Workflows[3] e Kubeflow [8], que não realizam a captura de proveniência. Essas soluções focam em áreas como Bioinformática, Big Data e AM e abordam o viés do Kubernetes para a execução de serviços independentes. Por conta do Kubernetes, elas oferecem gerenciamento de tarefas eficiente, recuperação automática, escalabilidade horizontal, monitoramento de recursos e outros aspectos, indo além das capacidades dos SWfMS. O Kubernetes é um sistema de código aberto para automatizar a implantação, escalabilidade e gerenciamento de software conteinerizado. A adoção desse sistema em instalações de PAD não é amplamente difundida. Quando implementado, ele requer ferramentas adicionais para facilitar a comunicação entre o Kubernetes e os escalonadores de PAD.

2.4.3 Contêiner *awareness*

A adição de proveniência ainda é um desafio em contêineres, e contêiner *awareness* é abordada por algumas abordagens. As abordagens atuais para proveniência de contêineres são limitadas a representar ações e processos de contêineres sem rastreabilidade de *workflow*. Não encontramos trabalhos relacionados que abordem a rastreabilidade da proveniência de *workflow* de forma contêiner *aware*. Capturar e relacionar dados de contêineres com a proveniência de *workflow* para análise é um problema em aberto, especialmente em PAD.

O suporte à proveniência de *workflows* não é um desafio novo [82]; entretanto, muitas soluções que alegam oferecer proveniência não representam as relações típicas que definem os caminhos de derivação para a rastreabilidade [69] ou não conseguem capturar a proveniência em PAD. PROV-IO$^+$ [34] é uma exceção, projetado para capturar a proveniência de *workflows* de AM em ambientes de PAD. Ele compartilha semelhanças com a `ProvDeploy`, como o uso de um modelo de dados extensível e compatível com o padrão W3C PROV, fornecendo relações por meio de proveniência prospectiva (*p-prov*), onde as etapas do *workflow* são representadas antes da execução, como uma receita a ser seguida, além da proveniência retrospectiva (*r-prov*), que é capturada durante a execução [26]. Apesar de oferecer uma rica proveniência de *workflows*, o PROV-IO$^+$ não é contêiner-*aware*.

Conteinerizar aplicações é um processo intuitivo, enquanto compor atividades de *workflow* em contêineres pode ser desafiador [48, 65]. A composição *coarse-grained* facilita a implantação; no entanto, substituir e reutilizar atividades de *workflow* é mais fácil com uma composição *fine-grained*. Reproduzir a execução de um *workflow* sem *awareness* de composição de contêineres pode ser desafiador. Em *workflows* de PAD, muitas vezes há restrições na geração de imagens [72]. Contêineres não são isolados como máquinas virtuais, pois dependem do sistema operacional hospedeiro

[3]https://argoproj.github.io/argo-workflows/

para executar seus processos isolados e podem ser afetados por outros contêineres, suas configurações [86] e pelo ambiente de execução [86, 92].

Todos esses problemas aumentam os desafios de conteinerização de *workflows*. Assim como os *workflows* científicos, em AM frequentemente contêineres são adotados por meio de estúdios de AM e outros serviços em nuvem que oferecem acesso a recursos de computação. No contexto de AM, os usuários enfrentam desafios nas composições de contêineres e um suporte limitado à proveniência [69, 77].

A captura de proveniência de contêineres é abordada por algumas soluções, com variações em quais dados são coletados e como são armazenados, dependendo de seus objetivos. A maioria dessas abordagens [1, 2, 12, 34, 79] coleta automaticamente metadados de contêineres de aplicações isoladas ou microsserviços. No entanto, esses metadados não relacionam artefatos da aplicação, possuem rastreamento de baixo nível, carecem de suporte a *workflows* e estão disponíveis apenas para análise *post-mortem*, ou seja, apenas após a execução. Nossa análise revela que as abordagens atuais limitam a rastreabilidade de *workflows* e a reprodutibilidade das imagens de contêiner.

Discutimos e comparamos algumas dessas abordagens com a `ProvDeploy` na Tabela 2.1. A coluna *Contêiner & Workflow* especifica se a proveniência capturada é capaz de representar a proveniência de contêineres relacionada ao *workflow*. A coluna *Nível de coleta de proveniência* detalha o nível de representação da proveniência de contêineres, indicando se representa aplicações isoladas, microsserviços ou *workflows*. *Grafo de proveniência* especifica se a abordagem permite a derivação de um grafo de proveniência. A coluna *Modelo de dados* descreve se a proveniência é representada de acordo com algum padrão, como o W3C PROV, ou de forma *ad-hoc*. *Disponibilidade de acesso* indica se os dados de proveniência estão disponíveis para análise durante a execução ou *post-mortem*. Por fim, *Suporte a consultas* indica como se dá o suporte para consultas na análise dos dados de proveniência.

CHEN *et al.* [12] discutem os desafios do rastreio de proveniência com precisão para microsserviços, propondo o `CLARION`, uma solução de rastreamento de proveniência baseada em *namespaces* que é contêiner *aware*. De forma semelhante, o `ALASTOR` permite o rastreamento de eventos suspeitos em aplicações *serverless*. O `PACED` [1] foi projetado para detectar ataques de *escape* de contêiner através do isolamento de eventos entre *namespaces*. SATAPATHY *et al.* [76] discutem a falta de captura de dados de proveniência para aplicações em microsserviços e propõem o DisProTrack [76] para capturar proveniência de microsserviços de forma integrada, lidando com chamadas paralelas inerentes aos microsserviços.

MODI *et al.* [60] discutem os desafios de capturar a proveniência de contêineres para aplicações independentes usando *namespaces* de contêiner. Eles examinam a

[4]Universal Provenance Graph

Tabela 2.1: Comparação de suporte à proveniência em *workflows* conteinerizados.

Solução de coleta	Contêiner & *Workflow*	Nível de coleta de proveniência	Grafo de proveniência	Modelo de dados	Disponibilidade do acesso	Suporte a consultas
`CLARION` [12]	Não	Microsserviço	Não	N/A	*Post-mortem*	N/A
`ALASTOR` [14]	Não	Microsserviço	Sim	*Ad-hoc*	*Post-mortem*	Grafo de proveniência
`PACED` [1]	Não	Microsserviço	Não	N/A	Execução	N/A
`DisProTrack` [76]	Não	Microsserviço	Sim	UPG[4]	*Post-mortem*	RegEx
MODI *et al.* [60]	Não	Aplicação	Sim	*Ad-hoc*	*Post-mortem*	Hipergrafo
WOFFORD *et al.* [92]	Não	Aplicação	Sim	*Ad-hoc*	Execução	SQLite
`PROV-CRT` [2]	Não	Aplicação	Sim	W3C PROV	Execução	Inteface Jupyter
ContainerEnv [67]	Sim	Workflow	Não	*Ad-hoc*	*Post-mortem*	Inteface Jupyter
`ProvDeploy`	Sim	Workflow	Sim	W3C PROV OCI	Execução	MonetDB

proveniência de contêiner *post-mortem* a partir de ferramentas de auditoria como o PROV-CRT e introduzem um modelo baseado em hipergrafos para rastrear proveniência em aplicações conteinerizadas isoladas. O modelo deles é limitado à análise *post-mortem* de aplicações isoladas.

WOFFORD *et al.* [92] propõem a definição de requisitos para capturar a proveniência de aplicações de PAD e os problemas relacionados à captura de metadados de hardware. Eles propõem o design e a implementação de um sistema de captura de proveniência baseado em contêineres, que é limitado a uma única aplicação.

`PROV-CRT` [2] é um módulo de proveniência integrado às *engines* de contêineres, como LXC e Docker. Ele rastreia e audita a proveniência durante a geração e execução do contêiner. `PROV-CRT` captura a proveniência na granularidade de chamadas de sistema, o que, embora seja difícil de analisar, possibilita a verificação e validação de cálculos durante a reprodução do contêiner, comparando os dados de proveniência auditados.

OLAYA *et al.* [67] propõem uma ferramenta que gera automaticamente um rastro de dados para cada contêiner de dados do *workflow*. Um registro do rastro é gerado dentro de cada contêiner de dados separadamente. Esta abordagem oferece uma interface Jupyter para processar esses contêineres de dados e unir os rastros em um grafo do *workflow*. Essa ferramenta não permite a geração de um grafo de proveniência independente a ser derivado por ferramentas de terceiros. Como a proveniência deles está vinculada a contêineres Singularity, eles dependem da compatibilidade de *kernel* e da disponibilidade do Singularity/Apptainer. O rastro de dados disponibilizado não é baseado em relações W3C PROV, o que obriga o usuário a aprender uma nova representação, e está disponível apenas para análise *post-mortem*, que também depende da disponibilidade dos contêineres de dados.

Nossa abordagem, implementada na `ProvDeploy` [43], está alinhada com WOF-

FORD *et al.* [92] e CANON [10], destacando a importância de documentar as características do contêiner para análise, explicabilidade e reprodutibilidade, dado que os contêineres podem empregar diferentes *drivers* para executar a mesma tarefa. A `ProvDeploy` [43, 44] é um *framework* que facilita a implantação de *workflows* científicos em ambientes PAD com captura integrada de proveniência. A `ProvDeploy` foi inicialmente desenvolvida para apoiar a captura de dados de proveniência em aplicações científicas, pois permitia o uso de serviços de proveniência por meio de contêineres. Em sua versão inicial, a `ProvDeploy` incluía um catálogo que armazenava metadados sobre as imagens de contêiner disponíveis para execução. Esse catálogo foi posteriormente expandido para fornecer informações mais detalhadas sobre as imagens de contêiner. Quando *reformulamos* a `ProvDeploy` para executar *workflows*, identificamos a necessidade de proveniência de contêiner que também registrasse o *workflow* e suas atividades. Até onde sabemos, OLAYA *et al.* [67] e `ProvDeploy` são as únicas abordagens que oferecem proveniência de contêiner no nível do *workflow*.

2.4.4 Reuso de contêiner

O reuso de contêineres, embora essencial, não é muito abordado, pois a maioria dos problemas relacionados a esse desafio ocorre em PAD. Para manter os contêineres reprodutíveis, CANON [10] propõe um conjunto de melhores práticas para a reprodutibilidade de contêineres que também promove o reuso. Da mesma forma, GRUENING *et al.* [31] apresenta uma lista de recomendações para o processo de geração de imagens de contêiner para manter os contêineres reutilizáveis, compatíveis e fáceis de integrar em *pipelines* e *workflows* de análise, mas algumas dessas práticas não são viáveis em muitos contextos, especialmente em PAD. WOFFORD *et al.* [92] destaca que, embora os usuários de PAD adotem contêineres pela mobilidade e reuso, a geração de imagens para cada ambiente PAD é uma tarefa comum de implantação, devido às imagens de contêiner se tornarem inutilizáveis. Existe também uma iniciativa, derivada da OCI, para suportar o reuso de contêineres em PAD: o *HPC Container Conformance*[5], que evidencia a urgência de definir melhores práticas para o uso de contêineres em instalações de PAD, conhecer as limitações dos contêineres e padronizar seu uso. Essa iniciativa visa definir rótulos e anotações para descrever qual ambiente de execução é esperado e o que o contêiner garante. Se uma imagem de contêiner se tornar inutilizável em ambientes específicos, essas anotações podem ser usadas para gerar uma nova imagem de contêiner com características semelhantes. WOFFORD *et al.* [92] e STRAESSER *et al.* [86] discutem a importância de registrar metadados que descrevem o ambiente de execução e os

[5]https://github.com/container-in-hpc/container-hpc-conformance

contêineres durante a execução. A maior parte desses dados é salva em arquivos e diretórios de acordo com a execução, em diferentes formatos e representações, e ainda não encontramos um grupo padrão de anotações ou representações para esses dados. Ainda assim, eles podem ser cruciais para recriar ou encontrar imagens de contêiner alternativas.

Neste capítulo apresentamos os conceitos de implantação de *workflows* e conceitos base associados a conteinerização que são utilizados ao longo desta tese. Discutimos os desafios de implantação de *workflows* conteinerizados e como as atuais abordagens se propõem a resolver os diferentes desafios. Nosso intuito foi evidenciar a falta de uma ferramenta que dê suporte à implantação de múltiplas composições de contêiner e que permita a coleta de proveniência de *workflow* e de contêiner com consistência entre as composições, para esclarecer a necessidade das mudanças aplicadas à `ProvDeploy`, que é apresentada no Capítulo 3.

Capítulo 3

Abordagem proposta

Neste capítulo, apresentamos uma análise qualitativa das composições de contêiner, uma vez que observamos que não há uma composição que seja melhor em todos os casos, apresentamos a `ProvDeploy` como forma de executar as múltiplas composições e o modelo de dados de proveniência para contêineres como forma de lidar com os os problemas já discutidos.

3.1 Análise Qualitativa das Composições de Contêiner para *Workflows*

Esta seção apresenta uma avaliação qualitativa para auxiliar na escolha de uma composição de contêiner. Existem diversas alternativas para a conteinerização de *workflows*, que vão desde composições *coarse-grained* até composições *fine-grained*, cada uma atendendo a diferentes propósitos. Classificamos as composições de contêiner para *workflows* em três grupos e as avaliamos com base em critérios qualitativos relacionados à portabilidade, implantação, compartilhamento de *workflows* e flexibilidade para reutilizar componentes de *workflows*, entre outros fatores. Esses critérios foram compilados a partir da literatura sobre contêineres e dos trabalhos relacionados discutidos anteriormente na Seção 2.4.

3.1.1 Composições de Contêiner para *Workflows*

A implantação de um *workflow* científico deve considerar como conteinerizar seus componentes. Esses componentes incluem os dados de entrada do *workflow*, as atividades do *workflow*, o gerenciamento dos dados associados a essas atividades e a captura de dados de proveniência da execução do *workflow*. Esses diversos componentes do *workflow* apresentam relações de dependência, compartilham dados e, em alguns casos, podem demandar versões diferentes de compiladores ou sistemas

operacionais conflitantes. Nesta tese, uma composição de conteinerização de *workflow* define como os componentes do *workflow* são organizados dentro de imagens de contêiner, sendo que a menor unidade de um *workflow* que pode ser encapsulada por um contêiner é uma atividade do *workflow*.

Consideramos três tipos de composição de contêiner para um *workflow* com captura de proveniência, conforme ilustrado na Figura 3.1: (i) *coarse-grained* - um único contêiner para o *workflow*, o serviço de proveniência e os serviços de gerenciamento de dados; (ii) *Híbrido* - um contêiner abrangendo todas as atividades do *workflow*, oferecendo, nesse caso, variações composicionais juntamente com os serviços de proveniência e dados; e (iii) *fine-grained* - um contêiner para cada atividade do *workflow*, um para o serviço de dados e outro para o serviço de proveniência.

Como a composição híbrida produz variações composicionais, ela pode ser subdividida em inúmeras composições, alguns exemplos que podemos citar: (i) *Modular de Proveniência* - um contêiner para todas as atividades do *workflow*, outro para o serviço de dados e outro para o serviço de proveniência; e (ii) *Modular Parcial*, conforme mostrado na Figura 3.1, com três variações. A composição modular parcial possui três variantes: #1 - um contêiner para o *workflow* com o serviço de proveniência e outro contêiner para os serviços de dados; #2 - um contêiner para o *workflow* com os serviços de dados e outro contêiner para o serviço de proveniência; e #3 - um contêiner para o *workflow* e outro contêiner para o serviço de proveniência com os serviços de dados, onde *workflow* significa que todas as atividades do *workflow* são implantadas em um único contêiner.

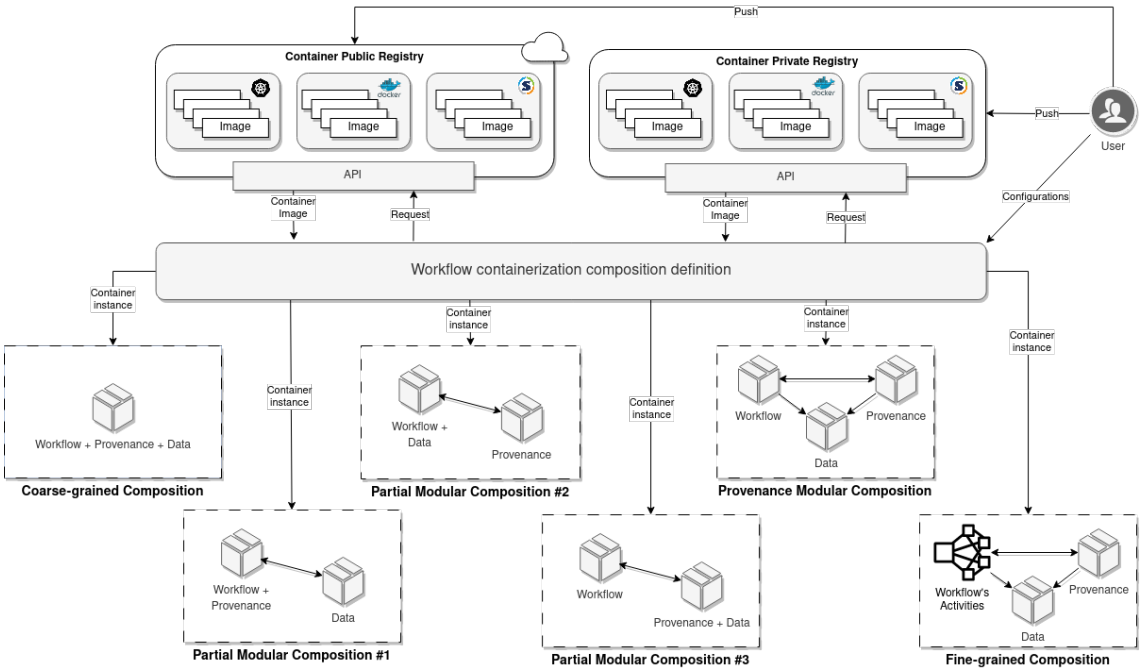


Figura 3.1: Alternativas de composições de contêineres para *workflows* com captura de proveniência.

Essas composições representam apenas um subconjunto das opções disponíveis para a conteinerização de *workflows*. A escolha de uma composição adequada não é uma decisão trivial e pode ser influenciada pela necessidade de agrupar componentes que compartilham propriedades do ambiente de execução. Esse agrupamento pode ser motivado por diversos fatores, como melhorar o desempenho ao isolar certos componentes de outros, evitar conflitos de dependências ou concentrar operações de E/S em arquivos compartilhados dentro de um contêiner distinto.

Por exemplo, pode-se optar por agrupar componentes do *workflow* que compartilham bibliotecas ou softwares comuns, como uma Máquina Virtual Java, dentro do mesmo contêiner. Alternativamente, é possível isolar operações de E/S em arquivos compartilhados alocando-as em um contêiner separado. Um desafio inerente à conteinerização de *workflows* reside na dependência de execução entre as atividades. Embora seja intuitivo agrupar atividades que geram dados para consumo pela próxima atividade, podem surgir conflitos relacionados ao ambiente de execução necessário.

As atividades de *workflows* científicos frequentemente envolvem a invocação de programas legados, e conflitos podem surgir quando um programa que exige uma distribuição específica de sistema operacional é seguido no *workflow* por outro que requer uma distribuição diferente ou quando duas atividades demandam versões distintas de uma biblioteca específica, *e.g.*, Numpy. Nesses casos, as composições híbridas e de contêineres *fine-grained* tornam-se cruciais para mitigar esses conflitos. Essas composições envolvem o isolamento de uma única imagem de componente em relação ao restante do *workflow*, facilitando uma implantação consistente e resolvendo problemas de compatibilidade.

3.1.2 Critérios Qualitativos para Conteinerização de *Workflows*

Nós avaliamos as composições de contêiner para *workflows* utilizando os critérios qualitativos apresentados na Tabela 3.1. Fornecemos um resumo dos níveis de dificuldade associados a cada composição de contêiner, levando em consideração fatores que podem influenciar o tempo de geração, a experimentação com componentes alternativos do *workflow*, o tempo de transferência da imagem e outros aspectos relevantes. A discussão abrange desafios relacionados aos níveis de dificuldade em termos de repetibilidade, compartilhamento de *workflows*, implantação, reprodutibilidade, reutilização de componentes, manutenção de *workflows*, gerenciamento de recursos, depuração e substituição. Os níveis de dificuldade dos desafios são categorizados como fácil, moderado e difícil, correspondendo ao esforço envolvido ou ao número total de ações/etapas necessárias para concluir a tarefa. A avaliação foi

realizada de forma sistemática com base em artigos como CANON [10], CHOI *et al.* [13], GRUENING *et al.* [31], SHAFFER *et al.* [79].

Tabela 3.1: Avaliação de critérios qualitativos de composições de contêiner.

Critério \ Composição	**Coarse-grained**	**Híbrida**	**Fine-grained**
Repetibilidade	fácil	moderado	difícil
Compartilhamento de *Workflow*	fácil	moderado	difícil
Implantação	fácil	moderado	difícil
Reprodutibilidade	difícil	fácil	fácil
Reutilização de Componentes	difícil	moderado	fácil
Manutenção do *Workflow*	difícil	moderado	moderado
Gestão de Recursos	difícil	moderado	fácil
Depuração	difícil	difícil	fácil
Substituição de Componentes	difícil	moderado	fácil

Repetir uma execução de *workflow* envolve executar o mesmo *workflow* com dados idênticos. O critério de *Repetibilidade* está associado ao esforço necessário para implantar o mesmo *workflow*, seja por um usuário diferente ou em um ambiente computacional diferente. Compartilhar um *workflow* e seus resultados é crucial para sua reprodutibilidade e confiabilidade. O critério de *Compartilhamento de Workflow* está relacionado ao esforço envolvido em compartilhar o *workflow* completo com outros usuários. Esses dois critérios são fundamentais para garantir que os resultados de um *workflow* possam ser replicados e validados por outros, além de garantir a acessibilidade e colaboração em ambientes de pesquisa ou desenvolvimento.

A implantação de um *workflow* pode ser agilizada por meio do uso de contêineres. O critério de *Implantação* está relacionado ao esforço necessário para implantar um *workflow* utilizando imagens de contêineres. Reproduzir os resultados de um *workflow* envolve o esforço para obter as mesmas conclusões usando dados ou implementações de algoritmos diferentes. A *Reprodutibilidade* abrange diversos aspectos, incluindo a verificação de rastros de proveniência. Esses critérios são essenciais para garantir que um *workflow* seja facilmente implantado em diferentes ambientes e que os resultados possam ser reproduzidos, possibilitando a validação dos resultados com diferentes entradas ou abordagens.

O critério de *Reutilização de Componentes* está relacionado ao uso de uma imagem de contêiner que já foi criada. O objetivo é empregar uma imagem correspondente a um componente do *workflow* sem modificações, permitindo que ela seja utilizada em um novo *workflow*/atividade. Por exemplo, reutilizar uma atividade de *workflow* com um programa específico ou utilizar um conjunto de dados com seu serviço de dados. Esse critério reduz o tempo e o esforço necessários para implantar novos *workflows*, aproveitando componentes e imagens já existentes.

A *Manutenção do Workflow* se torna necessária quando há a atualização da versão de um componente do *workflow*, do sistema operacional ou do compilador de

atividades específicas. O critério de manutenção do *workflow* está relacionado ao esforço envolvido na atualização do contêiner do *workflow* à medida que o *workflow* evolui e é transferido entre ambientes de execução (máquinas físicas). Esse critério é importante para garantir que o *workflow* continue funcional e eficiente à medida que componentes, bibliotecas ou ambientes de execução são modificados, além de permitir a adaptação do *workflow* a novos requisitos ou melhorias de desempenho.

O uso de *cgroups* possibilita a definição dos recursos aos quais um contêiner pode acessar. Além disso, tarefas podem ser executadas em paralelo ao implantar contêineres que executam a mesma atividade. Dada a flexibilidade no escalonamento e na paralelização das atividades do *workflow*, a gestão de recursos computacionais torna-se crucial para alinhar-se aos requisitos de execução do *workflow*. O critério de *Gestão de Recursos* está relacionado ao esforço envolvido no escalonamento, paralelização e gerenciamento de recursos. Esse critério é fundamental para otimizar o uso dos recursos disponíveis e garantir que o *workflow* seja executado de maneira eficiente, permitindo uma distribuição de recursos que melhor atenda a cada atividade do *workflow*. A gestão eficaz de recursos pode melhorar a escalabilidade e o desempenho do *workflow*, especialmente em ambientes distribuídos ou em larga escala.

O critério de *Depuração* está relacionado à facilidade de detectar e corrigir erros ou anomalias. Esse critério é fundamental para identificar problemas durante a execução do *workflow*, garantindo que os erros sejam rapidamente encontrados e corrigidos, o que melhora a confiabilidade do *workflow*.

O critério de *Substituição de Componentes* está associado à capacidade de tornar um componente do *workflow* facilmente localizável e substituível por uma implementação alternativa, como, por exemplo, um algoritmo diferente para a mesma atividade do *workflow*, ou uma implementação alternativa do mesmo algoritmo. Enquanto critério, consideramos que essa substituição ocorre através de imagens de contêiner alternativas. Em *workflows* científicos, é comum explorar diferentes métodos, substituindo e reutilizando atividades. Isso permite que o *workflow* se adapte a novas demandas ou que novos algoritmos sejam testados para melhorar os resultados, mantendo a flexibilidade e a capacidade de evolução do *workflow*.

3.1.3 Comparando composições de contêiner para *workflows*.

A seguir, discutimos as vantagens e desvantagens associadas a cada uma das composições de contêiner do *workflow*, conforme descrito na Tabela 3.1. Nesta seção, apresentamos cada composição utilizando o *workflow* SciPhy, detalhado na Seção 4.1.1.

Composição *coarse-grained*

Na Figura 3.2, exemplificamos a implementação de uma composição *coarse-grained* no SciPhy. A composição *coarse-grained*, como abordagem de execução em um único passo, é vantajosa para repetibilidade, compartilhamento do *workflow* e implantação, uma vez que a imagem inclui todas as informações necessárias para a execução. No entanto, essa composição pode resultar na criação de imagens de contêiner de tamanho considerável, prolongando o tempo necessário tanto para o processo de geração de imagem quanto para a transferência para diferentes ambientes. O usuário deve estar atento a essa característica, especialmente no contexto de implantação de *workflows* em larga escala.

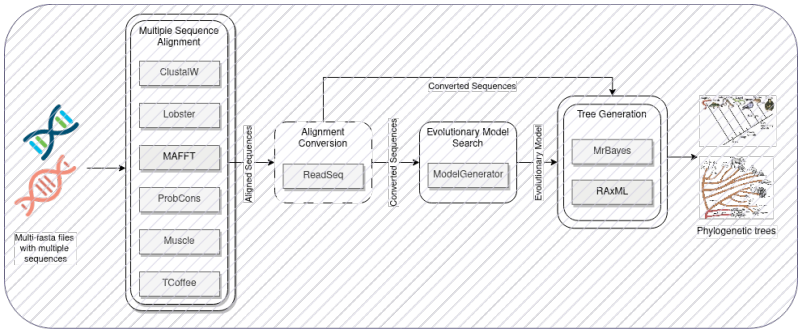


Figura 3.2: Composição *coarse-grained* aplicada ao SciPhy. A caixa hachurada representa um contêiner que encapsula todo o *workflow* e suas dependências.

A reprodutibilidade, por outro lado, encontra alguns obstáculos nessa composição devido à pilha de software altamente específica encapsulada no mesmo contêiner junto com os dados de entrada. Consequentemente, modificar qualquer elemento no contêiner gerado por essa composição é uma tarefa demorada e propensa a erros e conflitos de software. Da mesma forma, surge o desafio de reutilizar componentes, pois o uso de um componente específico do *workflow* exige a implantação de toda a imagem do contêiner. Isso ocorre devido à impossibilidade de separar componentes que foram conteinerizados de forma conjunta.

Na manutenção de componentes do *workflow*, a imagem única é propensa a conflitos de dependências de software ao longo do tempo. O uso de uma única imagem de contêiner também impõe limitações às ações de gerenciamento de recursos, já que qualquer configuração será aplicada à imagem como um todo. Consequentemente, tentar executar atividades específicas do *workflow* paralelizando os contêineres dentro da composição *coarse-grained* pode ser desafiador. Além disso, durante a execução, essa composição pode levar à iniciação de múltiplos processos pai, prática desencorajada devido ao potencial surgimento de processos zumbis ou órfãos[1]. Assim, o gestão de recursos torna-se ainda mais complexa e ineficiente nessa com-

[1]https://cloud.google.com/architecture/best-practices-for-building-containers

posição.

Pelos mesmos motivos, a depuração de erros é desafiadora em contêineres *coarse-grained*, onde avaliar o estado de funcionamento de contêineres que executam múltiplas aplicações é difícil. Além disso, corrigir erros exige a criação de uma nova imagem para o *workflow* com os recursos adicionais. A substituição de componentes do *workflow* em uma composição *coarse-grained* também se mostra desafiadora. Não apenas é necessário separar o conjunto de componentes a ser substituído, mas também é importante adicionar o novo componente à imagem. Esse processo é propenso a erros devido a possíveis conflitos ao integrar componentes de software adicionais para gerar uma nova imagem.

Composições híbridas

A composição híbrida envolve o agrupamento de certos componentes do *workflow* enquanto separa outros em contêineres distintos. Nas Figuras 3.3 e 3.4, exemplificamos a implementação de composições híbridas no SciPhy. Diversas alternativas existem dentro dessa composição, buscando agrupar componentes que compartilham recursos como dados, bibliotecas, *engines* virtuais e sistemas operacionais, ao mesmo tempo que isolam aqueles considerados "independentes". Abordagens descritas em estudos como NIDDODI *et al.* [63], OLAYA *et al.* [67] exploram o uso e a geração de contêineres de dados, investigando como *workflows* podem se beneficiar deles de forma eficiente.

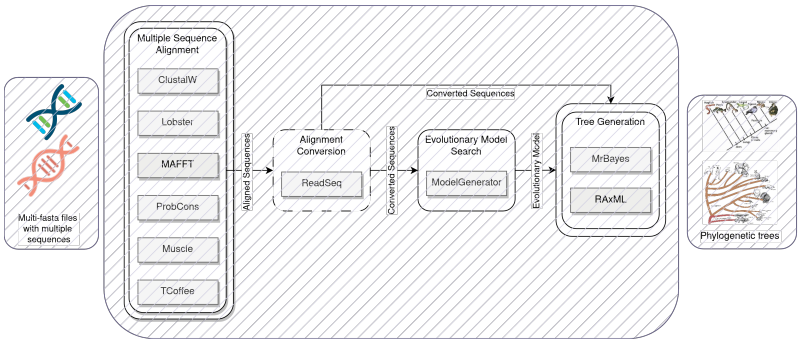


Figura 3.3: Composição híbrida aplicada ao SciPhy. As caixas hachuradas representam os contêineres que encapsulam as atividades do *workflow* e suas dependências. Essa composição é semelhante à *coarse-grained*, mas os dados de entrada e saída são isolados do *workflow*, favorecendo a reprodutibilidade.

O agrupamento de componentes em contêineres introduz várias tarefas, como o gerenciamento de requisições entre contêineres, acesso a dados, tolerância a falhas e outras responsabilidades, que chamamos de tarefas de gerenciamento de contêineres. Na composição híbrida, a repetibilidade e a implantação apresentam um nível de dificuldade moderado, já que dependem de múltiplas imagens que não são autocontidas. Comparada à abordagem *coarse-grained*, o nível de dificuldade para

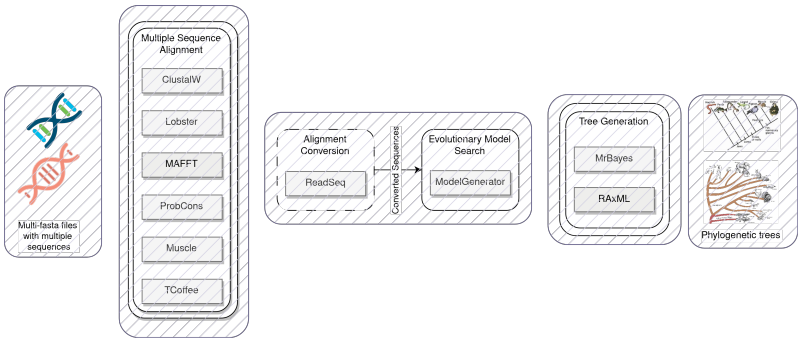


Figura 3.4: Composição híbrida aplicada ao SciPhy. As caixas hachuradas representam os contêineres que encapsulam as atividades do *workflow* e suas dependências. Essa composição é semelhante à *fine-grained*, mas duas atividades (Conversão de formato de alinhamento e Busca pelo melhor modelo evolutivo) são agrupadas, pois possuem os mesmos requisitos de software, evitando contêiner *sprawl*.

repetição/reexecução permanece moderado em composições híbridas devido às tarefas adicionais de gerenciamento de contêineres. O compartilhamento também tem um nível de dificuldade moderado, pois essa composição sempre resulta em no mínimo dois contêineres carecendo de uma entidade autocontida que abranja todo o *workflow*.

A reprodutibilidade, entretanto, se beneficia da composição híbrida, já que os dados e outros componentes podem ser isolados do restante do *workflow*, permitindo a reprodução da execução com diferentes entradas ou métodos equivalentes. As composições híbridas facilitam a reutilização de contêineres de dados envolvidos na composição, mas compartilham as mesmas limitações das composições *coarse-grained* quando se trata de reutilizar imagens para atividades do *workflow*. Em termos de manutenção, as composições híbridas enfrentam problemas semelhantes às composições *coarse-grained*, com a complexidade adicional das tarefas de gerenciamento de contêineres. No entanto, oferecem a vantagem de isolar componentes específicos que são independentes e propensos a erros ou conflitos dentro do *workflow*. A manutenção em composições híbridas é considerada de nível moderado.

No que diz respeito a gestão de recursos, as composições híbridas isolam pelo menos um componente, permitindo ações como a implantação e distribuição de partes em diferentes ambientes. Entretanto, algumas imagens podem agrupar atividades em um único contêiner, restringindo o alcance das ações de gestão de recursos, resultando em um nível moderado de dificuldade.

Na depuração, essa composição compartilha os mesmos desafios da composição *coarse-grained*, pois uma única imagem pode executar mais de uma aplicação. Para o critério de substituição de componentes, o nível de dificuldade também é moderado. Se o desenvolvedor do *workflow* tiver conhecimento prévio sobre os componentes a serem substituídos, esses componentes podem ser isolados e substituídos futura-

mente. No entanto, se for necessário substituir um componente que não está isolado, o desenvolvedor enfrentará um cenário semelhante ao da composição *coarse-grained*.

Composição *fine-grained*

A composição *fine-grained* é caracterizada pela modularização dos componentes do *workflow*, envolvendo não apenas a separação dos serviços de dados e proveniência, mas também o isolamento de cada atividade do *workflow*. Na Figura 3.5, exemplificamos a implementação da composição *fine-grained* no SciPhy. Semelhante às composições híbridas, a composição *fine-grained* exige o gerenciamento de tarefas relacionadas aos contêineres. No entanto, diferentemente das composições híbridas, a quantidade de imagens de contêiner necessárias em uma composição *fine-grained* é diretamente proporcional ao número de atividades do *workflow* e às dependências de dados. Consequentemente, o esforço para as tarefas de gerenciamento aumenta com a complexidade do *workflow*. Executar essas tarefas manualmente, sem ferramentas de suporte, pode ser trabalhoso e propenso a erros, o que torna a composição *fine-grained* difícil para repetição, implantação e compartilhamento do *workflow*.

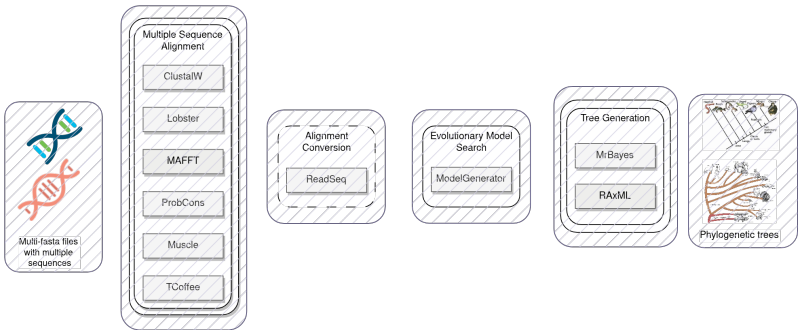


Figura 3.5: Composição *fine-grained* aplicada ao SciPhy. As caixas hachuradas representam cada contêiner que isola cada atividade do *workflow* e suas dependências.

Por outro lado, a reprodutibilidade pode ser considerada relativamente simples, pois envolve a substituição e análise do comportamento de atividades do *workflow* isoladamente. A reutilização de componentes do *workflow* é considerada fácil nessa composição, graças à separação das imagens destinadas às atividades do *workflow* e aos componentes de dados.

A manutenção nessa composição inclui tarefas de gerenciamento de contêineres que aumentam com a complexidade do *workflow*. No entanto, as imagens utilizadas geralmente apresentam tempos mais rápidos de geração, transferência e inicialização. A composição *fine-grained* resolve de forma eficiente potenciais conflitos dentro das atividades do *workflow* e entre o ambiente e a pilha de software do *workflow*, isolando todos os componentes. Adicionar ou substituir funcionalidades/bibliotecas no *workflow* pode ser feito reutilizando imagens de contêiner disponíveis em reposi-

tórios públicos, o que torna a manutenção do *workflow* em composições *fine-grained* mais simples em comparação a outras composições.

No entanto, é essencial reconhecer que o impacto cumulativo das tarefas de gerenciamento dentro da composição *fine-grained* influencia a complexidade da manutenção do *workflow* ao longo do tempo. Assim, categorizamos sua dificuldade como moderada.

Em termos de gestão de recursos, a composição *fine-grained* é considerada fácil, já que imagens menores são mais simples de gerenciar dentro do *workflow*. O desenvolvedor do *workflow* pode limitar recursos para atividades específicas, distribuí-las entre diferentes máquinas ou usar contêineres para a execução paralela de atividades do *workflow*, conforme discutido em SCHULZ *et al.* [78], ZHENG e THAIN [95].

No que diz respeito à depuração, a composição *fine-grained* oferece controle completo sobre os processos que cada contêiner executa. A depuração pode ser facilitada por meio da obtenção de imagens já existentes ou pela criação de novas imagens, dada a velocidade de geração dessas imagens que tendem a ser menores, quando comparadas à composição *coarse-grained*. De forma semelhante, substituir uma atividade do *workflow* ou um conjunto de dados é mais fácil nessa composição, graças à capacidade de identificar com precisão o que precisa ser substituído e, em seguida, gerar ou reutilizar a nova imagem de contêiner para tal.

As composições discutidas nesta seção apresentam seus prós e contras na implantação de *workflows*. Os critérios qualitativos devem ser considerados ao escolher uma composição adequada com base nas demandas de execução do *workflow*. Com base nos critérios mais relevantes para o *workflow*, a escolha deve privilegiar a composição de contêineres de menor dificuldade. Se o *workflow* for executado apenas uma vez para um caso específico, ou se o objetivo for apenas compartilhar e repetir resultados, a composição *coarse-grained* é recomendada. Caso o *workflow* apresente dependências ou requisitos conflitantes, seria mais apropriada uma composição híbrida ou *fine-grained*; se o objetivo é utilizar contêineres para melhorar a *gestão de recursos*, a escolha seria a *fine-grained*, *etc.* No entanto, a escolha está longe de ser simples e não deve ser predefinida pela ferramenta de execução. É necessário também analisar se há uma sobrecarga relevante associada à composição de contêineres, que discutiremos no Capítulo 4.

3.2 Implantação de *workflows* conteinerizados com `ProvDeploy`

Cada composição de contêiner apresenta vantagens e desvantagens. Observamos que o usuário deve ter a liberdade de escolher a composição que seja mais favorável ao seu

workflow e objetivos. No entanto, não encontramos ferramentas que permitissem implantação de diferentes composições do mesmo *workflow* ao mesmo tempo que a rastreabilidade dos dados de contêiner e de *workflow* fossem mantidas. Por isso, nesta seção, introduzimos a `ProvDeploy`, uma ferramenta que permite a implantação de diferentes composições de contêiner para *workflows* com coleta de proveniência de *workflow* e de contêiner.

3.2.1 `ProvDeploy`

A `ProvDeploy` [43] é uma ferramenta projetada para facilitar a implantação de *workflows* científicos conteinerizados em ambientes de PAD, incorporando serviços de proveniência de *workflow*. Ao contrário das abordagens atuais [8, 48, 65], a `ProvDeploy` automatiza a implantação de contêineres para serviços de proveniência integrados ao *workflow*. Isso permite que os usuários monitorem seus *workflows* durante a execução, auxiliando na depuração e possibilitando alterações nos parâmetros durante a execução (caso o *workflow* permita adaptações).

A `ProvDeploy` permite ao usuário escolher entre vários serviços de proveniência disponíveis, mas apenas um pode ser definido como padrão e será utilizado durante a execução do *workflow*. O serviço de proveniência escolhido pode exigir a implantação de contêineres com um sistema de gerenciamento de banco de dados (SGBD), *por exemplo*, MonetDB, PostgreSQL, *etc.*

A `ProvDeploy` recebe as seguintes informações como entrada: (i) a especificação do *workflow* (em um arquivo JSON), (ii) os conjuntos de dados a serem processados, (iii) os *scripts* do *workflow*, e (iv) um catálogo contendo informações sobre as imagens de contêineres disponíveis, que podem ser implantadas utilizando uma das composições de contêiner discutidas na seção anterior. Após a definição da composição de contêiner, a `ProvDeploy` realiza a implantação dos contêineres em um ambiente específico (como um *cluster* ou a nuvem).

A `ProvDeploy` é capaz de suportar a execução da composição escolhida pelo usuário. Caso a composição definida seja *coarse-grained*, a `ProvDeploy` iniciará a imagem única do contêiner. Se a composição for híbrida ou *fine-grained*, a `ProvDeploy` inicia e testa primeiro os contêineres essenciais para a pilha de proveniência, configurando volumes e vinculando diretórios conforme necessário. Seguindo a especificação do *workflow*, a `ProvDeploy` ativa sequencialmente as imagens de contêiner correspondentes a cada atividade do *workflow*, respeitando a ordem definida. À medida que as atividades dentro de um contêiner são concluídas, a `ProvDeploy` inicia o contêiner subsequente juntamente com suas atividades associadas.

No caso da composição *fine-grained*, a `ProvDeploy` copia os *scripts* de execução e substitui as chamadas de atividades por chamadas para a própria `ProvDeploy`,

referenciando a atividade original. Quando a `ProvDeploy` é chamada durante a execução de um *workflow*, ela consulta o banco de proveniência de contêiner para identificar a imagem associada à atividade requerida pelo *workflow* e, em seguida, dispara um contêiner com a chamada original da atividade. Já na composição híbrida, o procedimento é similar ao da composição *fine-grained*, mas a substituição das chamadas ocorre apenas no último *script* de um bloco de atividades agrupadas. Usando a Figura 3.4 como exemplo, as chamadas seriam substituídas apenas nos *scripts* correspondentes às atividades *Alinhamento de múltiplas sequencias* e *Busca do modelo evolutivo*.

É importante destacar que a `ProvDeploy` não controla a execução paralela e distribuída para não ser intrusiva e alterar o funcionamento original do *workflow*. Em cenários em que o *workflow* envolve atividades paralelas, estas são executadas de forma independente dentro dos contêineres, sem interferência da `ProvDeploy`, pois acreditamos que o usuário deve ter total poder de decisão sobre as melhores técnicas de otimização a serem aplicadas em seu *workflow*..

Após a conclusão, a `ProvDeploy` gera um objeto de pesquisa [4](*research object* - RO) como saída. O objeto de pesquisa gerado pela `ProvDeploy` encapsula todos os dados, metadados, bibliotecas e dependências usados na execução do *workflow*, visando à reprodutibilidade. No entanto, é importante notar que este objeto de pesquisa não captura as características específicas do ambiente PAD em que o *workflow* é executado. Em vez disso, ele foca em incluir a pilha de software necessária para reexecutar o *workflow*, o serviço de proveniência selecionado e seu banco de dados de proveniência de *workflow* e o de contêiner associado.

As imagens de contêiner a serem utilizadas pela `ProvDeploy` podem ser obtidas em vários repositórios públicos, como Docker Hub, Binder, NVIDIA NGC, *etc.* Ao aproveitar essas imagens públicas, nosso objetivo é minimizar o tempo de geração dos contêineres e auxiliar os usuários na exploração de composições de contêiner alternativas. Vale ressaltar que a `ProvDeploy` não foi projetada para substituir ferramentas automáticas de implantação e orquestração de contêineres, como o Kubernetes. Pelo contrário, ela pode ser usada em conjunto com o Kubernetes, já que a `ProvDeploy` determina quais contêineres serão incluídos em um *cluster* Kubernetes, e também pode oferecer paralelização e escalonamento horizontal e vertical para o *workflow*.

O código-fonte da `ProvDeploy` está disponível no Bitbucket, acessível pelo seguinte URL: `https://bitbucket.org/lilianeKunstmann/provdeploy/`. A `ProvDeploy` é um projeto de código aberto, o que significa que seu código-fonte está disponível gratuitamente para visualização, modificação e distribuição.

3.2.2 Arquitetura da `ProvDeploy`

A arquitetura da `ProvDeploy` é ilustrada na Figura 3.6 e consiste em cinco componentes principais: (i) Catálogo, (ii) Inicializador, (iii) *Prov-Parser*, (iii) *Deployer* e (iv) *Wrapper*. O *Catálogo* é uma entidade no banco de dados que armazena metadados relacionados às imagens de contêiner que podem ser implantadas pela `ProvDeploy`. Ele inclui informações como a *tag* da imagem, registro, descrição, arquivo de definição (*e.g.*, Dockerfiles, *recipes*), instruções de implantação e requisitos (*e.g.*, criação de volumes, portas públicas).

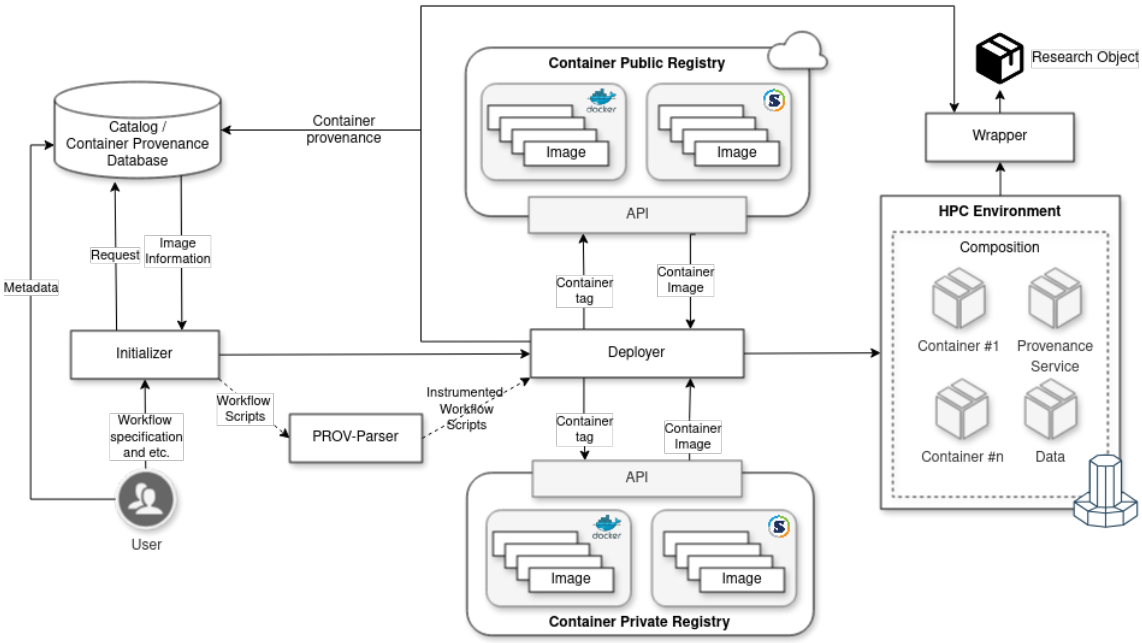


Figura 3.6: Arquitetura da `ProvDeploy`

O *Catálogo* foi integrado ao banco de dados de proveniência de contêiner apresentado na seção 3.3, que armazena automaticamente a proveniência de contêiner obtida durante a execução de diferentes composições. Essa proveniência é projetada para permitir que o usuário identifique facilmente as imagens utilizadas, portas, *drivers*, volumes e caminhos vinculados configurados, além do serviço de proveniência empregado e a composição de contêiner utilizada em um *workflow*.

As imagens presentes no *Catálogo* são adicionadas pelo usuário e podem ser locais, privadas ou públicas, sendo baixadas quando o *Deployer* inicia a execução. Além disso, o *Catálogo* contém detalhes sobre os serviços de proveniência compatíveis com a `ProvDeploy` (*e.g.*, DfAnalyzer [82], noWorkflow [68]), bem como imagens de contêiner de SGBDs (*e.g.*, MonetDB, PostgreSQL, MySQL, *etc.*). Esse componente também gerencia metadados sobre as versões dos arquivos de configuração de contêiner (*e.g.*, Dockerfile, *recipe*).

O *Inicializador* atua como a interface do usuário com a `ProvDeploy`, permitindo

a adição de novas imagens ao *Catálogo*, inclusão de serviços de proveniência, o envio do *workflow* para implantação e o acesso aos dados de proveniência, tanto do contêiner quanto do *workflow*, durante a execução do *workflow*. Ele consulta o *Catálogo* para verificar se as imagens necessárias para o *workflow* estão disponíveis e quais são seus requisitos.

Alguns serviços de proveniência, como a DfAnalyzer, exigem instrumentação, ou seja, a inserção de chamadas de proveniência no *script* do *workflow*. O *Inicializador* verifica se o *script* do *workflow* já está instrumentado para capturar dados de proveniência com o serviço selecionado. Caso o *script* já esteja preparado, ele é enviado diretamente ao *Deployer*. Caso contrário, o *Prov-Parser* é acionado para instrumentar o *script*, identificando os pontos onde os dados de proveniência devem ser capturados. O tipo de instrumentação realizada depende do serviço de proveniência selecionado, e envolve a inserção de chamadas para o serviço diretamente nos *scripts* do *workflow*.

O *Prov-Parser* analisa as funções presentes no *script* do *workflow*, associando cada função a uma transformação de dados no modelo de proveniência do *workflow*. Quando o *script* não possui funções explícitas (*e.g.*, `def <<Nome da Função>>():` em Python), o *Prov-Parser* assume que todo o *workflow* corresponde a uma única função. Embora mais trabalhosa e propensa a erros, a instrumentação manual oferece maior flexibilidade para capturar os dados de proveniência na granularidade desejada.

Na versão atual, a `ProvDeploy` utiliza o DfAnalyzer como serviço de proveniência padrão. O DfAnalyzer emprega o MonetDB como SGBD associado e o FastBit para indexação de dados, garantindo um armazenamento eficiente das informações capturadas. Opcionalmente, é possível substituir o *Prov-Parser* por ferramentas de instrumentação especializadas, como o *DfParser* (`https://dfparser.vercel.app/script`), que também é parte dos resultados desta pesquisa.

O *Deployer* é o componente da `ProvDeploy` que permite a aplicação de múltiplas composições de contêiner. Ele segue a composição descrita no arquivo de especificação do *workflow* e configura as chamadas para iniciar e parar os contêineres de acordo com as atividades do *workflow*. O *Deployer* configura o ambiente onde o *workflow* do usuário e o serviço de proveniência escolhido são implantados e podem ser executados.

Além disso, o *Deployer* é responsável por registrar a composição de contêiner, juntamente com as imagens utilizadas e o tempo de execução, auxiliando os usuários a identificar as imagens usadas na execução de atividades específicas. Ao final da execução, o *Deployer* invoca o *Wrapper* para gerar um objeto de pesquisa relacionado à execução do *workflow*.

A proveniência do contêiner capturada pela `ProvDeploy` é incluída no objeto de

pesquisa pelo *Wrapper*, junto com a proveniência do *workflow*. Embora o objeto de pesquisa não represente completamente a execução, a proveniência do contêiner fornece informações complementares, por exemplo para identificação do ambiente de execução. Essa arquitetura permite que cientistas aproveitem ao máximo seus recursos ao executar *workflows*, integrando-se à composição de contêiner implantada pela `ProvDeploy`.

A arquitetura da `ProvDeploy` contribui com a captura de dados de proveniência do *workflow* e do contêiner, além de viabilizar a implantação de composições híbridas.

3.3 Modelo de dados de proveniência contêiner-*aware*

Quando foi identificada a necessidade de estender a `ProvDeploy`, observamos que para mantê-la não intrusiva e atendendo a diferentes composições, seria necessário algum registro externo das imagens de contêiner e a sua associação às atividades do *workflow*. Junto a isso, observamos que há informações associadas a execução que os atuais serviços de proveniência não são capazes de capturar ou quando possibilitam a captura, esses serviços podem acabar gerando dados inconsistentes a respeito do ambiente de execução, se executados via contêiner. Além disso, uma vez que o cientista adota uma composição, a maneira que os contêineres estão associados a uma ou mais atividades é uma informação necessária a reexecução do *workflow*. Nesta seção, apresentamos um modelo abrangente de dados de proveniência de *workflows*, implementado na `ProvDeploy`, para torná-la contêiner-*aware*. Esse modelo estende os dados de proveniência do serviços acoplados à `ProvDeploy` com a proveniência de contêineres. Usando o modelo de dados apresentado na Figura 3.7, buscamos representar a proveniência de contêineres e fornecer análises significativas junto com a proveniência de *workflows*, sem vincular o usuário a um serviço específico de proveniência. Este modelo de dados é baseado inicialmente nos dados providos pela OCI e *Common HPC Container Conformance Initiative*[2], além de dados relevantes sobre inicialização e parada de contêineres e sua reprodução que são apresentados em CANON [10], GRUENING *et al.* [31], PRIEDHORSKY *et al.* [72], STRAESSER *et al.* [86], WOFFORD *et al.* [92].

O modelo apresentado na Figura 3.7 segue a recomendação W3C PROV-DM, onde a proveniência é representada em termos de entidades (os objetos de dados), atividades (transformações de dados) e agentes (usuários e sistemas), juntamente com suas relações definidas pelo W3C PROV-DM. No modelo proposto, as classes

[2]https://github.com/container-in-hpc/container-hpc-conformance

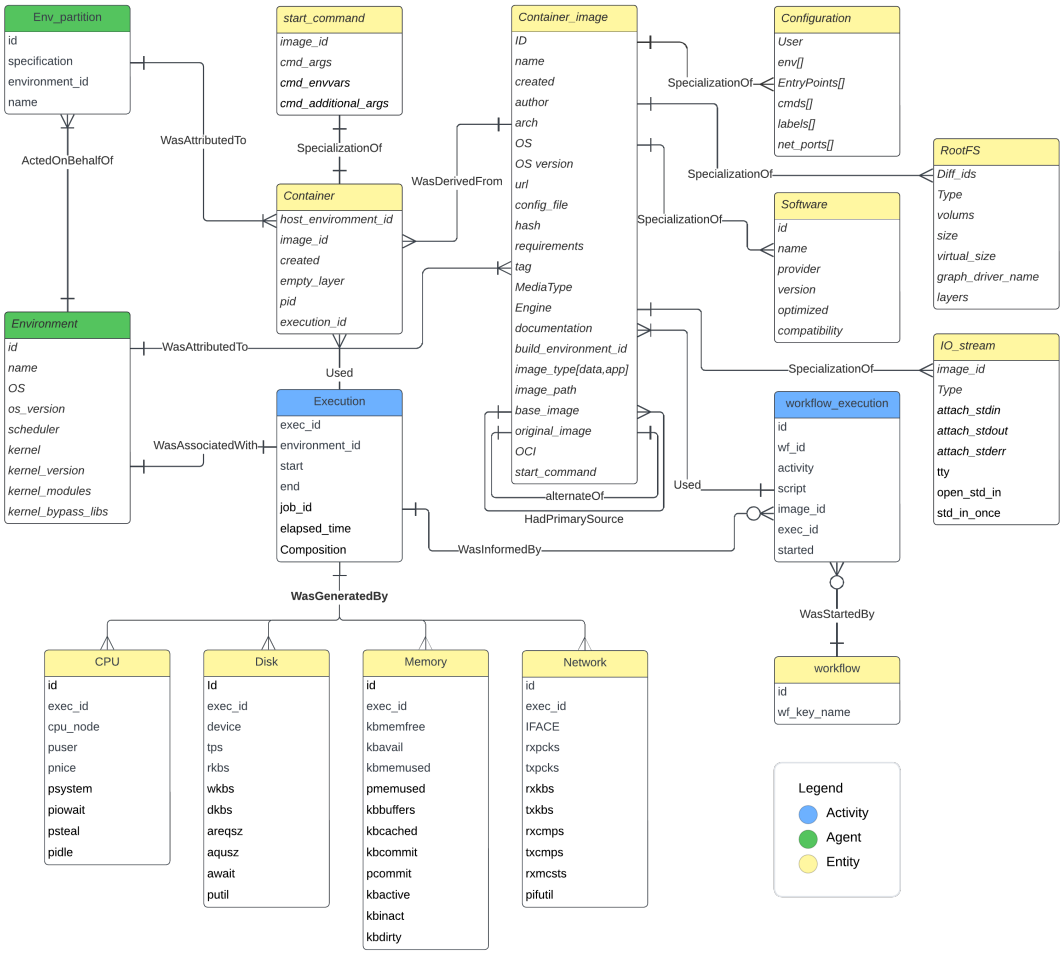


Figura 3.7: Diagrama PROV-DM do modelo de dados de proveniência contêiner-*aware*.

amarelas representam as Entidades, as classes verdes representam os Agentes e as classes azuis representam as Atividades. As cores do nosso diagrama de classes são representadas como estereótipos em conformidade com as classes do W3C PROV. Este modelo de dados tem como objetivo fornecer informações sobre as origens e características de imagens de contêineres, aprimorando sua *findability* e confiabilidade para o processo de reconstrução. Ao seguir as anotações da OCI, o modelo proposto é capaz de coletar dados de diferentes *engines* e no caso de imagens convertidas entre *engines*, é possível encontrar a imagem de contêiner original. Implementamos esse modelo de dados na `ProvDeploy` usando o MonetDB, um SGBD orientado a colunas, que como solução de banco de dados, permite consultar grafos de proveniência e facilita a integração com outras bibliotecas para análise e visualização.

A entidade que permite a conexão entre a proveniência de contêineres e a proveniência de *workflows* é a entidade *workflow*, que possui como atributo o nome do *workflow* (*wf_key_name*), utilizado para identificá-lo no banco de dados de pro-

veniência do serviço de captura de proveniência (*e.g.*, *dataflow_tag* na DfAnalyzer [82] ou *hashID* no noWorkflow [62]). A atividade *execution* associa a composição de contêineres, que pode ser *coarse-grained* (um contêiner para o *workflow* completo), *fine-grained* (um contêiner para cada atividade do *workflow*) ou híbrida (múltiplas atividades, mas não todas, executadas pelo mesmo contêiner). Essa entidade também armazena o *job_id*, proveniente de escalonadores em ambientes PAD, que pode ser utilizado posteriormente para ajuste fino e depuração.

A entidade *container_image* representa os detalhes da imagem de contêiner, permitindo que ela seja reconstruída. Uma imagem de contêiner é especializada pelas entidades *configuration*, *rootFS* (volumes requeridos), *software* que compõe a imagem e os dados de *io_stream*. Essas informações também são armazenadas no arquivo de configuração ou *recipe* da imagem de contêiner (*config_file*) e na própria imagem de contêiner sendo acessíveis via *inspect*. No entanto, não há garantia de que recompilar a imagem de contêiner usando o arquivo de configuração produzirá uma imagem idêntica [10], ou de que o arquivo de configuração esteja atualizado em relação à imagem. Esta entidade representa o *Catálogo* dentro da arquitetura da `ProvDeploy`, apresentada na Seção 3.2.2.

Adicionalmente, as imagens de contêiner estão vinculadas ao ambiente em que foram geradas (*build_environment_id*). Esse ambiente define as arquiteturas de *kernel* compatíveis com o contêiner. Uma imagem de contêiner pode dar origem a outras imagens, seja como uma fonte primária (*base_image*) para uma nova imagem de contêiner, ou por meio da geração de versões alternativas com diferentes *engines* de contêiner (*original_image*) através da conversão de imagens de contêiner. Por exemplo, uma imagem de contêiner criada no Docker pode ser convertida para Singularity. Nesses casos, ela deixa de estar associada a um arquivo de configuração (*config_file*) e passa a referenciar a imagem de contêiner original.

A entidade *container* descreve o processo isolado que é executado utilizando uma imagem de contêiner e especifica o ambiente no qual ele é executado. Um contêiner depende de uma imagem de contêiner para ser iniciado, e uma única imagem pode ser usada para executar múltiplos contêineres. Essa entidade é utilizada durante a atividade de *execution* e documenta o que ocorreu durante a execução do *workflow*. As atividades *execution* e *workflow_execution* capturam o comportamento esperado do *workflow* conteinerizado, conhecido como proveniência prospectiva. A entidade *container* registra as atividades realizadas, conhecidas como proveniência retrospectiva. Caso uma atividade não inicie, o contêiner relacionado a ela não é registrado. A entidade *container* é especializada pela entidade *start_command*, que armazena os comandos, argumentos e variáveis de ambiente usados ao iniciar o contêiner, pois o usuário pode iniciar uma imagem de contêiner com argumentos diferentes dos armazenados na *container_image*. Saber como uma imagem de con-

têiner é utilizada é importante para sabermos quais aspectos (programas e arquivos) da imagem são necessários à um determinado *workflow*. Essa informação é essencial caso haja necessidade de reduzir as imagens de contêiner e para reprodução do *workflow*.

O agente *environment* descreve o ambiente associado a uma execução (*execution*), incluindo sua arquitetura de *kernel* e escalonador. Ambientes de PAD e nuvem podem conter recursos heterogêneos, por isso o *env_partition* foi projetado para detalhar as características do ambiente específico utilizado. Essa partição contém as especificações de *hardware* do ambiente e é onde o *workflow* é executado.

A atividade *execution*, juntamente com a atividade *workflow_execution*, descreve a execução conteinerizada. Isso inclui a composição dos contêineres, as atividades do workflow e os contêineres utilizados, além dos comandos e parâmetros esperados para iniciar cada atividade. Essas atividades aprimoram o gerenciamento de execução pelo usuário, especialmente no caso de serviços de proveniência, como o DfAnalyzer [82], que não distinguem nativamente múltiplas execuções concorrentes de *workflows*.

Por último, adicionamos as entidades *CPU*, *Disk*, *Memory* e *Network*, com intuito de permitir a associação de dados de perfilagem à execução de um *workflow* e assim ter maior noção do impacto das diferentes composições de contêiner. Os atributos dessas entidades são baseados na biblioteca Sysstat. Ao associarmos esses dados à chamada dos contêineres, podemos gerar melhores explicações para o uso de recursos, em caso de uso de controle de recursos via contêiner.

3.3.1 Consultas de proveniência

Este modelo tem como objetivo oferecer suporte a uma análise abrangente de toda a execução do *workflow* em um ambiente conteinerizado. Para mostrar sua validade, levantamos alguns exemplos de consultas, apresentados na Tabela 3.2, que o modelo de dados de proveniência contêiner-*aware* pode responder quando conectado ao modelo de proveniência do *workflow*. Essas consultas são inspiradas no *First Provenance Challenge*[3] e estão categorizadas em informações relacionadas ao reuso de contêineres, reprodutibilidade de *workflows* e *insights* para análise de rastreabilidade de *workflows*.

Os *provenance challenges* representam iniciativas que buscam identificar, abordar e discutir problemas relacionados à modelagem, captura, armazenamento, consulta, interoperabilidade e uso de dados de proveniência em diferentes sistemas e aplicações. Esses desafios, frequentemente propostos pela comunidade científica de proveniência, como a *Open Provenance*[4], têm como objetivo aprimorar padrões, ferramentas e práticas no campo da proveniência.

[3]https://openprovenance.org/provenance-challenge/FirstProvenanceChallenge.html
[4]https://openprovenance.org/

Tabela 3.2: Consultas de proveniência de contêiner e *workflow* inspirada no *First Provenance Challenge*.

#	**Consulta**	**Tipo**
Q1	Recupere o *job_ id*, os contêineres e os ambientes de execução envolvidos na execução de um *workflow*.	Reuso Reprodutibilidade
Q2	Recupere as imagens de contêiner associadas ao *workflow* com seu *job_ id* e as atividades que elas executaram.	Reuso Reprodutibilidade
Q3	Recupere todos os *workflows* que foram executados com uma determinada imagem de contêiner.	Reuso
Q4	Recupere todas as atividades de *workflow* que foram executadas com uma determinada imagem de contêiner.	Reuse
Q5	Recupere todas as imagens que foram criadas por um usuário específico.	Reuso
Q6	Recupere todos os *workflows* em que uma determinada variável satisfez uma condição específica e foi executado com uma composição específica.	Análise Reprodutibilidade
Q7	Um usuário executou o *workflow* com diferentes composições e em ambientes distintos, aumentando o número de contêineres. Recupere as diferenças nas composições entre os diferentes ambientes.	Análise
Q8	Qual ambiente (máquina e contêiner) executou o *workflow* com os melhores resultados?	Análise Reprodutibilidade

A consulta Q1 recupera dados que geralmente estão disponíveis apenas durante a execução, com destaque para o registro dos contêineres efetivamente utilizados na execução do *workflow*, já que registrar apenas as imagens fornece informações limitadas sobre a execução. Essa consulta pode ser respondida com a junção das entidades *execution*, *workflow_ execution*, *container_ image* e *env_ partition* e definindo o *workflow* de interesse.

A consulta Q2 apresenta informações sobre as imagens de contêiner utilizadas na execução do *workflow*, o que pode ajudar os usuários em execuções futuras de *workflows* com uma pilha de software semelhante. Usando o *job_ id* associado, também é possível encontrar informações adicionais sobre o *job* executado na instalação de PAD associada, como possíveis tarefas realizadas antes ou depois do *workflow*, que não são capturadas pela proveniência do *workflow*. A consulta Q2 demanda a junção das entidades *workflow*, *workflow_ execution* e *container_ image*, definindo o *workflow* de interesse por meio do atributo *wf_ key_ name*.

A consulta Q3 mostra as possibilidades de reutilização de imagens de contêiner em múltiplos *workflows* com requisitos de software similares, e a consulta Q4 opera de forma semelhante à Q3, mas fornece informações mais detalhadas sobre as atividades executadas pelas imagens de contêiner. A consulta Q4 demanda a junção das entidades *container_ image*, *workflow* e *workflow_ execution*.

A consulta Q5 explora os atributos da entidade *container_ image*. Essa consulta ajuda a identificar imagens base compartilhadas e arquiteturas de execução (*arch*). Se um usuário planeja executar qualquer uma dessas imagens de contêiner em um ambiente diferente, o primeiro passo seria verificar se a imagem base tem uma versão compatível com a arquitetura do *kernel* do ambiente de destino.

As consultas Q6, Q7 e Q8 são consultas mais complexas que dependem tanto da proveniência do contêiner quanto da proveniência do *workflow*. Essas consultas permitem que o usuário enriqueça o modelo de proveniência do *workflow*, que é expandido pelo modelo de dados de proveniência contêiner-*aware*. Essas consultas fazem junção entre as entidades do modelo de proveniência de contêiner *execution, workflow_execution* e as entidades do modelo de proveniência do *workflow*, começando pela entidade que identifica o *workflow* com o mesmo identificador da entidade *workflow*. Para o modelo de proveniência da DfAnalyzer, esse identificador é armazenado em uma entidade chamada *dataflow*. Após a junção com a proveniência do *workflow*, o usuário pode explorar os dados dentro de um intervalo que começa no *execution_start* e termina no *elapsed_time* adicionado. Essas consultas ajudam a entender o impacto da composição do contêiner no desempenho geral do *workflow* em diferentes ambientes.

As consultas Q1, Q2 e Q7 não podem ser respondidas utilizando apenas a proveniência do *workflow*. Durante o desenvolvimento de um *workflow* científico, os usuários frequentemente exploram múltiplos algoritmos e bibliotecas de software, resultando em diversos contêineres e imagens de contêiner combinadas com diferentes origens e ambientes. Nas abordagens atuais, os contêineres não estão conectados às suas imagens de contêiner associadas, ambientes, bibliotecas, atividades ou *workflows*. Essas informações podem estar dispersas entre *logs*, arquivos e *scripts*. Sem uma estrutura predefinida para representar essas informações, sua análise se torna cada vez mais difícil e pode até não ser possível, embora seja essencial para a reprodutibilidade.

As consultas Q3, Q4, Q5 e Q7 podem melhorar o reuso de imagens de contêiner e reduzir a execução e o armazenamento de imagens de contêiner desnecessárias para executar um *workflow*, uma vez que uma imagem de contêiner pode ser usada para executar várias atividades do *workflow* e gerar novas imagens de contêiner. Além disso, com o modelo de dados proposto, é possível rastrear o caminho de derivação de uma imagem de contêiner para reconstruí-la.

3.3.2 Comparação do modelo de dados com abordagens atuais

Para comparar as abordagens existentes, nós avaliamos a capacidade das abordagens apresentadas na seção de trabalhos relacionados (2.4) de responder as consultas apresentadas na seção 3.3.1. O resultado da avaliação é apresentado na tabela 3.3

É importante destacar que as abordagens apresentadas, demandam diferentes quantidades de pós-processamento para atender as consultas apresentadas. Isto se dá por que diferente do modelo de dados proposto nesta tese, essas ferramentas

Tabela 3.3: Suporte por abordagem para as consultas apresentadas na seção 3.3.1.

Container Provenance Solution	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
`CLARION` [12]	-	-	-	Sim	-	-	-	-
`ALASTOR` [14]	-	-	-	Sim	-	-	-	-
`PACED` [1]	Sim	-	-	Sim	-	-	-	-
`DisProTrack` [76]	-	-	-	Sim	-	-	-	-
MODI *et al.* [60]	-	Sim	-	Sim	-	-	-	-
WOFFORD *et al.* [92]	Sim	-	-	Sim	-	-	-	-
`PROV-CRT` [2]	-	-	-	Sim	Sim	-	-	-
OLAYA *et al.* [67]	-	-	Sim	Sim	-	Sim	-	-
`ProvDeploy` [45]	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim

fazem seu registro através de *logs* e metadados que geralmente são capturados por contêiner, ficando espalhados e desconexos.

Dentre as consultas Q4 é a única que consegue ser respondida por todas as abordagens, no entanto, observamos que abordagens como `CLARION`, `ALASTOR` e `DisProTrack` , não realizam o registro da atividades ou *workflow*, mas esses dados que podem ser derivados a partir dos dados dos contêineres registrados. Enquanto no modelo apresentado, temos esses dados capturados de forma explícita e facilmente acessível.

`PACED` e WOFFORD *et al.* [92] são trabalhos que ao destacar a importância do ambiente de execução, conseguem incluí-lo em seu rastreio, possibilitando a resposta da consulta Q1. No entanto, não há um registro ou associação dos *workflows* e imagens executados. MODI *et al.* [60] se destaca enquanto única abordagem capaz de atender Q2, mesmo que parcialmente, pois essa abordagem não gerencia dados de ambiente como *job_id*.

O `PROV-CRT` como uma das primeiras abordagens para coleta de dados em *engines* de contêiner, se destaca como única abordagem capaz de responder Q5, pois a captura dos dados se dá principalmente na geração das imagens de contêiner. Por último, a abordagem de OLAYA *et al.* [67] se destaca ao responder consultas que demandam proveniência de *workflow* associada a de contêiner. No entanto a falta de suporte a composições distintas e uma proveniência de *workflow* que registra apenas parâmetros de entrada e saída impede a geração de análises mais ricas.

O modelo implementado na `ProvDeploy` é capaz de permitir todas as consultas apresentadas por modelar as relações entre os diferentes metadados associados a execução conteinerizada. Sua adoção do W3C-PROV permite que seja extensível e facilita o acoplamento de *plugins* e intraoperabilidade. Ao adotarmos as anotações da OCI, asseguramos que o modelo não permaneça associado a uma *engine* de contêineres específica. Além disso, permitimos ao usuário uma proveniência de *workflow*

que não é dependente dos contêineres ou da composição pois assumimos que existe um serviço de proveniência que captura dados de interesse do usuário e que serão complementados pela proveniência capturada pelo modelo de dados desenvolvido.

Neste capítulo, apresentamos a abordagem proposta. Inicialmente apresentamos uma bordagem qualitativa de composições de contêiner que evidencia as vantagens e desvantagens de cada composição. Dado que cada composição apresenta diferentes benefícios, argumentamos que sua adoção deve ser uma escolha do usuário. Para facilitar a adoção de diferentes composições com coleta de proveniência, apresentamos a `ProvDeploy` como forma de implantar as múltiplas composições. E observando os desafios da implantação de *workflows* conteinerizados, apresentamos o modelo de dados de proveniência para contêineres de modo a representar dados dos contêineres e integrá-los a proveniência do *workflow*. Por último, como forma de mostrar a validade do modelo, apresentamos uma séries de consultas pertinentes a proveniência de contêiner e de *workflow*, e finalizamos mostrando como os trabalhos relacionados atendem a essas consultas.

Capítulo 4

Experimentos

Este capítulo apresenta os experimentos sobre diferentes composições de contêiner com o objetivo de observar o impacto de cada composição no tempo de execução do *workflow* e consumo de recursos. Além disso, apresentamos um estudo de caso com a proveniência de *workflow* para contêiner. A `ProvDeploy` foi utilizada em todos os experimentos, e os dados de proveniência foram essenciais para fazer as análises de desempenho das composições. Com isso, a `ProvDeploy` foi avaliada em diferentes ambientes computacionais e *workflows*.

4.1 Avaliação de diferentes composições com *Workflows* do Mundo Real com a `ProvDeploy`

Esta seção apresenta a análise das composições de contêineres para três *workflows* científicos do mundo real: SciPhy [15], Montage[39] e DenseED [27]. O SciPhy é um *workflow* de Bioinformática intensivo em dados, o Montage é um *workflow* clássico da Astronomia, também intensivo em dados, e o DenseED é um *workflow* de Aprendizado de máquina guiado por Física, intensivo em CPU. O `ProvDeploy` é utilizado para capturar a proveniência das execuções, definir a composição e implantar os *workflows*. O SciPhy foi implantado com o `ProvDeploy` em dois ambientes: (i) o supercomputador Santos Dumont (SDumont) [7, 11], e (ii) o ambiente em nuvem da AWS com uma configuração bem menor do que a do SDumont. O Montage foi implantado apenas na nuvem da *Google Cloud Platform*(GCP) devido a não disponibilidade do SDumont na época de execução dos experimentos com Montage. O DenseED foi implantado com o `ProvDeploy` apenas no SDumont devido à sua execução intensiva em CPU e GPU. O objetivo dos experimentos é complementar a análise qualitativa com uma avaliação de desempenho de diferentes composições de contêiner. Os experimentos apresentados são avaliados com base no tempo decorrido e no consumo de recursos como CPU e memória.

4.1.1 Estudo de Caso: Análise Filogenética com o *Workflow* SciPhy

O SciPhy é um *workflow* que gera árvores filogenéticas com máxima verossimilhança, conforme mostrado na Figura 4.1. Inicialmente projetado para trabalhar com sequências de aminoácidos, ele pode ser utilizado para outros tipos de sequências biológicas. O SciPhy é um *script* composto por quatro atividades principais, que são: (i) alinhamento de múltiplas sequências (AMS); (ii) conversão de formato de alinhamento; (iii) busca pelo melhor modelo evolutivo; e (iv) construção da árvore filogenética. Existem várias alternativas de programas que podem ser usadas para a atividade de AMS. A Figura 4.1 representa seis alternativas de AMS (variantes): MAFFT [38], Lobster [54], ClustalW [88], Muscle [19], T-Coffee [64] e Prob-Cons [16]. Isso indica que os critérios de substituição e reutilização são necessários ao SciPhy, sugerindo isolar uma imagem de contêiner para a atividade de AMS. A atividade de conversão de alinhamento, ReadSeq [30], é opcional e também deve ser isolada. A última atividade representa a construção das árvores filogenéticas. Os programas alternativos para a atividade de Construção de Árvore Filogenética são RAxML [85] ou MrBayes [37], o que também sugere uma imagem de contêiner apenas para essa atividade, para auxiliar na substituição e reutilização de componentes.

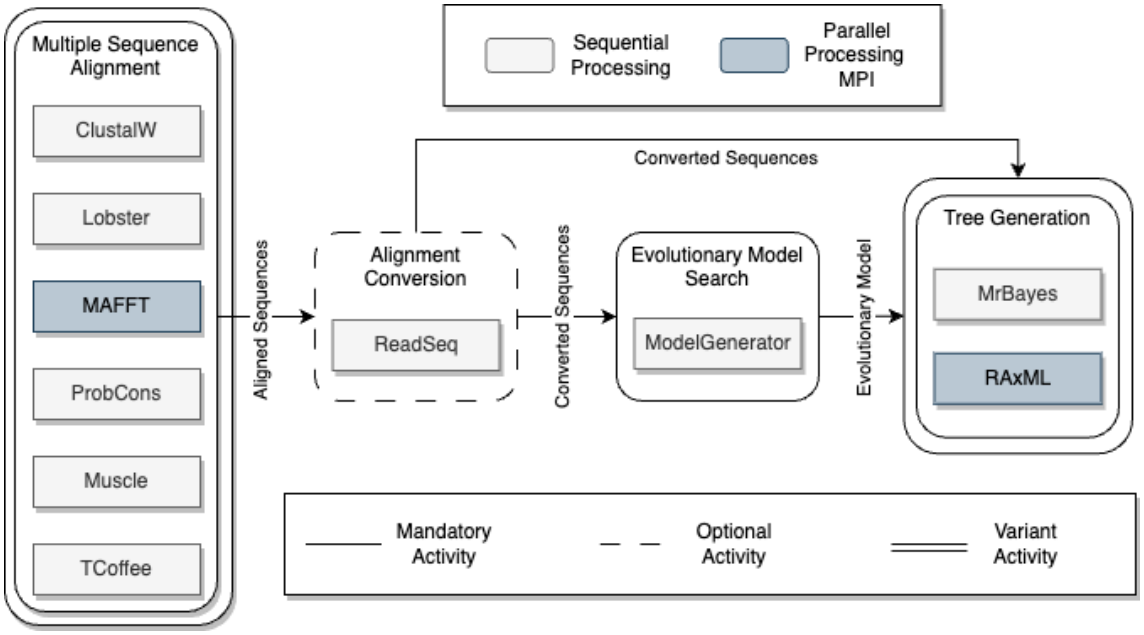


Figura 4.1: *Workflow* SciPhy

Com base nestas especificidades do *workflow*, uma análise qualitativa do SciPhy sugeriria uma composição híbrida ou *fine-grained* para promover a gestão de recursos, reutilização e substituição de componentes. Por outro lado, duas atividades do SciPhy e o serviço de proveniência exigem uma máquina virtual Java, o que sugeriria o compartilhamento da instalação do Java, agrupando essas três atividades em um

único contêiner, sugerindo uma composição híbrida ou *coarse-grained*. Os experimentos de desempenho têm como objetivo adicionar mais dados para encontrar a melhor composição de contêiner a ser executados com a `ProvDeploy`.

Composições de Contêiner

O SciPhy se beneficia de contêineres devido à sua pilha de software complexa. Em nossos experimentos, o SciPhy foi configurado com os seguintes programas de Bioinformática: MAFFT, ReadSeq, ModelGenerator e RAxML. O serviço de proveniência utilizado foi o DfAnalyzer. Embora o SciPhy seja computacionalmente simples, na prática, ele exige PAD devido ao processamento intensivo dos dados consumidos e produzidos. Um experimento típico de filogenia pode analisar centenas de arquivos pequenos, processando centenas ou milhares de sequências biológicas.

Um experimento do SciPhy executa repetidamente as quatro atividades do *workflow* para cada sequência biológica de entrada. De acordo com KELLER TESSER e BORIN [40], embora os contêineres em PAD possam alcançar um desempenho comparável ao nativo, eles tendem a degradar o desempenho à medida que o número de contêineres aumenta, especialmente com tarefas pequenas, como é o caso do SciPhy.

Executamos o SciPhy com quatro composições de contêineres: *coarse-grained*, duas híbridas (parcialmente modular e modular de proveniência) e *fine-grained*. A composição *coarse-grained* serve como base comparativa para avaliar a sobrecarga de ter mais de um contêiner nas outras composições.

A composição híbrida inicialmente avaliada foi a parcialmente modular(*partially modular*), com dois contêineres: um para as atividades do SciPhy com o serviço de proveniência, e outro para o banco de dados de proveniência do *workflow* e dados. Essa composição híbrida favorece o compartilhamento da instalação do Java, que é utilizada em duas atividades do SciPhy (ReadSeq e ModelGenerator) e no serviço de proveniência. A terceira composição adota a composição híbrida anterior, mas adiciona um contêiner separado para o serviço de proveniência, modular de proveniência (*provenance modular*). A composição *fine-grained* possui seis contêineres: uma imagem para cada uma das quatro atividades do SciPhy, outra para o serviço de proveniência e a última para o banco de dados de proveniência e dados.

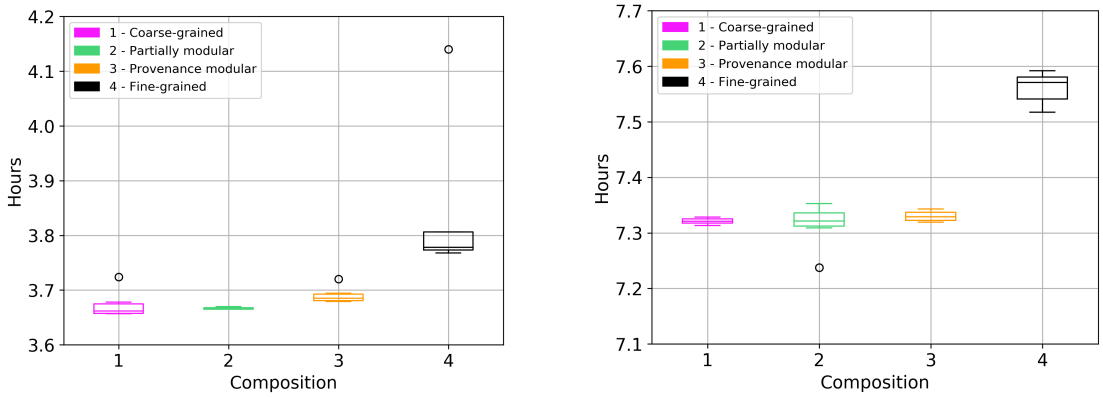
Avaliação Quantitativa no Supercomputador SDumont

O SDumont é um supercomputador que possui uma capacidade de processamento instalada na ordem de 5,1 Petaflop/s (5,1 x 10^{15} *float-point operations per second*), apresentando uma configuração híbrida de nós computacionais, no que diz respeito à arquitetura de processamento paralelo disponível. Em nossos experimentos, utilizamos um nó computacional com dois processadores Intel Xeon E5-2695v2 Ivy Bridge

de 2,4 GHz, 24 núcleos (12 por CPU), 64 GB de RAM DDR3 e sistema operacional Linux RedHat 7.6, e para o perfilamento, utilizamos a biblioteca sysstat 12.

Utilizamos o Singularity 3.8 como o *engine* de contêiner, pois ele é orientado para PAD e é a *engine* recomendada no SDumont e em vários trabalhos [13, 40]. Projetamos dois experimentos de filogenia com o SciPhy. Um possui 200 arquivos de dados multi-fasta de entrada e o outro 400 arquivos. Cada arquivo contém um número diferente de sequências biológicas a serem processadas. Nosso objetivo ao dobrar os arquivos de entrada é avaliar o comportamento de desempenho de acordo com o número de arquivos em todas as composições. Cada experimento foi repetido seis vezes. Quando uma execução foi identificada com valores atípicos, ela foi representada em nossos gráficos, mas descartada dos cálculos de métricas para comparação entre as composições. Nossa análise é baseada em valores de tempo decorrido e consumo de CPU. Também obtemos o tamanho das imagens dos contêineres para avaliar o impacto de ter imagens de contêiner maiores em comparação com acomodar imagens menores de cada vez, considerando que essas imagens precisam ser frequentemente descarregadas (*pull*) e carregadas (*push*).

Começamos comparando as quatro composições de contêiner apresentando o tempo decorrido em horas, conforme mostrado nas Figuras 4.2a e 4.2b, com 200 e 400 arquivos de entrada, respectivamente. Observamos que o tempo total de execução em todas as composições aumentou de acordo com o número de arquivos de entrada, indicando estabilidade na sobrecarga de gerenciamento de contêineres. Por exemplo, a composição *fine-grained* levou, em média, 3,7h para 200 arquivos e 7,37h para 400 arquivos.



(a) 200 arquivos multi-fasta; (b) 400 arquivos multi-fasta;

Figura 4.2: Tempo de execução do SciPhy em horas no SDumont.

Para 200 arquivos multi-fasta (Figura 4.2a), o tempo total de execução de todas as composições é muito semelhante, com um $\sigma \leq 0.04$ (σ - desvio padrão) para todas as composições. A composição *fine-grained* apresenta uma pequena diferença de ≈2,73% em relação ao tempo médio decorrido da composição *coarse-grained*.

A diferença em minutos foi de cerca de 6 minutos, o que é desprezível para uma execução que dura mais de 3 horas.

Na execução de 400 arquivos multi-fasta (Figura 4.2b), foi possível identificar novamente uma pequena variação no tempo de execução para as composições *coarse-grained*, parcialmente modular e modular de proveniência. No entanto, a composição *fine-grained* apresentou uma diferença de ≈3,19% em relação ao tempo médio decorrido da composição *coarse-grained*. Ao realizar o teste ANOVA unidirecional, identificou-se uma diferença significativa em $\overline{x}$ na composição *fine-grained* para as outras composições. A análise *post-hoc* com correção de Bonferroni, utilizando $\alpha = 1,25\%$, resultou em p-value $< \alpha$, rejeitando a hipótese nula, o que significa que a diferença encontrada na composição *fine-grained* é estatisticamente significativa. Isso indica que essa diferença entre a composição *fine-grained* e as outras composições tende a aumentar conforme o número de entradas cresce. No entanto, em termos absolutos, a diferença foi de cerca de 14 minutos, o que pode ser considerado desprezível em uma média de mais de 7 horas de tempo decorrido.

Para aprofundar a avaliação da sobrecarga de adicionar contêineres à execução do SciPhy com as quatro composições, medimos o consumo médio de CPU para cada execução de sequência multi-fasta do SciPhy. O SciPhy utiliza eficientemente o poder de processamento do nó do SDumont ao distribuir suas execuções por todos os 24 núcleos de CPU. Essa distribuição se reflete na observação de que o consumo da CPU atinge picos de 100% em diversos núcleos, enquanto outros núcleos podem apresentar 0% de uso, destacando a natureza dinâmica da alocação de recursos pelo SciPhy.

Para melhorar a legibilidade, na Figura 4.3, apresentamos o consumo de CPU de todas as composições durante os primeiros 180 segundos de execução. Como a execução do SciPhy para cada sequência de entrada dura cerca de 90 segundos, a Figura 4.3 mostra o comportamento para as duas primeiras sequências de entrada. As setas coloridas acima do eixo x indicam quando cada composição finaliza a execução de sua sequência de entrada. Considerando que as duas últimas atividades do SciPhy consomem mais CPU, é possível identificar o momento em que começam, aos 78s no caso da composição *coarse-grained*, e quando terminam, aos 84s.

Analisando a execução completa da primeira sequência, a diferença entre a composição *fine-grained* e as outras composições é de aproximadamente 25%. A composição *fine-grained* apresenta um atraso devido à inicialização de um maior número de contêineres.

Os recursos do SDumont são suficientes para executar o SciPhy, mas, como a composição *fine-grained* inicia e para múltiplos contêineres em intervalos muito curtos, isso leva a pontos de vale, que representam um tempo ocioso da CPU para o SciPhy. Com o passar do tempo, esse tempo ocioso faz com que a composição *fine-*

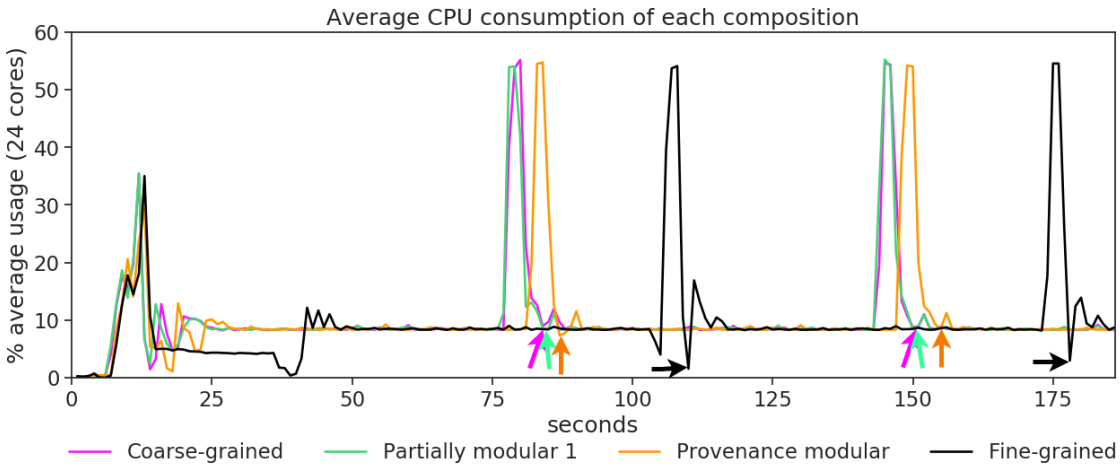


Figura 4.3: Consumo médio de CPU para as diferentes composições.

grained leve mais tempo de execução do que as outras composições. Um exemplo disso é que o MAFFT apresenta um uso de CPU mais elevado na composição *fine-grained* em comparação com as outras, porque o contêiner precisa ser reiniciado a cada execução.

Os tamanhos das imagens utilizadas no SDumont para as diferentes composições do SciPhy são os seguintes. A composição *coarse-grained* possui uma única imagem de 402,8MB. Todas as outras composições adotam um contêiner para os dados e o SGBD de proveniência, com tamanho de imagem de 85,5MB. Para a composição híbrida com dois contêineres (438MB): 352,5MB para a imagem do *workflow* e do serviço de proveniência; para a composição híbrida com três contêineres (615MB): 343,3MB para a imagem do *workflow* e 186,2MB para a imagem do serviço de proveniência; e para a composição *fine-grained* (612,5MB): 186,2MB para a imagem do serviço de proveniência, 45,2MB para o alinhamento de sequências múltiplas (MAFFT), 58,2MB para a conversão de alinhamento (ReadSeq), 187,8MB para a busca do modelo evolutivo (ModelGenerator) e 49,6MB para a geração da árvore (RAxML). Como o tamanho da imagem *coarse-grained* do SciPhy é pequeno, a composição *fine-grained* não se beneficiou da possibilidade de carregar imagens menores para executar cada atividade.

O nó do SDumont possui memória mais do que suficiente para essas imagens e com Singularity não permite gestão de recursos. Além disso, o Singularity possui técnicas de compactação muito eficientes para arquivos em contêineres e atinge um desempenho próximo ao *bare-metal* (execução na máquina física) durante a execução. Ao considerar o tamanho das imagens, a composição *coarse-grained* é a melhor para transferência das imagens, mas leva mais tempo para ser gerada em comparação com as outras composições.

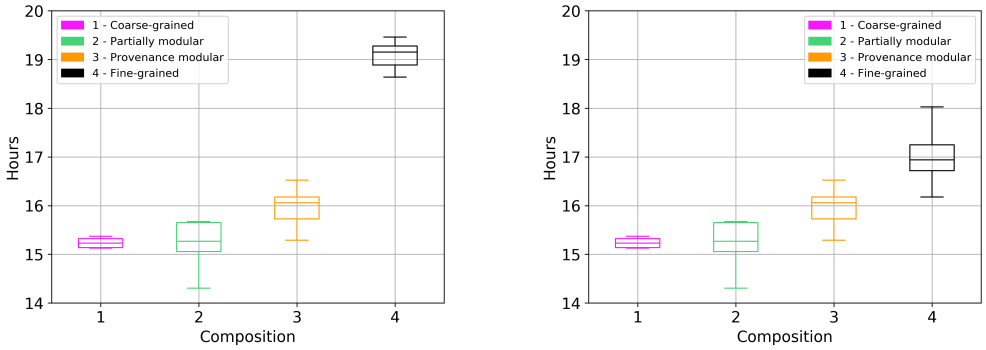
Como resultado dessa avaliação, podemos considerar que, quando o computa-

dor *hospdeiro* possui grande quantidade de recursos e poder de processamento em relação à demanda de recursos computacionais do *workflow*, a *sobrecarga* da composição *fine-grained* é desprezível e compensa suas vantagens de flexibilidade. Dado que, no SciPhy, é importante substituir atividades para o alinhamento múltiplo de sequências, a composição recomendada seria a composição *fine-grained*.

Avaliação quantitativa da Nuvem da AWS

O ambiente de nuvem nos fornece ambientes mais diversos, então exploramos algumas opções para testar a consistência dos resultados obtidos no SDumont. Primeiro, queríamos investigar se diminuir o poder computacional mudaria os resultados observados. Na AWS, utilizamos instâncias dos tipos c5.xlarge e c5.4xlarge, que possuem processador Intel Xeon Scalable (Cascade Lake) com clock turbo de 3,6 GHz. A instância c5.xlarge possui quatro vCPUs e 8 GiB de RAM, enquanto a c5.4xlarge possui 16 vCPUs e 32 GiB de RAM. A *engine* contêiner utilizada foi o Docker.

Inicialmente, executamos o SciPhy na instância c5.xlarge da AWS. A Figura 4.4a apresenta o tempo médio de execução em horas para os 200 arquivos de entrada multi-fasta, com base em dez execuções. A Figura 4.4a mostra que, em comparação com a composição de *coarse-grained*, a composição parcialmente modular teve tempo de execução semelhante, mas a composição modular de proveniência levou quase uma hora adicional, e a composição de *fine-grained* teve um tempo de execução mais de três horas maior em comparação com as outras composições. Os valores de desvio padrão (σ) também aumentaram, sendo o menor de 0,35 para a composição modular de proveniência e o maior de 0.65 para a composição parcialmente modular. Esse resultado pode indicar que a composição *fine-grained* não é uma opção adequada em uma VM c5.xlarge. Essa execução na c5.xlarge foi cerca de 10 horas mais lenta do que em SDumont, em parte devido ao Docker e à escassez de recursos.



(a) 200 arquivos multi-fasta ;

(b) 200 arquivos multi-fasta com gestão de recursos;

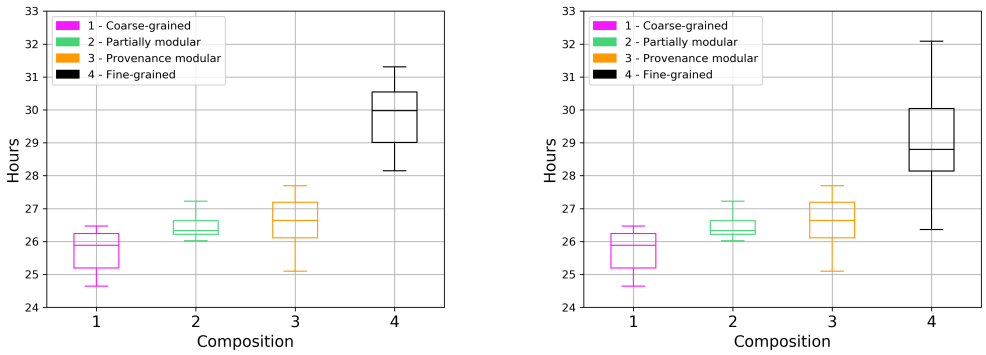
Figura 4.4: Tempo de execução do SciPhy em horas na nuvem da AWS, numa instância c5.xlarge (on-demand).

A configuração padrão do Docker permite que um contêiner utilize todos os recursos do computador conforme o escalonador de *kernel* da máquina hospedeira permita. Quando há múltiplos contêineres, eles começam a competir entre si por recursos. Para evitar essa competição, o Docker oferece uma configuração alternativa chamada "*resources constraints*"(restrições sobre recursos) que permite limitar o uso de memória e CPU pelos contêineres, ou seja, gestão de recursos. Portanto, executamos o SciPhy na instância c5.xlarge da AWS, restringindo seus recursos considerando o perfil das imagens dos contêineres. Nos experimentos, configuramos as restrições de recursos para todas as imagens do contêiner SciPhy, onde todas as imagens foram executadas com os recursos mínimos exigidos pela atividades, exceto para ModelGenerator e RAxML. A Figura 4.4b apresenta o tempo médio de execução em horas para 200 arquivos de entrada multi-fasta, mostrando as configurações com recursos restringidos.

Como esperado, a composição de *coarse-grained* continua a apresentar o melhor desempenho, pois não há contêineres competindo pelos mesmos recursos, e seu contêiner pode explorar os recursos como se estivesse executando diretamente na máquina hospedeira. Como a composição parcialmente modular é quase a mesma sem as restrições de recursos, essas restrições não geraram impacto. O mesmo pode ser dito sobre a composição modular de proveniência, que mostra que os componentes de proveniência usaram uma quantidade fixa de recursos, mesmo sem as restrições. A Figura 4.4b mostra que restringir recursos na instância c5.xlarge proporcionou uma melhoria significativa na composição *fine-grained*, em torno de 2 horas. Apesar das melhorias, quando o desempenho é uma questão importante, a composição *fine-grained* ainda não parece ser recomendada para esse tipo de instância devido à sobrecarga introduzida. Assim como no SDumont, a composição *fine-grained* gasta um tempo adicional trocando entre contêineres, nesse ambiente de nuvem, a troca não é tão rápida. No entanto, quando os critérios de qualidade de reutilização e substituição são prevalentes, como no caso do SciPhy, ter mais imagens ainda permite que os usuários explorem mais as diferentes opções de gestão de recursos do Docker para melhorar o tempo de execução e os custos associados à composição *fine-grained*.

Também executamos o SciPhy com 400 arquivos para comparar o desempenho geral com os experimentos realizados em SDumont. Os resultados estão mostrados na Figura 4.5a, e a composição de *fine-grained* novamente teve o pior desempenho. De modo inesperado, a restrição de recursos (Figura 4.5b) foi contraproducente, o que parece indicar as limitações das ações de gestão de recursos. Na Seção 4.1.1 (Figura 4.3), destacamos o atraso na execução que a composição de *fine-grained* tem devido à necessidade de iniciar e parar imagens de contêineres para cada execução. Esse atraso se acumula ao longo da execução. No supercomputador SDumont, isso é

desprezível. Em um cenário de escassez de recursos, como o c5.xlarge, esse impacto ainda pode ser insignificante para uma certa quantidade de iterações. No entanto, à medida que o número de arquivos, e portanto o número de iterações, aumenta, o atraso se acumula a ponto de a restrição de recursos se tornar ineficaz e causar mais atraso na execução causado por outras atividades e componentes que estão a executar com o mínimo de recursos.



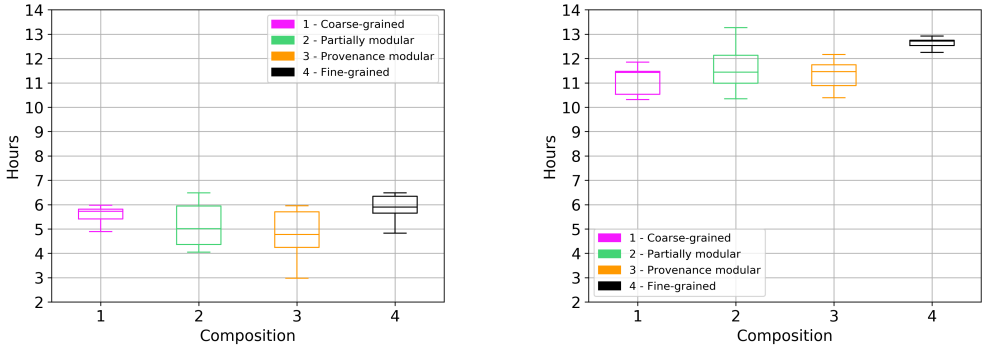
(a) 400 arquivos multi-fasta;

(b) 400 arquivos multi-fasta com gestão de recursos;

Figura 4.5: Tempo de execução do SciPhy em horas consumindo 400 arquivos na nuvem da AWS, numa instância c5.xlarge (*on-demand*).

Para avaliar os benefícios de ter quatro vezes mais poder computacional na AWS, os mesmos experimentos foram realizados com 200 e 400 arquivos de entrada em uma instância AWS mais potente, c5.4xlarge (Figuras 4.6a e 4.6b). Observamos uma melhoria significativa com a c5.4xlarge, especialmente na composição *fine-grained*, que se tornou competitiva com todas as outras composições. Neste caso, a restrição de recursos não foi necessária. Todos apresentaram um $\sigma \leq 1$ e $\sigma \geq 0.34$ para o tempo de execução, sugerindo execuções mais instáveis em comparação com o SDumont, o que era esperado. Embora a c5.4xlarge tenha menos memória e núcleos de CPU do que o SDumont, o *clock* da CPU é mais rápido, o que acreditamos mitigar o custo de iniciar e parar contêineres.

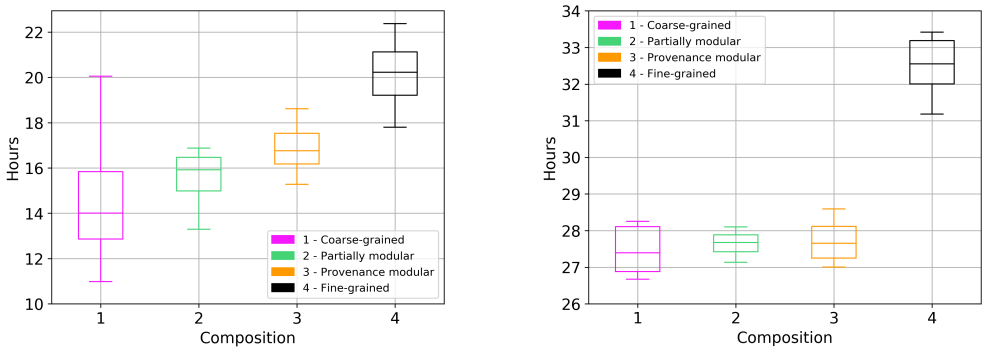
Por fim, também realizamos os mesmos experimentos nas instâncias c5.xlarge e c5.4xlarge do tipo *spot*, que tem preços mais acessíveis e disponibilidade variáveis [70]. Os resultados para 200 e 400 arquivos em ambas as instâncias são apresentados nas Figuras 4.7a, 4.7b, 4.8a e 4.8b. Os resultados são muito semelhantes aos das instâncias *on-demand*, no entanto, como esperado, apresentam valores de σ maiores, na maioria dos casos, $\sigma \leq 0.95$. A composição *coarse-grained* teve o maior valor de $\sigma \approx 2.78$ na c5.xlarge com 200 arquivos (Figura 4.7a). Comparando com a execução com 400 arquivos no mesmo tipo de instância, podemos observar que a instabilidade presente nas máquinas *spot* é mais evidente em execuções mais curtas. No entanto, o desempenho geral é impactado em todos os casos, aumentando o



(a) 200 arquivos multi-fasta; (b) 400 arquivos multi-fasta;

Figura 4.6: Tempo total de execução do SciPhy consumindo 200 arquivos multi-fasta na VM c5.4xlarge (*on-demand*) da nuvem AWS.

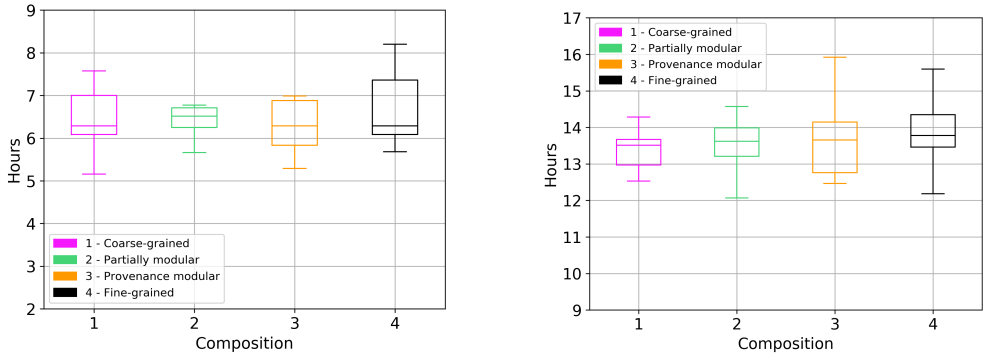
tempo de execução em cerca de uma hora. As execuções na c5.4xlarge novamente mostram quase nenhuma diferença entre as composições, assim como nas instancias c5.4xlarge *on-demand* e no SDumont, mostrando que em ambientes com recursos suficientes, as diferenças entre as composições são desprezíveis, mesmo com maior instabilidade.



(a) 200 arquivos multi-fasta na instância c5.xlarge; (b) 400 arquivos multi-fasta na instância c5.xlarge;

Figura 4.7: Tempo de execução do SciPhy em horas na nuvem AWS nas instâncias *spot* c5.xlarge.

Os tamanhos das imagens usadas na AWS para as diferentes composições do SciPhy são os seguintes: a composição *coarse-grained* tem 1,1 GB para sua única imagem . Todas as outras composições adotam um contêiner para os dados e SGBD de proveniência, com um tamanho de imagem de 380 MB. Para a composição híbrida com dois contêineres (1,3 GB): 956 MB para a imagem do *workflow* e serviço de proveniência; para a composição híbrida com três contêineres (1,4 GB): 896 MB para a imagem do *workflow* e 542 MB para a imagem do serviço de proveniência; e para a composição *fine-grained* (1,2 GB): 542 MB para a imagem do serviço de proveniência, 55,4 MB para alinhamento de sequências múltiplas (MAFFT), 60 MB para conversão de alinhamento (ReadSeq), 192,3 MB para busca de modelo evolutivo



(a) 200 arquivos multi-fasta na instância;

(b) 400 arquivos multi-fasta na instância c5.4xlarge;

Figura 4.8: Tempo de execução do SciPhy em horas na nuvem AWS nas instâncias *spot* c5.4xlarge.

(ModelGenerator) e 67,5 MB para geração de árvore filogenética (RAxML). Para a composição *fine-grained*, foi possível usar imagens já fornecidas no Docker Hub, que possuem versões minimizadas disponíveis.

As imagens Docker usadas nas instâncias da AWS são maiores do que as imagens Singularity utilizadas no SDumont. O Docker comprime arquivos dentro de camadas, resultando em imagens maiores. No geral, o Docker não é tão eficiente quanto o Singularity, mas é a *engine* de contêiner *de-facto*, e nossa intenção foi avaliar um cenário mais comum para *workflow* intensivos em dados.

Os experimentos na AWS com o SciPhy mostram que, apesar das vantagens qualitativas da composição de *fine-grained*, certamente ela não é a melhor escolha com poder de processamento limitado, como no caso da instância c5.xlarge, mesmo ao gerenciar os recursos da nuvem. Além disso, mesmo com a composição de *coarse-grained*, o tempo total de execução é muito alto, e à medida que mais arquivos de entrada são utilizados, a c5.xlarge pode não ser uma alternativa viável.

No geral, para o SciPhy, podemos perceber que a melhor composição depende de múltiplos fatores e que a composição do contêiner não deve ser pré-definida. Aspectos qualitativos sempre interferirão na escolha, mas a especificação do ambiente e a estabilidade devem ser consideradas. Esses experimentos mostraram que, quando há um quantidade abundante de recursos computacionais disponíveis, a composição pode ser baseada em aspectos qualitativos.

4.1.2 Estudo de Caso: Gerando imagens astronômicas com o *Workflow* Montage

Como segundo experimento, executamos outro *workflow* intensivo em dados: o Montage. O Montage [75] é um *workflow* projetado para criar mosaicos a partir de múl-

tiplas imagens do céu. Ele é composto por oito atividades, ilustradas na Figura 4.9, sendo cada uma associada a um programa do pacote Montage[1]: (i) `mProjectPP` - projeta imagens em uma escala definida; (ii) `mDiffFit` - calcula e ajusta diferenças entre imagens astronômicas; (iii) `mConcatFit` - combina múltiplas imagens em uma única ; (iv) `mBgModel` - modela e corrige diferenças de fundo em imagens sobrepostas; (v) `mBackground` - remove o fundo das imagens ; (vi) `mImgtbl` - extrai metadados para a geração do mosaico; (vii) `mAdd` - compõe as imagens para formar o mosaico; e (viii) `mViewer` - gera uma visualização do mosaico em PNG ou JPEG.

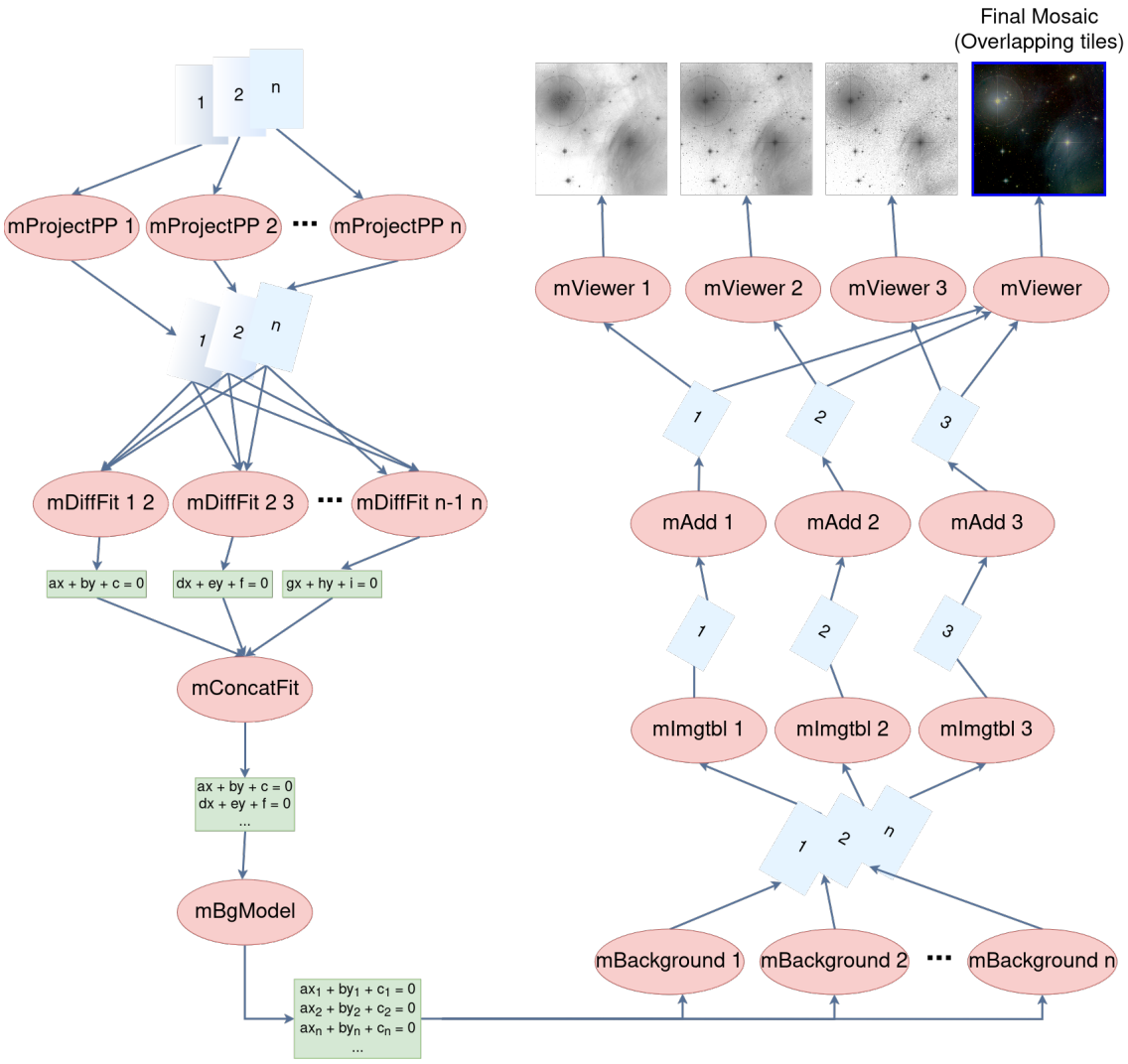


Figura 4.9: *Workflow* Montage, adaptado de [39].

O número total de atividades no Montage varia conforme à área de cobertura do céu. Apresentamos a quantidade de atividades associadas a cada grau executado nesta tese na Tabela 4.1. Embora o Montage seja um *workflow* intensivo em dados, seu funcionamento difere do SciPhy. A quantidade de atividades não cresce de forma diretamente proporcional ao grau da área de cobertura do céu, e o aumento

[1] `http://montage.ipac.caltech.edu/`

no número de atividades isoladas não é linear. Em alguns casos, como na atividade `mConcatFit`, o número de execuções é fixo, independentemente da área de cobertura, mas a atividade processa uma quantidade muito maior de entradas para cada grau, podendo gerar gargalos. Por fim, o Montage não realiza repetições de atividades com base nas entradas, o que também o distingue do SciPhy.

Tabela 4.1: Número de atividades do Montage de acordo com a área de cobertura do céu.

	Área de cobertura do céu		
Atividade	**0.5 grau**	**1.0 grau**	**2.0 graus**
`mProjectPP`	12	48	192
`mDiffFit`	18	360	6048
`mConcatFit`	3	3	3
`mBgModel`	3	3	3
`mBackground`	12	48	192
`mImgtbl`	3	3	3
`mAdd`	3	3	3
`mViewer`	4	4	4
Total	58	472	6448

Com base nestas especificidades do *workflow*, uma análise qualitativa do Montage sugeriria uma composição híbrida ou *coarse-grained* pois o Montage já é disponibilizado na forma de um pacote, baseados em C, que podem ser instalados via gerenciadores de pacote como *apt*. Por outro lado, observando que as atividades do Montage tem demandas paralelas que diferem significativamente entre si, a composição *fine-grained* poderia auxiliar na gestão de recursos, além de permitir a execução do Montage em máquinas diferentes. Os experimentos de desempenho têm como objetivo adicionar mais dados para encontrar a melhor composição de contêiner a ser executados com a `ProvDeploy`.

Composições de contêiner

O Montage é disponibilizado como um pacote de ferramentas e depende principalmente de bibliotecas em C/C++ para sua compilação. A primeira composição de contêiner foi *coarse-grained*, englobando todas as atividades *workflow*. Essa composição permite a repetição e o compartilhamento do *workflow*.

Para a composição *fine-grained*, geramos um contêiner para cada atividade do Montage utilizada, resultando em oito contêineres. Como o Montage é disponibilizado como um pacote de ferramentas, pode parecer contraintuitivo dividi-lo em programas isolados. No entanto, ao fazer isso, é possível executar suas atividades em diferentes *sites* ou nós e alocar quantidades distintas de recursos para cada atividade. Por exemplo, `mProjectPP` é uma atividade intensiva em CPU no Montage,

podendo ser executada com mais núcleos de CPU que as demais atividades.

Vale destacar que a geração de múltiplas imagens, nesse caso, aumenta o armazenamento total necessário. Contudo, como as imagens utilizadas nesses experimentos foram geradas no Docker, elas puderam compartilhar camadas, o que acaba reduzindo o espaço necessário para armazenamento das imagens. Nosso principal objetivo com essas composições em um *workflow* intensivo em dados foi observar se havia diferenças no comportamento em memória entre elas.

Chegamos a testar uma composição híbrida com Montage, onde agrupávamos as atividades que executam em quantidades fixas (*e.g.*, `mConcatFit`, `mImgTbl` e *etc*), no entanto, não identificamos benefícios qualitativos ou quantitativos que já não fossem observados com a composição *fine-grained*.

Avaliação quantitativa na GCP

Executamos as duas composições em uma instância *n2-highcpu-32*, uma instância de CPU na nuvem da GCP que possui processador Intel Xeon Cascade Lake Gold 6268CL 2.8 - 3.9 GHz com 32 vCPUs e 32 GB de RAM, onde utilizamos Docker. Nesse caso, o Docker foi usado para simular o cenário mais comum em um ambiente de nuvem. Na Figura 4.10, apresentamos o tempo médio de execução, em segundos, de seis execuções para cada composição de conteinerização, considerando áreas de cobertura de 0.5, 1.0 e 2.0 graus utilizando os 32 núcleos e 32 GB. A quantidade de memória definida para instância foi assim feita por ser a quantidade mínima para a execução do Montage com 2.0 graus.

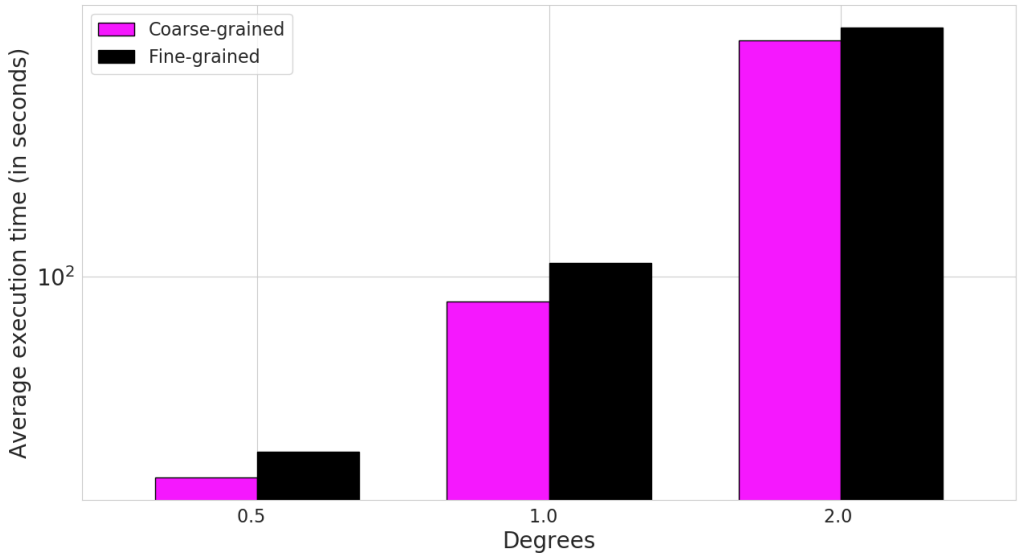


Figura 4.10: Tempo médio de execução do Montage

Para 0.5 grau, a diferença entre as composições *coarse-grained* e *fine-grained* é de aproximadamente 17%. Para 1.0 grau, essa diferença é de cerca de 27%. Entretanto,

para a cobertura de 2.0 graus, que é a mais intensiva computacionalmente, o aumento entre *fine-grained* e *coarse-grained* é de apenas 8%.

A Figura 4.10 utiliza uma escala logarítmica para evidenciar que os três experimentos parecem indicar uma sobrecarga fixa para a composição *fine-grained*, que se dilui ao longo do tempo. Os valores absolutos médios para cada grau são apresentados na Tabela 4.2, onde também exibimos o desvio padrão (σ). Esses resultados mostram que a diferença entre a composição *fine-grained* e *coarse-grained* em 2,0 graus é muito pequena, não havendo uma diferença significativa entre as duas composições considerando o desvio padrão.

Tabela 4.2: Tempo médio de execução em segundos e desvio padrão para diferentes composições no Montage

Composição	**Grau**					
	0.5		1.0		2.0	
	x	σ	x	σ	x	σ
Coarse-grained	28.22	0.38	85.23	1.35	441.39	8.10
Fine-grained	33.20	2.87	108.89	0.72	479.87	16.04

Usando os 32 núcleos da instância, o consumo médio de CPU do Montage com 0.5 graus foi em torno de 11% para ambas as composições, com 1.0 grau foi de cerca de 28% para a composição *coarse-grained*, e 21% para a composição *fine-grained*, e em torno de 23% para a composição *coarse-grained*, e 20% para a composição *fine-grained* com 2,0 graus. A atividade `mProjectPP`, que é a primeira e intensiva em CPU, utiliza 100% de todos os núcleos, exceto em 0,5 graus, onde processa apenas 12 atividades `mProjectPP`, cada uma alocada em um núcleo, deixando 20 núcleos da máquina ociosos. Cada núcleo de CPU executa uma atividade `mProjectPP`, e como `mDiffFit` e outras atividades têm dependências de dados com `mProjectPP`, esta deve terminar a execução para que outras atividades possam ser executadas. Assim como no experimento anterior, mostramos a atividade de CPU de cada composição em uma execução de 2.0 graus na Figura 4.11, onde podemos observar que todas as composições apresentam consumo de CPU semelhante durante a execução. Inicialmente, esperávamos que a composição *fine-grained* fosse mais exigente em termos de CPU, devido ao início e à parada de mais contêineres. No entanto, como cada atividade leva tempo para ser executada, os contêineres compartilham camadas e são iniciados e parados apenas uma vez para cada atividade, e não há escassez de recursos, o impacto de ter mais contêineres não foi significativo na CPU. No entanto, podemos observar que a composição *coarse-grained* é executada sem pausas causadas pelos contêineres o que se reflete em seu tempo total.

Nas primeiras tentativas de execução do Montage, utilizando a composição *coarse-grained* com 2.0 graus, enfrentamos problemas de falta de memória. Para

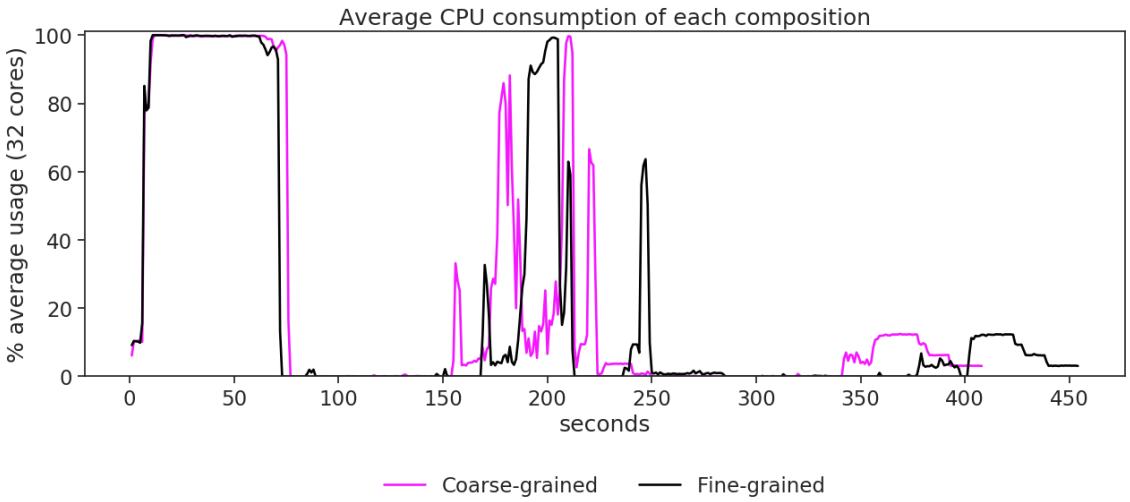


Figura 4.11: Consumo médio de CPU do Montage com 2.0 graus.

resolver esse problema, passamos a utilizar volumes em todas as composições, de modo que a composição *coarse-grained* encapsula apenas o pacote de programas do Montage, deixando os dados fora dos contêineres. No entanto, ao reduzir a quantidade de memória RAM para testar os limites de execução, percebemos que cada grau de cobertura tinha um valor mínimo necessário. Quando essa quantidade mínima não era disponibilizada, parte das atividades disparadas era interrompida, resultando em falhas na execução do *workflow*. A composição mais afetada por esse problema foi a *coarse-grained*, mesmo após a adoção de volumes.

Após identificar a quantidade de memória suficiente para a execução bem-sucedida, decidimos examinar o comportamento das composições também em relação ao uso de memória. Para 0.5 graus, o consumo médio de memória foi de cerca de 6% em ambas as composições. Em 1.0 grau, o consumo médio foi de aproximadamente 5% para a composição *fine-grained* e 10% para a *coarse-grained*. Já em 2.0 graus, o consumo médio de memória chegou a 45% na *coarse-grained* e 36% na *fine-grained*. Esse comportamento pode ser observado na Figura 4.12.

Durante a execução, enquanto os contêineres *fine-grained* são parados e reiniciados ao longo do da execução, o contêiner *coarse-grained* permanece ativo até o término da execução, sem pausas. Essa característica, combinada com seu tamanho maior, faz com que a composição *coarse-grained* consuma mais memória, pois acaba se expandido livremente e de forma desnecessária.

O tempo adicional na composição *fine-grained* pode ser identificado nos vales mais longo nos intervalos de 150 a 175 segundos, 210 a 240 segundos e 370 a 420 segundos, que se sobrepõem a momentos em que o *workflow* aguarda a finalização de atividades (*e.g.*, `mProjectPP`). Na Figura 4.12, apresentamos o uso de memória em uma única execução de cada composição. Nessa figura, é possível observar pontos em que a composição *coarse-grained* requer memória extra (entre 220 e 400 segundos),

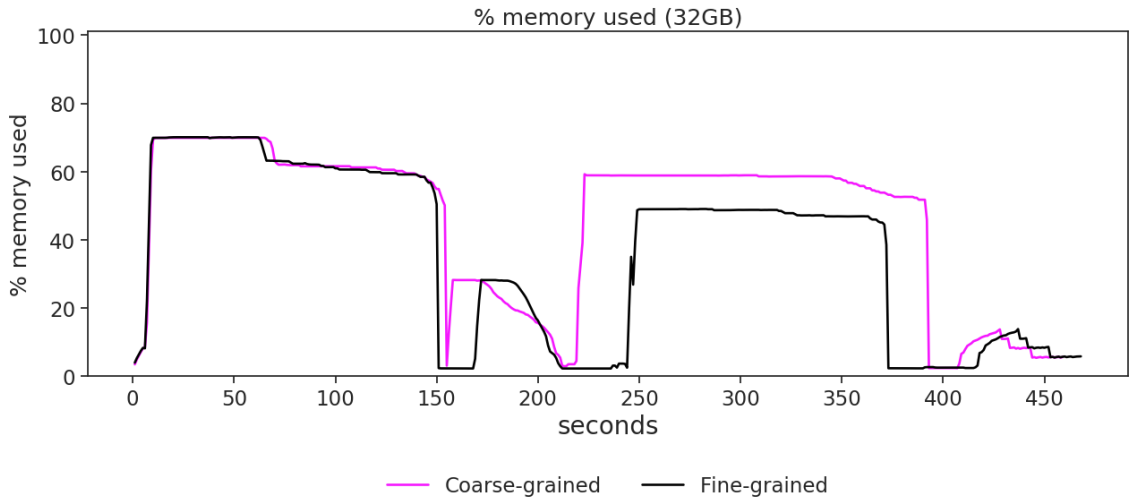


Figura 4.12: Uso de memória de cada composição do Montage com 2.0 graus.

correspondente às atividades `mBackground`, `mImgTbl` e `mAdd`.

Na Tabela 4.3, apresentamos os tamanhos das imagens utilizadas nos experimentos com o *Montage*. Embora a composição *coarse-grained* tenha um tamanho total ligeiramente menor, transferir e gerenciar essas imagens de contêiner pode ser desafiador. Caso um contêiner seja incompatível com um novo ambiente, perde-se o acesso aos seus dados e funcionalidades. Por outro lado, a composição *fine-grained* possui um tamanho total maior para transferência, mas esse tamanho está distribuído em imagens que compartilham camadas por serem Docker, tem fácil transferência e substituição. Assim, quando uma é descarregada, as próximas só vem com as camadas diferentes e uma vez armazenadas no mesmo repositório local, não requerem a mesma quantidade de espaço que suas versões isoladas. No caso das imagens utilizadas, elas ocuparam cerca de 3,7 GB no disco da instância em nuvem e não $\approx$ 6 GB como suas versões isoladas apresentadas na tabela 4.3 .

Tabela 4.3: Volumes de dados e contêineres para diferentes composições e graus no Montage.

Grau	**Composição**	**Dados Gerados**	**Imagem de contêiner (tamanho da imagem)**	**Tamanho total**
0.5	Coarse-grained	3 GB	Montage (1.03 GB)	4.03 GB
1.0		13 GB		14.03 GB
2.0		49 GB		50.03 GB
0.5	Fine-grained	3 GB	mProject (844.8 MB), mDiffFit (848.8 MB), mBackground (844.1 MB), mConcatFit (839.4 MB), mBgModel (839.4 MB), mImgtbl (844.2 MB), mAdd (844.1 MB), mViewer (845.2 MB)	9 GB
1.0		13 GB		19 GB
2.0		49 GB		55 GB

Em resumo, esse experimento acabou por mostrar embora seja possível identificar diferenças entre as composições, essas diferenças não parecem ser significativas

na escolha de uma composição mais satisfatória. O desempenho também depende das características do *workflow* e do ambiente, sendo essencial conhecê-las para fins de análise e reutilização dos contêineres. Trabalhos como OLAYA *et al.* [67], apontam que a composição *fine-grained* geralmente apresenta um desempenho inferior à composição *coarse-grained* e tende a ter um custo computacional mais elevado. Ainda assim, a composição *fine-grained* é uma opção melhor para rastreabilidade e reprodutibilidade, enquanto a *coarse-grained* se destaca em desempenho e para verificação e repetição de resultados.

Além dessas composições, existem diversas outras opções que podem ser mais adequadas para diferentes *workflows*. Por isso, é importante oferecer suporte a múltiplas composições, garantindo a coleta e a rastreabilidade da proveniência.

4.1.3 Estudo de Caso: Aprendizado de máquina científico com a DenseED

O uso de contêineres para *workflows* de aprendizado de máquina (AM) facilita sua implantação em diferentes ambientes computacionais. Vários dos critérios qualitativos definidos na Seção 3.1 têm relevância significativa no contexto de *workflows* de AM. LI *et al.* [50] destacam a importância da reutilização e da reprodutibilidade para *workflows* de AM. A captura de proveniência também é necessária para o compartilhamento de *workflows* e para fomentar a colaboração na comunidade científica. A gestão de recursos é outro critério relevante, pois a execução de *workflows* de AM de forma eficiente em diversos ambientes (*e.g.*, supercomputadores e nuvem) contribui para a portabilidade.

A substituição de componentes também é muito importante. Por exemplo, para executar o TensorFlow em GPUs, é necessário combinar versões do Python, do compilador C, do Bazel, do CUDA e do cuDNN com o dispositivo de GPU disponível. Para evitar todo esse trabalho, o TensorFlow já é disponibilizado oficialmente por meio de contêineres. O uso de uma imagem de contêiner TensorFlow para GPUs, no Docker e no Singularity, requer apenas habilitar o uso da GPU por meio de *flags*. Caso a imagem utilizada em um *workflow* de AM seja incompatível com a GPU disponível ao mover o *workflow* entre ambientes, em vez de criar uma nova imagem, o usuário pode simplesmente explorar outras imagens de registros públicos, como NGC, Docker e Binder.

O DenseED é um *workflow* de aprendizado de máquina científico proposto por FREITAS *et al.* [27]. A arquitetura do DenseED é baseada em uma rede neural convolucional (CNN) guiada por física, conforme definido em ZHU e ZABARAS [97]. A rede é composta por camadas convolucionais e blocos densos, seguindo uma estrutura de rede neural do tipo *encoder-decoder* para lidar com a potencial alta

dimensionalidade de entradas e saídas. O DenseED utiliza a CNN guiada por física como modelo substitutivo para os cálculos da Migração Reversa no Tempo (*Reverse Time Migration* -RTM) com o objetivo de possibilitar a quantificação de incertezas [27] na RTM. Resolver as equações da RTM é uma tarefa intensiva em CPU e demorada. Para quantificar as incertezas, é necessário resolver uma RTM para cada distribuição de probabilidade, o que é inviável. O DenseED busca substituir o cálculo das equações da RTM por um modelo treinado guiado por física. A arquitetura do DenseED está ilustrada na Figura 4.13. Os conjuntos de dados de entrada são amostras de velocidades, e as saídas são imagens sísmicas com incertezas.

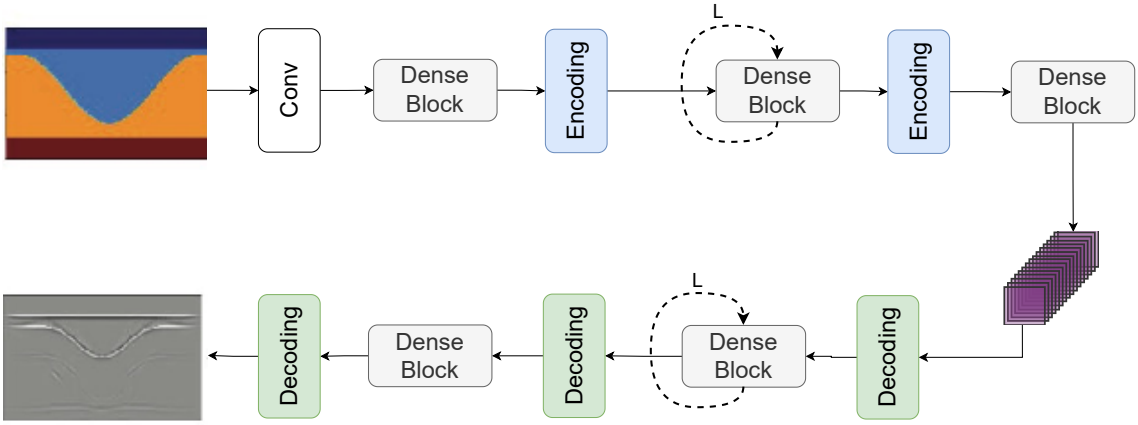


Figura 4.13: Arquitetura de DenseED, adaptado de FREITAS *et al.* [27].

O DenseED é intensivo em uso de CPU, pois processa um volume substancial de entradas composto por 200.000 campos de velocidade. Deste conjunto, seleciona aleatoriamente 10%, dividindo-os para treinamento e teste, calcula métricas como desvio padrão e média, salva os resultados em arquivos e prossegue para construir, treinar e testar o modelo. Nosso objetivo principal foi avaliar e compreender o comportamento do DenseED quando utilizado em diferentes composições de contêineres com proveniência em um supercomputador, bem como analisar o custo-benefício à longo prazo entre essas composições.

Vale ressaltar que a versão do DenseED utilizada neste estudo é uma variante reduzida, treinada com metade do número mínimo de épocas necessárias para resultados satisfatórios. Esse ajuste intencional foi feito para acelerar as observações do comportamento das composições. Além disso, o DenseED, que é feito para ambientes de PAD, foi executado no SDumont.

Composições de contêiner

Embora o *workflow* do DenseED envolva o cálculo de métricas sobre o conjunto de dados de entrada antes de iniciar a compilação do modelo, as atividades de codificação e decodificação durante o treinamento da CNN, ele ainda pode ser percebido

como uma única atividade monolítica. Todas essas atividades específicas dentro do treinamento da CNN orientada por Física dependem do TensorFlow e suas dependências. Como resultado, nossa avaliação inclui três composições de contêineres: *coarse-grained*, parcial (*partial modular*) e modular de proveniência (*provenance modular*), uma vez que a composição *fine-grained* não é aplicável.

A composição *coarse-grained* consiste em um único contêiner que engloba o *workflow* do DenseED, o serviço de proveniência, o SGBD para o banco de dados de proveniência e os dados gerados. Essa abordagem suporta de forma eficaz o compartilhamento do *workflow* e dos resultados do DenseED, facilitando transições de *frameworks* como o Jupyter para ambientes de PAD. Além disso, essa composição permite a repetição de experimentos com captura de dados de proveniência, pois suporta execução em uma única etapa.

A composição de contêineres parcial modular utiliza duas imagens de contêiner: uma para o *workflow* do DenseED e o serviço de proveniência, e a outra para o banco de dados de proveniência e os dados. Essa composição aprimora a flexibilidade, especialmente para substituir as amostras de velocidade na entrada. Ao adotar essa composição, o contêiner DenseED evita o compartilhamento de recursos com o SGBD de proveniência, que recebe solicitações a cada época de treinamento. Além disso, essa composição permite aplicar ações de gestão de recursos para os contêineres DenseED e SGBD e/ou execução em diferentes nós/ambientes. Embora não suporte execução em uma única etapa, ela reduz o tempo de geração de contêiner, permitindo o reaproveitamento da imagem do contêiner MonetDB.

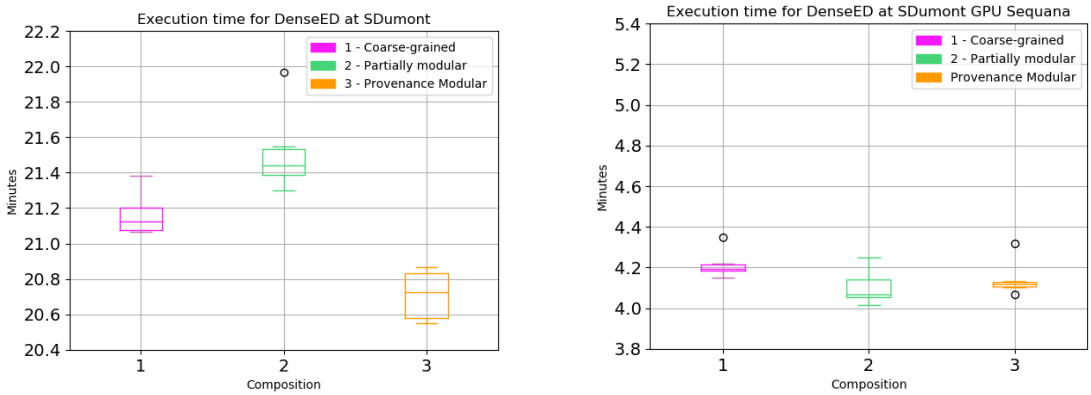
Cada composição descrita favorece cenários diferentes; no entanto, também devemos investigar se as diferenças entre elas adicionam mais tempo à execução, a fim de escolher a composição mais adequada para executar o *workflow*.

Avaliação quantitativa no Supercomputador SDumont

O DenseED foi implantado com o `ProvDeploy` no SDumont utilizando dois nós computacionais diferentes: um nó CPU e um nó GPU. O nó CPU possui dois processadores Intel Xeon E5-2695v2 Ivy Bridge de 2,4 GHz, 24 núcleos (12 por CPU) e 64 GB de RAM DDR3. O nó GPU faz parte da partição expandida BullSequana X do SDumont, equipado com dois processadores Intel Xeon Skylake de 2,1 GHz, 48 núcleos (24 por CPU), 384 GB de RAM e quatro GPUs NVIDIA Volta V100. Em ambos os casos, utilizamos o sistema operacional Linux RedHat 7.6, o motor de contêiner Singularity 3.8, e para perfilagem, a biblioteca sysstat 12.

Nas Figuras 4.14a e 4.14b, apresentamos o tempo de execução em minutos para dez execuções de cada composição de contêiner com $L = 4$ e $K = 24$, utilizando 100 épocas em um nó CPU. Em todas as composições, $\sigma \leq 0.9$. No nó de CPU, o tempo médio de processamento por época é de aproximadamente 22 segundos, re-

sultando em um intervalo significativo entre as chamadas de proveniência para cada época. Observa-se uma diferença marginal nos tempos de execução para as composições *coarse-grained*, parcial modular e modular de proveniência. Curiosamente, no caso da composição modular de proveniência, as chamadas de proveniência entre os contêineres não contribuíram para um aumento no tempo de execução total. Ela foi executada cerca de meia-minuto mais rápido do que as outras composições de contêiner.



(a) Execução em CPU; (b) Execução em GPU;

Figura 4.14: (a) Tempo de execução do DenseED entre 20,4 e 22 minutos; (b) Tempo de execução do DenseED utilizando GPUs entre 3,8 e 5,4 minutos.

Para examinar as diferenças entre as composições de contêiner na CPU, a média de consumo de CPU nos 24 núcleos para cada composição no nó de CPU do SDumont foi registrada por segundo. Os primeiros 180 segundos de uma execução de cada composição estão representados na Figura 4.15.

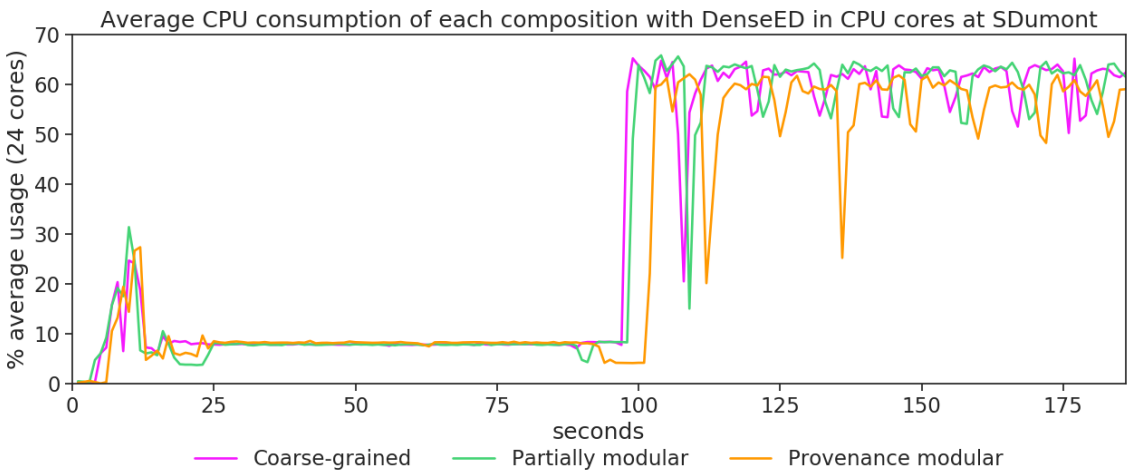


Figura 4.15: Consumo médio de CPU do DenseED.

O DenseED processa o conjunto de dados de entrada selecionando pontos aleatórios para treinamento e teste. Ele calcula métricas, como média e desvio padrão,

sobre esses dados e salva os resultados em diferentes arquivos, essas atividades podem ser vistas na Figura 4.15 no intervalo de ≈10 a ≈100 segundos. Posteriormente, o DenseED prossegue para treinar e validar a rede neural, realiza a fase de validação e armazena as previsões junto com o *Root Mean Square Error* (RMSE) tanto para as fases de treinamento quanto de validação. As etapas de treinamento, teste e validação são intensivas em CPU, resultando em 100% de uso da CPU em diferentes núcleos, enquanto o consumo geral da CPU raramente cai para 0%. O consumo médio de CPU para o DenseED é de 55% para a composição *coarse-grained*, 56% para a composição modular parcial e 52% para a composição modular de proveniência.

No DenseED, a composição modular de proveniência evita a sobrecarga de iniciar e parar contêineres com frequência em intervalos curtos. O processo de treinamento começa em aproximadamente 100 segundos. Embora a composição modular de proveniência incorra em um pequeno atraso, seu consumo de CPU é ligeiramente menor em comparação com as composições *coarse-grained* e modular parcial. Assim como no caso do Montage, parece que contêineres que executam tarefas distintas tendem a consumir mais recursos. Ao mesmo tem que na composição modular de proveniência, os contêineres sobrecarregam a CPU com múltiplas atividades de um mesmo contêiner. Além disso, quando os contêineres estão inativos na CPU, mais recursos ficam disponíveis para os contêineres ativos. Considerando os benefícios qualitativos de reutilização de imagens, manutenção de *workflows* e gestão de recursos em *workflows* de AM, a composição modular de proveniência se destaca como uma alternativa atraente para o DenseED em CPUs.

Em seguida, executamos o DenseED em GPUs para investigar se a redução do tempo decorrido para o processamento de cada época levaria a diferenças no tempo total de execução para as mesmas três composições de contêiner. Os resultados para o nó de GPU são apresentados na Figura 4.14b. O tempo médio decorrido para processar cada época foi reduzido para cerca de 2 segundos. Curiosamente, o comportamento não é consistente com a execução no nó de CPU, indicando que as chamadas de proveniência entre contêineres impactam o tempo total de execução. Embora a composição modular com proveniência tenha sido a mais rápida nas CPUs, seu tempo de execução nas GPUs é semelhante ao das outras composições.

Ao analisar as Figuras 4.14a, 4.14b e 4.15, observa-se uma variação aparentemente desprezível nos tempos de execução entre as diferentes composições. Consequentemente, realizamos testes de hipótese para CPU e GPU. No cenário de GPU, ao executar o teste ANOVA unidirecional com $\alpha = 5\%$, em todos os casos, p-value $> \alpha$, aceitando a hipótese nula. Portanto, não há diferenças significativas entre as composições apresentadas.

Já nas CPUs, ao realizar o teste ANOVA unidirecional, identificou-se uma diferença significativa em $\overline{x}$ das três composições. A análise *post-hoc* com correção de

Bonferroni, utilizando $\alpha = 1,66\%$, resultou em p-value $< \alpha$, rejeitando a hipótese nula. Assim, há diferenças significativas entre as composições apresentadas. Notavelmente, a composição modular parcial surge como a mais demorada nas CPUs, o que pode ser atribuído a gargalos na execução, onde um contêiner pode ser sobrecarregado com tarefas distintas, como executar DenseED e recuperar dados, ou lidar com chamadas de proveniência, persistência e recuperação de dados.

Embora haja diferença nos resultados das CPUs, podemos considerar a diferença, tanto nas CPUs quanto nas GPUs, desprezível. Isso pode indicar que as diferentes composições não afetam significativamente o desempenho, proporcionando maior autonomia na escolha da composição de acordo com as necessidades do usuário.

Em relação aos tamanhos de imagens usados no SDumont para diferentes composições no DenseED, a composição *coarse-grained* utiliza uma única imagem de 6,6 GB. Todas as outras composições adotam um contêiner para os dados e o banco de dados de proveniência, com um tamanho de imagem de 508,3 MB. A composição modular parcial (6,7 GB) é composta por 6,2 GB para a imagem do *workflow* e serviço de proveniência. Já a composição modular com proveniência (6,8 GB) consiste em 6,1 GB para a imagem do *workflow* e 201,3 MB para a imagem do serviço de proveniência.

A imagem do TensorFlow utilizada é a TensorFlow Release 22.05, com TensorFlow 2.8 e Python 3.8, proveniente do NGC, compatível com GPUs Volta V100. Embora essa imagem tenha aproximadamente 6 GB, ela é otimizada com o NVIDIA HPC Toolkit, economizando um tempo considerável na geração e teste de imagens que precisam combinar uma pilha de software complexa. Por conta do tamanho dessa imagem que foi utilizada em todas as composições de contêiner aplicadas sobre a DenseED não é possível apontar grandes mudanças entre as composições com relação a quantidade de armazenamento necessária.

4.2 Avaliação de consultas usando o modelo de proveniência de *workflow* e de contêiner

Consultar a proveniência de contêineres ajuda a escolher a melhor composição e ajustar decisões de conteinerização de acordo com o ambiente de execução alvo. A `ProvDeploy` através do modelo de proveniência de contêiner suporta todas as consultas listadas na Tabela 3.2, e esta seção discute algumas dessas consultas no contexto do DenseED. O DenseED foi escolhido nessas análises por ter um modelo de proveniência de *workflow* mais familiar à autora, que é explorado com mais detalhes em [42]. Nas consultas apresentada temos a citação de um computador pessoal com a seguinte configuração: processador CPU Intel Core i7-10700K, 3.80GHz, 16

núcleos, e 15,5 GiB, que também foi utilizado para executar o DenseED e gerar as imagens de contêiner utilizadas nos experimentos da Seção 4.1.

A Tabela 4.4 mostra como a consulta Q5 explora imagens de contêiner criadas pela usuária 'Liliane'. Nesse cenário, Liliane possui um banco de dados com imagens de contêiner que podem ser utilizadas em diversos *workflows*. As imagens marcadas como *dfanalyzer*, *py-readseq-modelGenerator* e *java-readseq-modelGenerator* são todas baseadas em uma imagem Java que não foi criada por Liliane e compartilham a arquitetura de *kernel* amd64. Ao reproduzir este *workflow* com qualquer uma dessas imagens de contêiner, é vantajoso verificar se a imagem base possui uma versão compatível com a nova arquitetura alvo.

Tabela 4.4: Q5: *Recupere todas as imagens que foram criadas por um usuário específico.* - A tabela mostra as imagens de contêiner geradas pelo usuário 'Liliane';

id	autor	arch	vendor	image tags	descrição	Env name	tipo	Base Image
8	Liliane	amd64	Singularity	denseed	DenseED with DfAnalyzer and MonetDB	liliane-imac20	application	tensorflow
9	Liliane	amd64	Singularity	provData	DfAnalyzer and MonetDB	liliane-imac20	provenance	dfanalyzer
1	Liliane	amd64	Docker	icc	Intel compiler with dfa-lib-cpp	liliane-ubuntu	application	N/A
3	Liliane	amd64	Singularity	dfanalyzer	Container of dfanalyzer	liliane-ubuntu	provCollector	java
4	Liliane	amd64	Singularity	monetdb	Container of provdeploy database	liliane-imac20	database	N/A
5	Liliane	amd64	Singularity	py-readseq-modelGenerator	ReadSeq, python2, java, raxml, dfa-lib-python with telemetry, and psutils for python applications	liliane-iMac20	application	java
6	Liliane	N/A	N/A	java-readseq-modelGenerator	ReadSeq, python2, java, raxml, dfa-lib-python with telemetry, and psutils for Java applications	liliane-iMac20	application	java

O DenseED explora variações em sua arquitetura de CNN (convolucional, discriminador/gerador, condicional, etc.), métricas de treinamento e técnicas de paralelização. Além disso, ele testa múltiplos ambientes de execução com diferentes versões do TensorFlow e do CUDA (*i.e.*, múltiplos contêineres), e essas mudanças impactam o tempo de execução do treinamento.

Analisar todos esses diferentes detalhes ao longo do tempo torna-se cada vez mais complexo, levando a um processo de tentativa e erro. Sem a proveniência de contêiner, é difícil reproduzir os mesmos resultados ou associar o DenseED aos contêineres compatíveis. Por exemplo, mesmo ao utilizar um único ambiente como o SDumont, os usuários enfrentam esse desafio, já que há várias partições de GPU disponíveis com dispositivos distintos (*e.g.*, K40 e V100) e em quantidades variáveis. Esses dispositivos exigem combinações diferentes de cuDNN, CUDA, TensorFlow e outros softwares, resultando em imagens de contêiner distintas. Além disso, essas combinações são afetadas pela origem das imagens (*e.g.*, Docker Hub, NCG, etc.), que, no melhor cenário, ignorarão as GPUs caso sejam implantadas em uma GPU incompatível.

Com o uso do `ProvDeploy`, essas informações são capturadas e ficam disponíveis

por meio do modelo de dados proposto, preenchendo lacunas nas informações de execução, conforme requerido na consulta Q7. A Tabela 4.5 apresenta os resultados da consulta Q7 para cada ambiente com diferentes composições de contêineres. Apresentamos apenas a composição de contêiner com o menor tempo decorrido em cada ambiente e o menor R^2 para a execução do DenseED.

Tabela 4.5: Q7: *Um usuário executou o workflow com diferentes composições e em ambientes distintos, aumentando o número de contêineres. Recupere as diferenças nas composições entre os diferentes ambientes.* - A tabela mostra as composições de contêiner ordenadas pelo menor tempo decorrido por ambiente e o melhor R^2.

job_id	**Composição**	**Tempo decorrido (m)**	**Env Name**	R^2
10898072	Partial modular	4.01	sequana_gpu	0.9979
10898073	Provenance modular	4.10	sequana_gpu	0.9978
10898075	Coarse-grained	4.19	sequana_gpu	0.9975
10905992	Provenance Modular	20.55	cpu	0.9976
10901804	Coarse-grained	21.10	cpu	0.9978
10900921	Partial modular	21.32	cpu	0.9976
N/A	Coarse-grained	32.78	liliane-iMac20-1	0.9975
N/A	Partial modular	33.53	liliane-iMac20-1	0.9977
N/A	Provenance modular	34.35	liliane-iMac20-1	0.9975

A consulta Q7 exige a junção das entidades *execution*, *env_partition*, *workflow_execution* e *workflow* com aquelas do modelo de dados de proveniência do DenseED, incluindo hiperparâmetros e métricas dentro de um intervalo restrito da entidade *execution*. Também seria possível listar todas as imagens de contêiner associadas com cada uma dessas composições através da entidade *container_image* e saber quais características dessas imagens são utilizadas em sua chamada através da entidade *container* e *start_command*.

Na consulta Q7, as diferenças nos valores de R^2 são pequenas, mas observa-se que, como esperado, o tempo decorrido aumenta de acordo com o tipo de ambiente. A melhor composição de contêiner para o *workflow* varia conforme a especificação do ambiente, especialmente em cenários com recursos limitados, como o 'liliane-iMac20-1', que é um computador pessoal. Neste ambiente, a composição *coarse-grained* foi a melhor quando considerado apenas o tempo decorrido. Isso ocorre porque uma composição *coarse-grained* envolve uma única imagem de contêiner, permitindo que ela se expanda e utilize livremente os recursos disponíveis. Em cenários com recursos limitados, outras composições tendem a enfrentar competição por recursos entre contêineres. Além disso, elas exigem tempo extra para iniciar e parar contêineres, o que se acumula ao longo do tempo, aumentando o tempo total decorrido.

A consulta Q8 pode ser executada identificando o melhor resultado no DenseED com o maior R^2 e o menor RMSE. A consulta Q8 realiza a junção das entidades *exe-*

cution, *env_partition*, *workflow_execution*, *container_image* e *workflow* com aquelas do modelo de proveniência do DenseED, como hiperparâmetros e métricas dentro de um intervalo restrito do treinamento do DenseED. Considerando que o DenseED foi executado com várias composições e em diferentes máquinas hospedeiras, a proveniência de contêineres é crucial. Sem isso, o usuário só teria conhecimento dos valores de K e L que levaram o DenseED aos valores apresentados de R^2 e RMSE. O usuário ficaria sem informações sobre as imagens de contêiner (tensorflow, dfanalyzer, monetdb) e o ambiente em que esse modelo foi executado (sequana_gpu), tornando a repetição ou reprodução da execução muito mais difícil.

Tabela 4.6: Q8: *Qual ambiente (máquina e contêiner) executou o workflow com os melhores resultados?* - A tabela mostra o ambiente com o melhor RMSE.

job_id	**Env Name**	**Image tag**	**RMSE**	R^2	K	L
10898072	sequana_gpu	tensorflow, dfanalyzer, monetdb	0.00093	0.9979	32	9
10897983	sequana_gpu	denseed	0.00103	0.9976	16	9
10898090	sequana_gpu	tensorflow, provdata	0.00111	0.9975	24	8

Além de consultas utilizando SQL, por observar as relações do W3C PROV o modelo permite a representação dos dados em forma de grafo de proveniência. Na Figura 4.16, apresentamos a proveniência utilizada para consultas como Q6, Q7 e Q8 pois consulta ao agente *environment*, às entidades *container_image*, *container* e *workflow* e às atividades *execution* e *workflow_execution*.

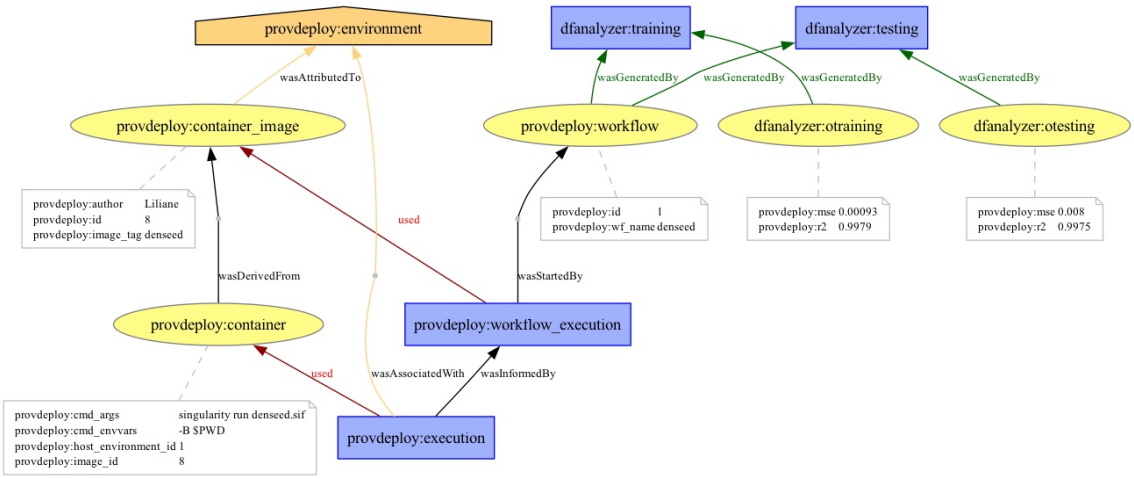


Figura 4.16: Representação de proveniência com W3C PROV

Neste capítulo apresentamos experimentos sobre diferentes composições de contêiner com o objetivo de observar o impacto de cada composição no tempo de execução do *workflow* e consumo de recursos. O que foi observado é que não há uma composição que seja melhor em todos os casos e que seu desempenho é combinado com as características e demandas do *workflow*. Além disso, apresentamos um es-

tudo de caso com a proveniência de contêiner que mostra como sua associação com a proveniência de *workflows* pode enriquecer as análises e auxiliar o uso de contêineres.

Capítulo 5

Conclusões

Nesta tese, exploramos os desafios da implantação de *workflows* científicos por meio de contêineres. Como os contêineres podem impactar significativamente a execução de *workflows*, propomos o uso de proveniência de contêineres como uma abordagem para melhorar a interpretabilidade dos resultados e preencher lacunas introduzidas por sua utilização.

A pergunta de pesquisa que guiou essa tese foi a seguinte: **"Como podemos prover suporte à captura de proveniência em diferentes composições para a implantação de conteinerizados em larga escala?"**.

A conteinerização de um *workflow* envolve diversas possibilidades de organização das atividades, às quais chamamos de composições. Para permitir que os usuários avaliem essas composições, conduzimos uma avaliação qualitativa com diferentes *workflows*, que revelou que, embora algumas composições apresentem menor tempo de execução ou consumo de recursos, outras podem oferecer benefícios a longo prazo, se mantendo viáveis para implantação. Também pudemos observar a partir dos experimentos que a perda de desempenho na composição *fine-grained* poder ser considerada insignificante em relação aos benefícios relacionados à manutenção e a portabilidade. Essa avaliação foi feita utilizando a `ProvDeploy`, que foi reformulada para possibilitar a execução e avaliação de múltiplas composições. Além disso, implementamos um modelo de dados de proveniência de *workflows* para contêineres na `ProvDeploy` de acordo com as anotações de OCI e seguindo o W3C PROV.

Como consequência da pergunta de pesquisa, listamos questões específicas com as contribuições associadas abaixo.

- **Como auxiliar a definição de composição de contêineres para *workflows*?** Exploramos diferentes composições para conteinerização de *workflows* de modo quantitativo e qualitativo. A partir de *workflows* com comportamentos e características computacionais distintas, elencamos aspectos qualitativos relacionados à dificuldade de implantação e reprodução de *workflows* contei-

nerizados. Para saber o reflexo de tais composições no desempenho computacional do *workflow*, executamos experimentos em ambientes PAD e em nuvens computacionais utilizando a `ProvDeploy`. Estas avaliações revelaram que embora algumas composições apresentem menor tempo de execução ou consumo de recursos, outras podem oferecer benefícios a longo prazo, mantendo-se viáveis.

- **Como capturar dados de proveniência em diferentes composições?** Para realizar a captura de dados, remodelamos a `ProvDeploy` para implantar diferentes composições de contêiner que eram registradas no banco de dados de proveniência de *workflow* com dados de contêiner utilizando um banco de dados colunar que permite consultas SQL e integração com diferentes bibliotecas como Pandas, Seaborn, ElastiSearch, Kibana e etc. Além disso, ao seguir as relações do W3C PROV, temos a possibilidade de gerar grafos e documentos de proveniência com o ProvN, que aumentam a confiabilidade e rastreabilidade dos dados capturados. O uso de anotações da OCI assegura que os dados coletados possam representar diversas *engines* de contêiner, permitindo que o usuário explore diferentes *engines* sem prejuízo a representação da proveniência de contêiner.

- **Como consultar dados de proveniência do *workflow* junto aos dados dos contêineres?** Realizamos um levantamento dos dados relevantes de contêineres de acordo com a literatura e as anotações da OCI para serem registrados em um modelo de dados para proveniência de contêineres que foi proposto observando a recomendação do W3C-PROV. Anteriormente a esta tese não foi encontrado um modelo de dados de proveniência que incluísse dados de contêineres. Este modelo foi implementado na `ProvDeploy`, observando as anotações do W3C PROV e permite consultas via SQL e geração de grafos de proveniência. Essas consultas integram dados de proveniência de *workflow* aos dados de contêiner adicionando contexto de contêiner aos dados do *workflow*.

O modelo de dados de proveniência de contêiner foi desenvolvido com foco em interoperabilidade e facilidade de análise, ao combinar o padrão W3C-PROV com as anotações da OCI. Seus principais objetivos incluem facilitar o reuso de imagens, a reprodutibilidade e a análise de experimentos, auxiliando a implantação de *workflows* conteinerizados ao longo do tempo. Embora o modelo possa ser implementado por diferentes abordagens, sua integração à `ProvDeploy` permitiu capturar e organizar informações de execução de *workflows* conteinerizados, especialmente em ambientes com características distintas, simplificando nossas análises e permitindo comparações mais robustas.

Com isso, contribuímos para a implantação de *workflows* científicos conteinerizados, integrando a coleta de proveniência de contêiner à proveniência do *workflow* em si. Essa integração possibilitou maior rastreabilidade e suporte para a tomada de decisões durante a execução dos *workflows*.

Reconhecemos, no entanto, algumas limitações. O modelo apresentado e a `ProvDeploy` ainda não oferecem suporte a execuções distribuídas em múltiplos ambientes. Nesse contexto, acreditamos que o uso de ferramentas como Kubernetes, em conjunto com um *middleware* adequado, seja mais apropriado, conforme explorado em FERREIRA *et al.* [24] que propõe execução de *workflows* científicos em múltiplos ambientes conteinerizados através do Kubernetes. Por ser extensível, o modelo proposto pode ser adaptado futuramente para suportar cenários mais complexos.

Observamos que as avaliações de integração com o modelo foram direcionadas a ferramentas compatíveis com o padrão W3C PROV, como a DfAnalyzer. Essa escolha foi motivada pela familiaridade com essas ferramentas e pela crescente adoção do W3C PROV, que facilita tanto a realização de consultas quanto a geração de grafos e documentos de proveniência. Destacamos que a integração do modelo com ferramentas não compatíveis com o padrão W3C PROV, como o noWorkflow, exigirá etapas de pós-processamento, que já fazem parte do funcionamento intrínseco dessa ferramenta.

Além disso, os modelos de proveniência de *workflows* e contêineres atualmente não abrangem ferramentas que representem *workflows* sem a definição de uma chave global. Isso limita sua aplicabilidade em determinados cenários, uma vez que uma chave global identificadora do *workflow* é necessária para integrar o modelo de proveniência contêiner-*aware* com a proveniência do *workflow*.

Adicionalmente, destacamos que a abordagem implementada na `ProvDeploy` foi projetada para execuções longas e em larga escala, permitindo que a sobrecarga introduzida seja diluída ao longo do tempo.

Dessa forma, como trabalhos futuros, sugerimos:

1. Estudos mais aprofundados sobre serviços de proveniência de *workflow*, com o objetivo de desenvolver uma modelagem que atenda a uma maior diversidade de serviços.

2. Integração do modelo com ferramentas de orquestração de contêineres, como Kubernetes, para otimizar ainda mais o gerenciamento de recursos.

3. Uso dos dados de proveniência coletados para identificar padrões e recomendar automaticamente a composições mais adequadas a diferentes *workflows*.

4. Exploração das informações obtidas dos dados de proveniência para identificar requisitos mínimos de recursos e características específicas de cada *workflow*,

otimizando sua implantação.

5. Desenvolvimento de mecanismo que permita a persistência e leitura de dados dentro e fora dos contêineres de maneira simultânea. Pois facilitaria o armazenamento e transferência de dados em contêineres e permitiria o uso de volumes sem necessidade de copiar os dados para fora dos contêineres.

6. Uso dos dados de proveniência de contêiner para melhorar dimensionamento do ambiente conteinerizados para um *workflow* específico.

A proveniência de contêineres representa um avanço significativo para a implantação de *workflows*, ao preencher lacunas introduzidas pela conteinerização. Ferramentas como a `ProvDeploy` têm o potencial de simplificar a coleta de proveniência e a implantação desses *workflows*, facilitando a reprodução de experimentos e aumentando a eficiência no desenvolvimento de *workflows* conteinerizados.

Referências Bibliográficas

[1] ABBAS, M., KHAN, S., MONUM, A., ZAFFAR, F., TAHIR, R., EYERS, D., IRSHAD, H., GEHANI, A., YEGNESWARAN, V., PASQUIER, T., 2022, "PACED: Provenance-based Automated Container Escape Detection". In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 261–272, November. doi: 10.1109/IC2E55432.2022.00035.

[2] AHMAD, R., NAKAMURA, Y., MANNE, N. N., MALIK, T., 2020, "{PROV-CRT}: Provenance support for container runtimes". In: *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*, June.

[3] AL-DHURAIBI, Y., PARAISO, F., DJARALLAH, N., MERLE, P., 2017, "Autonomic vertical elasticity of docker containers with elasticdocker", pp. 472–479.

[4] BECHHOFER, S., DE ROURE, D., GAMBLE, M., GOBLE, C., BUCHAN, I., 2010, "Research objects: Towards exchange and reuse of digital knowledge", *Nature Proc.*, pp. 1–6.

[5] BELHAJJAME, K., B'FAR, R., CHENEY, J., COPPENS, S., CRESSWELL, S., GIL, Y., GROTH, P., KLYNE, G., LEBO, T., MCCUSKER, J., MILES, S., MYERS, J., SAHOO, S., TILMES, C., 2013, "Prov-dm: The prov data model", *W3C Recommendation*, v. 14, pp. 15–16.

[6] BENTALEB, O., BELLOUM, A. S., SEBAA, A., EL-MAOUHAB, A., 2022, "Containerization technologies: Taxonomies, applications and challenges", *The Journal of Supercomputing*, v. 78, n. 1, pp. 1144–1181.

[7] BEZ, J. L., CARNEIRO, A. R., PAVAN, P. J., GIRELLI, V. S., BOITO, F. Z., FAGUNDES, B. A., OSTHOFF, C., DA SILVA DIAS, P. L., MÉHAUT, J.-F., NAVAUX, P. O., 2020, "I/O performance of the Santos Dumont supercomputer", *The International Journal of High Performance Computing Applications*, v. 34, n. 2, pp. 227–245.

[8] BISONG, E., 2019, "Kubeflow and Kubeflow Pipelines". In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pp. 671–685, Berkeley, CA, Apress. ISBN: 978-1-4842-4470-8. doi: 10.1007/978-1-4842-4470-8_46. Disponível em: <https://doi.org/10.1007/978-1-4842-4470-8_46>.

[9] BUTT, A. S., FITCH, P., 2021, "A provenance model for control-flow driven scientific workflows", *Data & Knowledge Engineering*, v. 131, pp. 101877.

[10] CANON, R. S., 2020, "The role of containers in reproducibility". In: *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 19–25. IEEE, December.

[11] CARNEIRO, A. R., BEZ, J. L., BOITO, F. Z., FAGUNDES, B. A., OSTHOFF, C., NAVAUX, P. O., 2018, "Collective i/o performance on the santos dumont supercomputer". In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 45–52. IEEE.

[12] CHEN, X., IRSHAD, H., CHEN, Y., GEHANI, A., YEGNESWARAN, V., 2021, "{CLARION}: Sound and clear provenance tracking for microservice deployments". In: *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3989–4006, August.

[13] CHOI, Y.-D., ROY, B., NGUYEN, J., AHMAD, R., MAGHAMI, I., NASSAR, A., LI, Z., CASTRONOVA, A. M., MALIK, T., WANG, S., GOODALL, J. L., 2023, "Comparing containerization-based approaches for reproducible computational modeling of environmental systems", *Environmental Modelling & Software*, v. 167, pp. 105760. ISSN: 1364-8152. doi: https://doi.org/10.1016/j.envsoft.2023.105760. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1364815223001469>.

[14] DATTA, P., POLINSKY, I., INAM, M. A., BATES, A., ENCK, W., 2022, "ALASTOR: Reconstructing the Provenance of Serverless Intrusions". In: *31st USENIX Security*, pp. 2443–2460.

[15] DE OLIVEIRA, D., OCAÑA, K. A. C. S., BAIÃO, F. A., MATTOSO, M., 2012, "A Provenance-based Adaptive Scheduling Heuristic for Parallel Scientific Workflows in Clouds", *Journal of grid Computing*, v. 10, n. 3, pp. 521–552. doi: 10.1007/s10723-012-9227-2.

[16] DO, C. B., BRUDNO, M., BATZOGLOU, S., 2004, "Prob Cons: probabilistic consistency-based multiple alignment of amino acid sequences". In: *AAAI*, pp. 703–708.

[17] DULAM, N., IMMANENI, J., 2023, "Kubernetes 1.27: Enhancements for Large-Scale AI Workloads", *Journal of Artificial Intelligence Research and Applications*, v. 3, n. 2, pp. 1149–1171.

[18] EDER, M., 2016, "Hypervisor-vs. container-based virtualization", *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, v. 1.

[19] EDGAR, R. C., 2004, "MUSCLE: a multiple sequence alignment method with reduced time and space complexity", *BMC bioinformatics*, v. 5, n. 1, pp. 1–19.

[20] ELIA, D., FIORE, S., ALOISIO, G., 2021, "Towards HPC and Big Data Analytics Convergence: Design and Experimental Evaluation of a HPDA Framework for eScience at Scale", *IEEE Access*, v. 9, pp. 73307–73326. doi: 10.1109/ACCESS.2021.3079139.

[21] ELISSEEV, V., MANSON-SAWKO, R., PEÑA-MONFERRER, C., LUPIERI, G., SEATON, M., BOCCARDO, G., HANDGRAAF, J.-W., TODOROV, I., MARCHISIO, D., KOWALSKI, A., 2022, "Multiscale scientific workflows on high-performance hybrid cloud". In: *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 1–11. IEEE.

[22] ENGLBRECHT, L., LANGNER, G., PERNUL, G., QUIRCHMAYR, G., 2019, "Enhancing credibility of digital evidence through provenance-based incident response handling". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pp. 1–6.

[23] FELTER, W., FERREIRA, A., RAJAMONY, R., RUBIO, J., 2015, "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.

[24] FERREIRA, W., KUNSTMANN, L., PAES, A., BEDO, M., DE OLIVEIRA, D., 2024, "AkôFlow: um Middleware para Execução de Workflows Científicos em Múltiplos Ambientes Conteinerizados". In: *Anais do XXXIX Simpósio Brasileiro de Bancos de Dados*, pp. 27–39, Porto Ale-

gre, RS, Brasil. SBC. doi: 10.5753/sbbd.2024.241126. Disponível em: <https://sol.sbc.org.br/index.php/sbbd/article/view/30680>.

[25] FILGUEIRA, R., DA SILVA, R. F., KRAUSE, A., DEELMAN, E., ATKINSON, M., 2017, "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, June.

[26] FREIRE, J., KOOP, D., SANTOS, E., SILVA, C. T., 2008, "Provenance for computational tasks: A survey", *Computing in Science & Engineering*, v. 10, n. 3, pp. 11–21.

[27] FREITAS, R. S., BARBOSA, C. H., GUERRA, G. M., COUTINHO, A. L., ROCHINHA, F. A., 2021, "An encoder-decoder deep surrogate for reverse time migration in seismic imaging under uncertainty", *Computational Geosciences*, v. 25, pp. 1229–1250.

[28] GERHARDT, L., BHIMJI, W., CANON, S., FASEL, M., JACOBSEN, D., MUSTAFA, M., PORTER, J., TSULAIA, V., 2017, "Shifter: Containers for HPC", *Journal of Physics: Conference Series*, v. 898, n. 8 (oct), pp. 082021. doi: 10.1088/1742-6596/898/8/082021. Disponível em: <https://dx.doi.org/10.1088/1742-6596/898/8/082021>.

[29] GERLACH, W., TANG, W., KEEGAN, K., HARRISON, T., WILKE, A., BISCHOF, J., D'SOUZA, M., DEVOID, S., MURPHY-OLSON, D., DESAI, N., MEYER, F., 2014, "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. doi: 10.1109/DataCloud.2014.6.

[30] GILBERT, D., 2003, "Sequence File Format Conversion with Command-Line Readseq", *Current Protocols in Bioinformatics*, , n. 1, pp. A–1E.

[31] GRUENING, B., SALLOU, O., MORENO, P., DA VEIGA LEPREVOST, F., MÉNAGER, H., SØNDERGAARD, D., RÖST, H., SACHSENBERG, T., O'CONNOR, B., MADEIRA, F., DOMINGUEZ DEL ANGEL, V., CRUSOE, M. R., VARMA, S., BLANKENBERG, D., JIMENEZ, R. C., COMMUNITY, B., PEREZ-RIVEROL, Y., 2018, "Recommendations for the packaging and containerizing of bioinformatics software", *F1000Research*, v. 7.

[32] GUEDES, T., SILVA, V., MATTOSO, M., BEDO, M. V., DE OLIVEIRA, D., 2019, "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. IEEE, February.

[33] GUEDES, T., MARTINS, L. B., FALCI, M. L. F., SILVA, V., OCAÑA, K. A., MATTOSO, M., BEDO, M., DE OLIVEIRA, D., 2020, "Capturing and analyzing provenance from spark-based scientific workflows with SAMbA-RaP", *Future Generation Computer Systems*, v. 112, pp. 658–669.

[34] HAN, R., ZHENG, M., BYNA, S., TANG, H., DONG, B., DAI, D., CHEN, Y., KIM, D., HASSOUN, J., THORSLEY, D., 2024, "PROV-IO$^+$: A Cross-Platform Provenance Framework for Scientific Data on HPC Systems", *IEEE Transactions on Parallel and Distributed Systems*, v. 35, n. 5, pp. 844–861. doi: 10.1109/TPDS.2024.3374555.

[35] HOBSON, T., YILDIZ, O., NICOLAE, B., HUANG, J., PETERKA, T., 2021, "Shared-Memory Communication for Containerized Workflows". In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 123–132. doi: 10.1109/CCGrid51090.2021.00022.

[36] HU, G., ZHANG, Y., CHEN, W., 2019, "Exploring the Performance of Singularity for High Performance Computing Scenarios". In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 2587–2593. doi: 10.1109/HPCC/SmartCity/DSS.2019.00362.

[37] HUELSENBECK, J. P., RONQUIST, F., 2001, "MRBAYES: Bayesian inference of phylogenetic trees", *Bioinformatics*, v. 17, n. 8, pp. 754–755.

[38] KATOH, K., TOH, H., 2008, "Recent developments in the MAFFT multiple sequence alignment program", *Briefings in bioinformatics*, v. 9, n. 4, pp. 286–298.

[39] KATZ, D. S., JACOB, J. C., DEELMAN, E., KESSELMAN, C., SINGH, G., SU, M.-H., BERRIMAN, G., GOOD, J., LAITY, A., PRINCE, T. A., 2005, "A comparison of two methods for building astronomical image mosaics on a grid". In: *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pp. 85–94. IEEE.

[40] KELLER TESSER, R., BORIN, E., 2022, "Containers in HPC: A Survey", *J. Supercomput.*, v. 79, n. 5 (oct), pp. 5759–5827. ISSN: 0920-8542. doi: 10.1007/s11227-022-04848-y. Disponível em: <`https://doi.org/10.1007/s11227-022-04848-y`>.

[41] KENNEDY, D., OLAYA, P., LOFSTEAD, J., VARGAS, R., TAUFER, M., 2022, "Augmenting Singularity to Generate Fine-grained Workflows, Record Trails, and Data Provenance". In: *2022 IEEE 18th International Conference on e-Science (e-Science)*, pp. 403–404. doi: 10.1109/eScience55777.2022.00059.

[42] KUNSTMANN, L., PINA, D., SILVA, F., PAES, A., VALDURIEZ, P., DE OLIVEIRA, D., MATTOSO, M., 2021, "Online Deep Learning Hyperparameter Tuning based on Provenance Analysis", *Journal of Information and Data Management*, v. 12, n. 5, pp. 396–414.

[43] KUNSTMANN, L., PINA, D., OLIVEIRA, L., OLIVEIRA, D., MATTOSO, M., 2022, "`ProvDeploy`: Explorando Alternativas de Conteinerização com Proveniência para Aplicações Científicas com PAD (in Portuguese)". In: *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pp. 49–60, Porto Alegre, RS, Brasil. SBC. doi: 10.5753/wscad.2022.226363.

[44] KUNSTMANN, L., PINA, D., DE OLIVEIRA, D., MATTOSO, M., 2024, "`ProvDeploy`: Provenance-oriented Containerization of High Performance Computing Scientific Workflows", *arXiv preprint arXiv:2403.15324 / Under Review.*

[45] KUNSTMANN, L., PINA, D., DE OLIVEIRA, D., MATTOSO, M., 2024, "Scientific Workflow Deployment: Container Provenance in High-Performance Computing". In: *Anais do XXXIX Simpósio Brasileiro de Bancos de Dados*, pp. 457–470, Porto Alegre, RS, Brasil, . SBC. doi: 10.5753/sbbd.2024.240194. Disponível em: <`https://sol.sbc.org.br/index.php/sbbd/article/view/30713`>.

[46] KUNSTMANN, L. N. D. O., 2020, *ProvDeploy: Apoio à coleta de dados de proveniência em scripts de execução de códigos científicos.* Tese de Mestrado, Universidade Federal do Rio de Janeiro.

[47] KURTZER, G. M., SOCHAT, V., BAUER, M. W., 2017, "Singularity: Scientific containers for mobility of compute", *PLOS ONE*, v. 12, n. 5 (05), pp. 1–20. doi: 10.1371/journal.pone.0177459. Disponível em: <`https://doi.org/10.1371/journal.pone.0177459`>.

[48] LAMPA, S., DAHLÖ, M., ALVARSSON, J., SPJUTH, O., 2019, "SciPipe: A workflow library for agile development of complex and dynamic bioinformatics pipelines", *GigaScience*, v. 8, n. 5, pp. giz044.

[49] LI, F., TAN, W. J., CAI, W., 2022, "A wholistic optimization of containerized workflow scheduling and deployment in the cloud–edge environment", *Simulation Modelling Practice and Theory*, v. 118, pp. 102521.

[50] LI, Z., CHARD, R., WARD, L., CHARD, K., SKLUZACEK, T. J., BABUJI, Y., WOODARD, A., TUECKE, S., BLAISZIK, B., FRANKLIN, M. J., FOSTER, I., 2021, "DLHub: Simplifying publication, discovery, and use of machine learning models in science", *Journal of Parallel and Distributed Computing*, v. 147, pp. 64–76. ISSN: 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2020.08.006. Disponível em: <`https://www.sciencedirect.com/science/article/pii/S0743731520303464`>.

[51] LIU, J., CHLOSTA, M., SCHABER, N., SIDLER, J., LUTZ, R., SCHLACHTER, T., HAGENMEYER, V., 2023, "Introducing PROOF-A PROcess Orchestration Framework for the Automation of Computational Scientific Workflows and Co-Simulations". In: *2023 Open Source Modelling and Simulation of Energy Systems (OSMSES)*, pp. 1–6. IEEE.

[52] MALIK, T., YUAN, Z., ESSAWY, B. T., CASTRONOVA, A. M., GAN, T., TARBOTON, D. G., GOODALL, J. L., PECKHAM, S. D., CHOI, E., BHATT, A., 2018, "Sciunits: Reusable Research Objects". In: *AGU Fall Meeting Abstracts*, v. 2018, pp. IN34B–10. AGU.

[53] MANNE, N. N., SATPATI, S., MALIK, T., BAGCHI, A., GEHANI, A., CHAUDHARY, A., 2022, "CHEX: multiversion replay with ordered checkpoints", *Proceedings of the VLDB Endowment*, v. 15, n. 6, pp. 1297–1310.

[54] MARTI-RENOM, M. A., MADHUSUDHAN, M., SALI, A., 2004, "Alignment of protein sequences by their profiles", *Protein Science*, v. 13, n. 4, pp. 1071–1087.

[55] MCCARTY, S., 2018, "A Practical Introduction to container terminology", *RedHat Developer, Blog.*

[56] MCPHILLIPS, T., SONG, T., KOLISNIK, T., AULENBACH, S., BELHAJJAME, K., BOCINSKY, K., CAO, Y., CHIRIGATI, F., DEY, S., FREIRE, J., OTHERS, 2015, "YesWorkflow: a user-oriented, language-independent tool for recovering workflow information from scripts", *arXiv preprint arXiv:1502.02403.*

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

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

[59] MISSIER, P., WOODMAN, S., HIDEN, H., WATSON, P., 2016, "Provenance and data differencing for workflow reproducibility analysis", *Concurrency and Computation: Practice and Experience*, v. 28, n. 4, pp. 995–1015.

[60] MODI, A., REYAD, M., MALIK, T., GEHANI, A., 2023, "Querying Container Provenance". In: *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, p. 1564–1567, New York, NY, USA. Association for Computing Machinery. ISBN: 9781450394192. doi: 10.1145/3543873.3587568. Disponível em: <`https://doi.org/10.1145/3543873.3587568`>.

[61] MORA-CANTALLOPS, M., SÁNCHEZ-ALONSO, S., GARCÍA-BARRIOCANAL, E., SICILIA, M.-A., 2021, "Traceability for trustworthy AI: a review of models and tools", *Big Data and Cognitive Computing*, v. 5, n. 2, pp. 20.

[62] MURTA, L., BRAGANHOLO, V., CHIRIGATI, F., KOOP, D., FREIRE, J., 2015, "noWorkflow: capturing and analyzing provenance of scripts". In: *IPAW 2014*, pp. 71–83. Springer.

[63] NIDDODI, C., GEHANI, A., MALIK, T., MOHAN, S., RILEE, M. L., 2023, "IOSPReD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility", *IEEE Access*, v. 11, pp. 1718–1731.

[64] NOTREDAME, C., HIGGINS, D. G., HERINGA, J., 2000, "T-Coffee: A novel method for fast and accurate multiple sequence alignment", *Journal of molecular biology*, v. 302, n. 1, pp. 205–217.

[65] NOVELLA, J. A., EMAMI KHOONSARI, P., HERMAN, S., WHITENACK, D., CAPUCCINI, M., BURMAN, J., KULTIMA, K., SPJUTH, O., 2019, "Container-based bioinformatics with Pachyderm", *Bioinformatics*, v. 35, n. 5, pp. 839–846.

[66] OCAÑA, K. A., DE OLIVEIRA, D., OGASAWARA, E., DÁVILA, A. M., LIMA, A. A., MATTOSO, M., 2011, "Sciphy: a cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes". In: *Advances in Bioinformatics and Computational Biology: 6th Brazilian Symposium on Bioinformatics, BSB 2011, Brasilia, Brazil, August 10-12, 2011. Proceedings 6*, pp. 66–70. Springer, August.

[67] OLAYA, P., KENNEDY, D., LLAMAS, R., VALERA, L., VARGAS, R., LOFSTEAD, J., TAUFER, M., 2022, "Building Trust in Earth Science Findings through Data Traceability and Results Explainability", *IEEE Transactions on Parallel and Distributed Systems*, v. 34, n. 2, pp. 704–717.

[68] PIMENTEL, J. F., MURTA, L., BRAGANHOLO, V., FREIRE, J., 2017, "noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts", *Proc. VLDB Endow.*, v. 10, n. 12, pp. 1841–1844. doi: 10.14778/3137765.3137789. Disponível em: <`http://www.vldb.org/pvldb/vol10/p1841-pimentel.pdf`>.

[69] PINA, D., CHAPMAN, A., KUNSTMANN, L., DE OLIVEIRA, D., MATTOSO, M., 2024, "`DLProv`: A Data-Centric Support for Deep Learning Workflow Analyses". In: *Companion of the 2024 ACM SIGMOD/PODS, Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning.*, DEEM '24, pp. 77–85. ACM. ISBN: 9798400706110. doi: 10.1145/3650203.3663337. Disponível em: <`https://doi.org/10.1145/3650203.3663337`>.

[70] PORTELLA, G., NAKANO, E., RODRIGUES, G. N., MELO, A. C., 2019, "Utility-Based Strategy for Balanced Cost and Availability at the Cloud Spot Market". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 214–218. doi: 10.1109/CLOUD.2019.00045.

[71] PRIEDHORSKY, R., RANDLES, T., 2017, "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA. Association for Computing Machinery. ISBN: 9781450351140. doi: 10.1145/3126908.3126925. Disponível em: <`https://doi.org/10.1145/3126908.3126925`>.

[72] PRIEDHORSKY, R., CANON, R. S., RANDLES, T., YOUNGE, A. J., 2021, "Minimizing privilege for building HPC containers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, November.

[73] QASHA, R., CALA, J., WATSON, P., 2015, "Towards automated workflow deployment in the cloud using tosca". In: *2015 IEEE 8th International Conference on Cloud Computing*, pp. 1037–1040. IEEE, August.

[74] RAMACHANDRAN, A., KANTARCIOGLU, M., 2018, "Smartprovenance: a distributed, blockchain based dataprovenance system". In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pp. 35–42.

[75] SAKELLARIOU, R., ZHAO, H., DEELMAN, E., 2009, "Mapping Workflows on Grid Resources: Experiments with the Montage Workflow". In: *ERCIM W. Group on Grids*, pp. 119–132. doi: 10.1007/978-1-4419-6794-7_10. Disponível em: <https://doi.org/10.1007/978-1-4419-6794-7_10>.

[76] SATAPATHY, U., THAKUR, R., CHATTOPADHYAY, S., CHAKRABORTY, S., 2023, "Disprotrack: Distributed provenance tracking over serverless applications". In: *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pp. 1–10. IEEE.

[77] SCHLEGEL, M., SATTLER, K.-U., 2023, "Management of Machine Learning Lifecycle Artifacts: A Survey", *SIGMOD Rec.*, v. 51, n. 4 (jan), pp. 18–35. ISSN: 0163-5808. doi: 10.1145/3582302.3582306. Disponível em: <https://doi.org/10.1145/3582302.3582306>.

[78] SCHULZ, W. L., DURANT, T. J., SIDDON, A. J., TORRES, R., 2016, "Use of application containers and workflows for genomic data analysis", *Journal of pathology informatics*, v. 7, n. 1, pp. 53.

[79] SHAFFER, T., PHUNG, T. S., CHARD, K., THAIN, D., 2023, "Landlord: Coordinating Dynamic Software Environments to Reduce Container Sprawl", *IEEE Transactions on Parallel and Distributed Systems*, v. 34, n. 5, pp. 1376–1389.

[80] SHAN, C., XIA, Y., ZHAN, Y., ZHANG, J., 2023, "KubeAdaptor: A docking framework for workflow containerization on Kubernetes", *Future Generation Computer Systems*.

[81] SHEFFIELD, N. C., 2019, "Bulker: A multi-container environment manager", *Databio*.

[82] SILVA, V., CAMPOS, V., GUEDES, T., CAMATA, J., DE OLIVEIRA, D., COUTINHO, A. L., VALDURIEZ, P., MATTOSO, M., 2020, "DfAnalyzer: Runtime dataflow analysis tool for Computational Science and Engi-

neering applications", *SoftwareX*, v. 12, pp. 100592. doi: 10.1016/j.softx.2020.100592.

[83] SOUZA, R., MATTOSO, M., 2018, "Provenance of dynamic adaptations in user-steered dataflows". In: *Provenance and Annotation of Data and Processes: 7th International Provenance and Annotation Workshop, IPAW 2018, London, UK, July 9-10, 2018, Proceedings*, pp. 16–29. Springer, September.

[84] SOUZA, R., SILVA, V., COUTINHO, A. L., VALDURIEZ, P., MATTOSO, M., 2020, "Data reduction in scientific workflows using provenance monitoring and user steering", *Future Generation Computer Systems*, v. 110, pp. 481–501.

[85] STAMATAKIS, A., 2006, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models", *Bioinformatics*, v. 22, n. 21 (08), pp. 2688–2690. ISSN: 1367-4803. doi: 10.1093/bioinformatics/btl446. Disponível em: <`https://doi.org/10.1093/bioinformatics/btl446`>.

[86] STRAESSER, M., BAUER, A., LEPPICH, R., HERBST, N., CHARD, K., FOSTER, I., KOUNEV, S., 2023, "An Empirical Study of Container Image Configurations and Their Impact on Start Times". In: *2023 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). io*, July.

[87] The Kubernetes Authors, 2023. "Overview - Kubernetes". `https://kubernetes.io/docs/concepts/overview/`. Accessed: 2023-07-26.

[88] THOMPSON, J. D., GIBSON, T. J., HIGGINS, D. G., 2003, "Multiple sequence alignment using ClustalW and ClustalX", *Current protocols in bioinformatics*, , n. 1, pp. 2–3.

[89] TITO, L. S., OCANA, K. A., DE OLIVEIRA, D., 2018, "Enriquecimento de Dados de Proveniência de Análises Filogenéticas com Dados do NCBI: uma Abordagem Prática". In: *Anais do XII Brazilian e-Science Workshop*. SBC, July.

[90] TORREZ, A., PRIEDHORSKY, R., RANDLES, T., 2020, "HPC container runtime performance overhead: At first order, there is none", .

[91] VAHI, K., RYNGE, M., PAPADIMITRIOU, G., BROWN, D. A., MAYANI, R., DA SILVA, R. F., DEELMAN, E., MANDAL, A., LYONS, E., ZINK,

M., 2020, "Custom execution environments with containers in pegasus-enabled scientific workflows". In: *2019 15th International Conference on eScience (eScience)*, pp. 281–290. IEEE, March.

[92] WOFFORD, Q., HURD, J., GREENBERG, H., BRIDGES, P. G., AHRENS, J., 2023, "Complete Provenance for Application Experiments with Containers and Hardware Interface Metadata". In: *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pp. 12–24. IEEE, February.

[93] YUAN, D. Y., WILDISH, T., 2020, "Bioinformatics application with Kubeflow for batch processing in clouds". In: *HPDC*, pp. 355–367. Springer.

[94] ZHENG, C., TOVAR, B., THAIN, D., 2017, "Deploying high throughput scientific workflows on container schedulers with makeflow and mesos". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 130–139. IEEE, July.

[95] ZHENG, C., THAIN, D., 2015, "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, June.

[96] ZHOU, N., GEORGIOU, Y., POSPIESZNY, M., ZHONG, L., ZHOU, H., NIETHAMMER, C., PEJAK, B., MARKO, O., HOPPE, D., 2021, "Container orchestration on HPC systems through Kubernetes", *Journal of Cloud Computing*, v. 10, n. 1, pp. 1–14.

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

[98] ZIPPERLE, M., GOTTWALT, F., CHANG, E., DILLON, T., 2022, "Provenance-based intrusion detection systems: A survey", *ACM Computing Surveys*, v. 55, n. 7, pp. 1–36.