ビルドエラーを引き起こすDockerfileの依存関係の分析

坂本 廉也
和歌山大学システム工学部
sakamoto.renya@g.wakayama-u.jp

東 裕之輔
日本総合研究所/和歌山大学システム工学研究科
higashi.yunosuke@g.wakayama-u.jp

大平 雅雄
和歌山大学システム工学部
masao@wakayama-u.ac.jp

要旨

近年，企業で急速に普及しつつあるコンテナ型仮想化技術を用いた仮想化プラットフォームとして *Docker* が挙げられる．*Docker* では，コンテナと呼ばれるアプリケーション実行環境を生成する *Docker* イメージを *Dockerfile* からビルドする．しかし，*Dockerfile* はビルドに失敗することが多々あり，その修正には多くの時間が多く費やされる *[1]*．そこで，*Dockerfile* のビルドエラーに対する自動修正手法である *Shipwright [2]* が提案されている．しかし，*Shipwright* では *Dockerfile* が持つ依存関係までは考慮しておらず，自動修正がおこなえた *Dockerfile* は *18.9%* にとどまっている．本研究では，自動修正手法の改善に向けて *Dockerfile* の依存関係を考慮したビルドエラーの分析をおこなう．*RQ1* では，依存関係が原因のビルドエラーがどの程度存在するのかを調査した．その結果，ベースイメージレイヤとカスタムイメージレイヤの間で発生する依存関係が半数以上を占めていることが明らかになった．*RQ2* では，依存関係が原因のビルドエラーを修正するのに要する時間を調査した．その結果，レイヤレベルの依存関係によるビルドエラーの修正時間は約 *5.8* 時間，イメージレベルの依存関係によるビルドエラーの修正時間は約 *6.3* 時間であることが分かったが，それぞれのビルドエラーの修正時間に統計的な有意差は見られないことが明らかになった．これらの結果から，今後，依存関係が原因のビルドエラーを体系化することで，*Dockerfile* の自動修正手法を大きく改善できる可能性があることがわかった．

1. はじめに

1.1 Dockerfile と Docker イメージ

高度情報化社会において，めまぐるしく移り変わる社会のニーズに対応するために，近年のソフトウェア開発ではプロダクトの迅速なデリバリーが求められている．しかし，さまざまな障壁によって迅速な開発が妨げられることがあり，例えば，障壁となる要素の 1 つとして開発環境の違いが挙げられる．特にオープンソースソフトウェア（OSS）開発では，チームメンバーが開発環境を各々で用意し，ソフトウェアの実装やテストをおこなう．しかし，チームメンバーごとに使用する OS やライブラリが異なるために，実装やテストが実施できないことがある．前述の問題を解決するべく，コンテナ型仮想化技術を利用したソフトウェア環境として Docker が多くの企業で利用されている [3]．Docker を用いて開発環境を軽量なイメージファイル（Docker イメージ）を作成し，各チームメンバーに配布することで，同様の環境上での開発を可能にする．図 1 にコンテナが作成されるまでの流れを示す．

Docker では，**Docker イメージ（以下，イメージ）**を基にコンテナと呼ばれる仮想的なアプリケーション環境を生成する．イメージとはコンテナを動作させるために必要な特定のアプリケーション環境のスナップショットである．また，イメージは，Docker Hub[1] と呼ばれるコンテナレジストリサービスなどで公開することもできる．Docker Hub にはあらゆるユーザが作成したイメージが

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

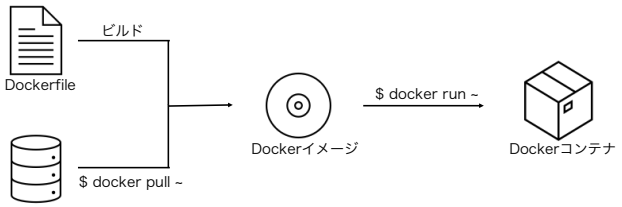

図 1. コンテナ作成までの流れ

```
1  FROM    ubuntu:20.04
2  RUN     apt-get update && apt-get install python3
3  COPY    .   /app
4  WORKDIR /app
```

図 2. Dockerfile の例

公開されており，それらのイメージを基に新しいイメージを開発することも可能である．

Docker Hub やローカル環境のイメージを基にして，新しいイメージを開発するには，**Dockerfile** を記述する必要がある．Dockerfile は，イメージに含める内容を指示するための命令が記述されたテキストファイルである．Dockerfile の命令を上から順に実行すること（以下，ビルド）でイメージを作成することができる．イメージは，レイヤ構造となっており，Dockerfile の 1 つの命令に対し 1 つのレイヤが追加される．ビルドにより新しく追加されるレイヤは，以前に積み重ねられたレイヤとの親子関係を持つため，イメージは依存関係によってビルドされるといえる．Dockerfile には，イメージを作成するための様々な命令が用意されている．まず最初は FROM 命令を記述する必要がある[2]．FROM 命令で既存のイメージ名を指定することで，その既存イメージの内容に変更を加える形で新たなイメージを容易に作成することができる．一方，既存のイメージを流用せずに新たなイメージを作成したい場合は FROM 命令に “scratch” を指定する．本研究では，既存のイメージを流用することをイメージの継承と呼び，継承元のイメージをベースイメージと呼ぶ．図 2 に Dockerfile の例を示す．図 2 では，1 行目で FROM 命令を用いてベースイメージに Ubuntu のバージョン 20.04 のイメージを継承している．イメージは “＜イメージ名＞：＜タグ名＞” の形で記載する．イメージ名には利用したい既存のイメージ名を記載し，タグ名はオプションであるが，利用したいバージョンを記載する [4]．タグ名を記載せずに Dockerfile をビルドした場合，“latest” タグが割り当てられ，最新バージョンのイメージが自動で選択される．

しかしながら，Dockerfile はビルドに失敗することがある（以下，**ビルドエラー**）．Dockerfile は他のプログラミング言語がベースのソフトウェアプロジェクトのビルドよりも失敗しやすいこと [5] や，その修正には多くの時間が費やされること [1] が明らかになっている．そこで，ビルドエラーが発生している Dockerfile の修正を支援するためのツールとして，Shipwright [2] が提案されている．Shipwright では，ビルドエラーが発生している Dockerfile のビルドログを分類し，その分類を基にビルドエラーに対して修正パッチを作成している．ただし，Shipwright は完全にビルドエラーを解決できるわけではない．ビルドエラーが発生した Dockerfile のうち，18.9%しか修正できていないことが報告されてる [2].

1.2 動機と目的

既存の Dockerfile の自動修正手法が完全にビルドエラーを解決できない理由の一つとして，ビルドエラーの原因を考慮した修正をおこなえていないと考えられる．Dockerfile の依存関係は非常に複雑であり，ビルドエラーの根本原因の特定が困難であると想定される．そのため，Dockerfile の自動修正手法を大きく改善するには，まず，ビルドエラーの原因を考慮した自動修正手法の性能向上に向けた分析が必要となる．しかし，Dockerfile の依存関係がビルドエラーに与える影響は未だ明らかになっていない．そこで，本研究では，依存関係が原因で発生しているビルドエラーがどの程度存在するのかを明らかにし，そういったビルドエラーがどの程度の修正時間で修正されているのかを明らかにする．

2. イメージのレイヤ構造とビルドエラー

2.1 イメージのレイヤ構造

イメージを構成するレイヤはベースイメージレイヤとカスタムイメージレイヤの 2 つに大別できる．図 3 にそれぞれの関係を示す．

ベースイメージレイヤ：ベースイメージで実行された命令のみで構成される読み込み専用のレイヤである．ベースイメージは，ある Dockerfile から既にビルドされたイメージであり，それを開発者が自由に選択し Dockerfile

[2] https://docs.docker.jp/engine/reference/builder.html#from

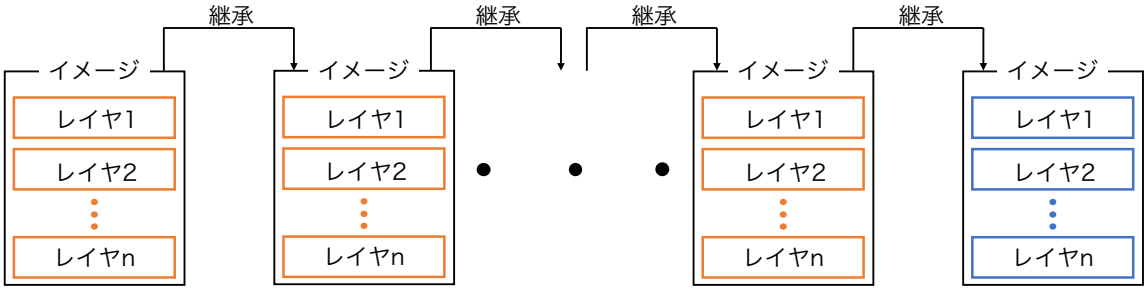

図 3. ベースイメージレイヤとカスタムイメージレイヤ

内で継承をおこなう．図 3 のように，ある開発者が記述した Dockerfile で指定されたベースイメージがさらに別のイメージを継承している，というようなネスト構造になることがある．この場合，継承元のイメージは全てベースイメージレイヤになる．

カスタムイメージレイヤ：新たに Dockerfile 内で記述された命令に対応するレイヤである．ベースイメージレイヤの上にカスタムイメージレイヤが構築され，ベースイメージレイヤとカスタムイメージレイヤには親子関係が存在することになり，ベースイメージの機能を継承する．開発者が変更できる範囲は自身で記述した Dockerfile 内の命令のみであり，開発者はベースイメージレイヤを構成する命令との互換性を意識して新たな命令を記述する必要がある．

2.2 Dockerfile のビルドエラーと依存関係

Dockerfile のビルドは他の言語がベースのソフトウェアよりも失敗しやすいことが明らかになっている [5]．そこで，ビルドエラーが発生している Dockerfile の修正を支援するためのツールとして Shipwright が提案されている．Shipwright は，ビルドエラーが発生している Dockerfile のビルドログを分類し，その分類を基にビルドエラーに対して修正をおこなう．しかしながら，Shipwright ではビルドエラーが発生した Dockerfile のうち，18.9%の Dockerfile しか修正できていない．この原因として，Shipwright はビルドエラーの原因を特定した修正をおこなっていないことが考えられる．ベースイメージレイヤの書き換えは不可能であるため，ビルドエラーの根本原因がベースイメージ内に存在する場合，ビルドエラーを直接的に解消するのではなく，何らかの回避策によってビルドエラーを解消する必要がある．つまり，Shipwright が正しくビルドエラーを修正可能になるためには，まず，そのエラーが Dockerfile の直接的な修正で対応が可能なのか，FROM 命令で指定している既存イメージを考慮した変更が必要なのかに分類することが必要であると言える．

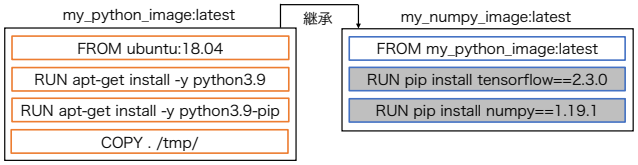

図 4. レイヤレベルの依存関係でのビルドエラー

Dockerfile が持つ依存関係はカスタムイメージレイヤ間で発生する依存関係と，ベースイメージレイヤとカスタムイメージレイヤ間で発生する依存関係に大別できる．

レイヤレベルの依存関係：カスタムイメージレイヤ内に存在するレイヤ同士の依存関係である．つまり，開発者が記述した命令間で発生する依存関係である．レイヤレベルの依存関係が原因で発生するビルドエラーの例を図 4 に示す．図 4 では，“my_python_image” という名前の “latest” タグを持ったイメージをベースイメージとすることで “my_numpy_image” イメージを作成している．しかし，灰色で示す tensorflow のバージョン 2.3.0 と numpy のバージョン 1.9.1 との間に互換性がないため，ビルドエラーが発生する．

イメージレベルの依存関係：ベースイメージレイヤ内のあるレイヤとカスタムイメージレイヤのあるレイヤの間で発生する依存関係である．イメージレベルの依存関係が原因で発生するビルドエラーの例を図 5 に示す．図 5 の場合，ベースイメージレイヤでは python3.9 をインストールする命令が記述されており，カスタムイメージレイヤでは numpy1.14 をインストールするする命令が記述されている．しかし，python3.9 と numpy1.14 には互

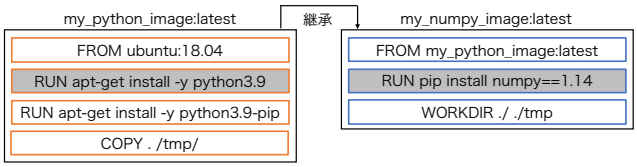

図 5. イメージレベルの依存関係でのビルドエラー

換性がないため，ビルドエラーが発生する．また，図 3 のように継承がネスト構造になっている場合があり，開発者が指定したベースイメージがさらに継承しているイメージの各レイヤとの依存関係も存在する．

Docker Hub 上のイメージは，Dockerfile やドキュメントと共に管理することができるが，Dockerfile と共に管理されているイメージは多くない [6]．Docker Hub からベースイメージのみを取得した場合，開発者はベースイメージの具体的なファイル構成，バージョン情報を意識できない可能性がある．結果的に，カスタムイメージレイヤ内の命令とベースイメージレイヤ内の命令との互換性を意識した修正が困難になると考えられる．そこで本研究では，ビルドエラーの自動修正手法の性能向上に向けて，2.2 節で定義した 2 つの依存関係に着目し，それぞれの依存関係がビルドエラーに与える影響を調査する．

3. 分析

本研究では，Dockerfile の依存関係がビルドエラーに与える影響を明らかにするため，以下の 2 つのリサーチクエスチョンに取り組む．

RQ1：レイヤレベルの依存関係とイメージレベルの依存関係が原因のビルドエラーはどの程度存在するか
RQ2：レイヤレベルの依存関係とイメージレベルの依存関係が原因のビルドエラーはどの程度の時間で修正されているか

3.1 データセット

本研究では Shipwright で使用されたデータセットを使用する．Shipwright では，binnacle データセット [7] を拡張したものを使用しており，ルートディレクトリに単一の Dockerfile が存在する 20,526 件の GitHub リポジトリを取得している．また Shipwright では，すべての Dockerfile に対してビルドをおこなっており，データセット内の Dockerfile に対応するビルドログも提供されている．本研究で扱うデータセットは，20,526 件のリポジトリから，以下の条件に合うデータを除外する．

1. 破損しているデータを持つリポジトリ
2. 現在は存在していないリポジトリ
3. マルチステージビルドを利用している Dockerfile を持つリポジトリ

Docker では 2017 年 5 月以降，マルチステージビルドと呼ばれる機能がサポートされている．マルチステージビルドとは，イメージサイズを小さくするために，ビルド途中で生成される不要な生成物を取り除くことを効率的に実現する機能である．マルチステージビルドでは複数の FROM 命令が記述され，各 FROM 命令でビルドするステージが異なる．以前に実行されたステージの実行結果から，イメージに必要な生成物のみを次のステージに引き継ぐことが可能である．そのため依存関係がベースイメージレイヤ間で発生する可能性があり，1 章で定義した Dockerfile が持つ依存関係とは別の依存関係が発生している可能性があるため，今後更なる調査が必要であるが，本研究では分析対象外とする．

これらの Dockerfile をフィルタリングした結果，16,099 件のリポジトリを本研究のデータセット（表 1）として扱う．16,099 件のうち，4,440 件の Dockerfile でビルドエラーが発生しており，本データセットにおける Dockerfile のビルドエラー発生率は 27.5%である．

表 1. 本論文のデータセット

Dockerfile 数	ビルドエラー件数	ビルドエラー率
16,099 件	4,440 件	27.5%

3.2 RQ1：レイヤレベルの依存関係とイメージレベルの依存関係が原因のビルドエラーはどの程度存在するか

動機：Shipwright ではビルドエラーが発生している Dockerfile のうち，18.9%しか自動修正できていない．ビルドログにはカスタムイメージレイヤに対応する命令の実行結果しか記録されていないため，ベースイメージレイヤとカスタムイメージレイヤの間の依存関係により発生したビルドエラーは見つけることができない．Shipwright

```
Sending build context to Docker daemon  274.9MB

Step 1/57 : ARG UBUNTU_VERSION
Step 2/57 : ARG CUDA_VERSION
Step 3/57 : ARG CUDNN_VERSION
Step 4/57 : FROM nvidia/cuda:${CUDA_VERSION}-cudnn${CUDNN_VERSION}-devel-ubuntu${UBUNTU_VERSION}
```

図 6. レイヤレベルの依存関係が原因で発生しているビルドエラーのビルドログ

```
Sending build context to Docker daemon   34.3kB

Step 1/9 : FROM python:3-alpine
3-alpine: Pulling from library/python
df20fa9351a1: Already exists
36b3adc4ff6f: Already exists
7031d6d6c7f1: Already exists
81b7f5a7444b: Already exists
0f8a54c5d7c7: Already exists
Digest: sha256:c5623df482648cacece4f9652a0ae04b51576c93773ccd43ad459e2a195906dd
Status: Downloaded newer image for python:3-alpine
 ---> 8ecf5a48c789
Step 2/9 : RUN apt-get update   && apt-get upgrade -yqf   && apt-get install -yqq build-essential
 ---> Running in 5f58dd070d65
/bin/sh: apt-get: not found
```

図 7. イメージレベルの依存関係が原因で発生しているビルドエラーのビルドログ

の手法でビルドエラーの修正率が低いのは，イメージレベルの依存関係がビルドエラーの原因の多くを占めている可能性がある．そのため，まずビルドエラーの原因うち，本研究で定義した 2 種類の依存関係がどの程度占めているのかを明らかにし，依存関係を修正することが自動修正手法の性能向上につながるのかを分析する必要がある．

分析方法：ビルドエラーが発生している 4,440 件のビルドログを目視することにより，ビルドエラーの原因をレイヤレベルの依存関係，イメージレベルの依存関係，その他の 3 つに分類する．例として，レイヤレベルの依存関係が原因で発生しているビルドエラーのビルドログを図 6 に示す．図 6 では，ARG 命令を用いて環境変数を FROM 命令に渡そうとしているが，環境変数の受け渡しがうまくおこなわれていないため，発生するエラーである．このように，ビルドエラーが発生した命令行自体にエラーの根本原因がなく，ビルドログ内にその原因を特定できる命令が存在する場合，レイヤレベルの依存関係として扱う．

イメージレベルの依存関係が原因で発生しているビルドエラーのビルドログの例を図 7 に示す．図 7 では，ベースイメージの OS に Alpine Linux が使用されておりパッケージマネージャは APK が採用されているにもかかわらず，Dockerfile 内で debian 系の Linux ディストリビューションで採用されているパッケージマネージャの APT の apt-get コマンドを用いてパッケージをインストールしようとしている．この場合，ベース OS に採用されている OS の種類を確認せずに，パッケージインストールコマンドを記述したことにより，ビルドエラーが発生したと考えられる．このように，Dockerfile 内の命令でベースイメージの機能に関する操作をおこなっており，その行でビルドエラーが発生している場合，イメージレベルの依存関係として扱う．

表 2. 依存関係が原因でビルドエラーが発生した Dockerfile の数

ビルドエラーの原因	Dockerfile 数	割合
レイヤレベルの依存関係	621	14.0%
イメージレベルの依存関係	2,546	57.3%
その他	1,273	28.7%
合計	4,440	100.0%

結果：4,440 件のビルドログを目視した結果を表 2 に示す．表 2 より，レイヤレベルの依存関係が原因で発生しているビルドエラーが 621 件（14.0%），イメージレベルの依存関係が原因で発生しているビルドエラーが 2,546 件（57.3%），その他が原因で発生しているビルドエラーが 1,273 件（28.7%）存在することがわかった．その他が原因で発生しているビルドエラーとは，命令の記述の誤りによるものや，変更が必要な外部ファイルによるものなどである．命令の記述の誤りは，引数やオプションの指定が記述規則に則っていないことがビルドエラーの原因になる．変更が必要な外部ファイルとは，シェルスクリプトや Makefile などの外部ファイルを指し，外部ファイルの誤りがビルドエラーの原因となる．本研究では Dockerfile 以外のファイルを修正することを想定していないため，外部ファイルの誤りが原因となるビルドエラーは本研究の対象外とし，“その他” へ分類している．

RQ1 の結果から，Dockefile のビルドエラーは，イメージレベルの依存関係が原因で発生するビルドエラーが原因の半数以上を占めていることがわかった．つまり，ベースイメージレイヤとカスタムイメージレイヤの命令間に互換性がないことがビルドエラーの主な原因となっている．一方で，レイヤレベルの依存関係が原因，つまり，カスタムイメージレイヤ内のある 2 つの依存関係によって引き起こされるビルドエラーは少数である．これらの結果より，ベースイメージレイヤがビルドエラーにもたらす影響が大きいことが明らかになった．

図 8. Dockerfile の修正時間の取得方法の例

3.3 RQ2：2 つの依存関係はどの程度の時間で修正されているか

動機：RQ2 では修正が必要な Dockerfile を特定するために分析をおこなう．Dockerfile の修正には多くの時間が費やされることが明らかになっている [1] が，その中でも特に原因特定や修正に時間のかかる種類のビルドエラーに対して自動修正をおこなうことで，ソフトウェアの迅速なデリバリーを実現するための支援がおこなえると考える．特に，Dockerfile が持つ 2 つの依存関係のうちイメージレベルの依存関係は，ビルドエラーの根本原因が Dockerfile 内に記述されていないため，原因の特定により多くの時間がかかる可能性がある．

分析方法：RQ2 ではデータセットに含まれる各リポジトリのメタ情報を用いて，GitHub API[3]で各リポジトリの Dockerfile に対するコミットを全て取得する．Shipwright のデータ収集の直前におこなわれた Dockerfile へのリビジョンを特定し，その 1 つ後の Dockerfile へのリビジョンとの時間差を Dockerfile の修正時間として計算する．図 8 に RQ2 での Dockerfile の修正時間の取得方法を示す．図 8 では，Dockerfile が ver.2 のときにデータ取得をおこなっている．このとき ver.2 の Dockerfile の修正時間は，rev.2 のコミットがおこなわれた時間と rev.3 のコミットがおこなわれた時間の時間の差とする．

一般的にビルドエラーが発生している Dockerfile がリポジトリにある場合，その Dockerfile への変更内容はビルドエラーの修正である可能性が非常に高い．そのため，データセットの Dockerfile に対するコミットと，その直後のコミットを Dockerfile の修正時間として計算する．また，Google の研究 [8] で提案されているように，結果の公平性を保つためにビルドエラーの修正に 12 時間以上の時間を要しているビルドエラーを除外する．その後，修正時間の四分位範囲の 1.5 倍を上下限を外れ値として除去する．事前調査の分析結果を基にレイヤレベルの依存関係とイメージレベルの依存関係によるビルドエラーの修正時間の差を比較する．なお，RQ2 では，データセット内の Dockerfile が収集された時点以降変更されていない Dockerfile については，修正時間を計算することができないため，RQ2 では分析対象外としている．各ビルドエラーの修正時間を算出した後，レイヤレベルの依存関係が原因のビルドエラーとイメージレベルの依存関係が原因のビルドエラーの間に統計的な有意差があるのかについて統計的検定をおこなう．

[3]https://docs.apitester.org/guides/github-graphql-api-guide

図 9. 依存関係別のビルドエラーの修正時間の分布

結果：依存関係別のビルドエラーの修正時間の分布を表す箱ひげ図を図 9 に示す．また，箱ひげ図に示した各依存関係が原因で発生しているビルドエラーでの標本数，および修正時間の平均値と中央値を表 3 に示す．

表 3 より，レイヤレベルの依存関係によるビルドエラーの修正時間の平均値は 6.9 時間，中央値は 5.8 時間であった．一方，イメージレベルの依存関係によるビルドエラーの修正時間の平均値は 7.5 時間，中央値は 6.3 時間であった．つまり，レイヤレベルの依存関係が原因のビルドエラーの修正時間に比べ，イメージレベルの依存関係が原因のビルドエラーの修正時間は約 1 時間長いことがわかる．また，平均値についても，イメージレベルの依存関係が原因で発生しているビルドエラーの修正時間の方が約 0.6 時間長いことがわかる．

マンホイットニーの U 検定を用いて 2 つの依存関係が原因のビルドエラーの修正時間に統計的有意差（有意水準 1%）があるかを調査した結果，p 値が 0.430 となり，2 つの依存関係およびその他が原因のビルドエラーの修正時間に有意差はないことが明らかになった．これらの結果より，依存関係の種類により修正時間に有意な違いはないが，RQ1 の結果を踏まえるとイメージレベ

表 3. 依存関係が原因で発生するビルドエラーの修正時間

ビルドエラーの原因	Dockerfile 数	修正時間の平均値（時）	修正時間の中央値（時）
レイヤレベルの依存関係	88	6.9	5.8
イメージレベルの依存関係	297	7.5	6.3
その他	129	6.3	5.7

ルの依存関係によるビルドエラーに重点を置く方が自動修正率の向上に寄与することがわかった．

4. 考察

4.1 継承の深さとビルドエラーの関係

RQ1 では，イメージレベルの依存関係が，ビルドエラーの原因の半数以上を占めていることが明らかになった．ただし，Dockerfile 内で指定したベースイメージに対応する Dockerfile がさらに別のベースイメージを指定している場合がある．その場合，Dockefile の依存関係が複雑になり，ビルドエラーが発生しやすくなる可能性がある．本節では，Dockerfile 内で指定されたベースイメージの継承の深さがビルドエラーに与える影響について追加調査をおこなう．Dockerfile の継承の深さとは，Dockerfile が継承しているベースイメージの数を意味する．例えば，ベースイメージの FROM 命令にコンテナ内の OS 部分にあたる “ubuntu:latest” が指定されている場合を考える．このとき，“ubuntu:latest” イメージを作成するための Dockerfile の FROM 命令には “scratch” を指定しているため，“ubuntu:latest” のさらなる継承は存在しない．そのため，“ubuntu:latest” を FROM 命令で指定しているイメージの継承の深さは 1 になる．

継承の深さ別のビルドエラー件数とその割合を表 4 と表 5 に示す．表 4 および表 5 より，継承が深くなるほどビルドエラーの発生率が高くなるという関係は存在しないことがわかる．特にイメージレベルの依存関係が原因でビルドエラーが発生する件数は継承の深さが 1 の場合が最も多い（表 5）ことからも，継承の深さよりも継承をおこなうこと自体がビルドエラーを誘発する原因となっていることがわかる．これは継承を活用したイメージ作成を推奨するベストプラクティス[4]とは相反する事象であると思われるが，少なくともベースイメージの選択には細心の注意を払う必要があることを示唆している．

[4]https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

4.2 “latest” タグがビルドエラーに与える影響

前節で，継承の深さはビルドエラーに影響はなく，継承をおこなうこと自体がビルドエラーを引き起こしやすくなる可能性があることがわかった．ただし，継承をおこなう際には，イメージの選択の他に，タグも一緒に選択する必要がある．タグを選択する場合に，“latest” タグもしくはタグ名記載なしで継承をおこなう場合，ビルドするたびに継承するイメージのバージョンが最新化されることで，ビルドエラーが発生する可能性がある．そのため，本節では，イメージレベルの依存関係において “latest” タグがビルドエラーに与える影響が大きいのかについて追加調査をおこなった．“latest” タグを使用している Dockerfile のビルドエラー率は，2 つの依存関係によって異なるのかを調査した．

表 6 に “latest” タグを使用しておりビルドエラーが発生している Dockerfile の件数を，依存関係別に示す．また，表 6 に示した結果に統計的有意差が見られるかを確認する．そのために，本研究で定義した 2 つの依存関係において，“latest” タグでベースイメージを継承している Dockerfile のビルドエラーの発生割合が有意に違いがあるのか，有意水準を 0.01 としてカイ二乗検定をおこなった．その結果，p 値が 0.003 となり，つまり，イメージを “latest” タグで継承することはイメージレベルの依存関係を引き起こしやすいと言える．

4.3 自動修正手法の構築における修正対象

RQ2 では 2 種類の依存関係及びその他が原因で発生するビルドエラーの修正時間において，それぞれの修正時間の間に統計的有意差は見られないことがわかった．しかしながらデータセットに含まれる Dockerfile に，Shipwright がデータ収集をおこなった 2020 年 6 月以降，変更（修正）されていない Dockerfile が多く存在していた．表 7 にデータセット収集以降 Dockerfile を修正していないリポジトリの件数を示す．表 7 より，イメージレベルの依存関係が原因のビルドエラーは，レイヤレベル

表 4. レイヤレベルの依存関係が原因でビルドエラーが発生した Dockerfile の割合（継承の深さ別）

継承の深さ	Dockerfile 数	ビルドエラーが発生した Dockerfile の数	割合
1	3,349	231	6.8%
2	1,517	34	2.2%
3	158	5	3.1%
4	57	4	7.0%
5	309	3	0.9%
6	8	0	0.0%

表 5. イメージレベルの依存関係が原因でビルドエラーが発生した Dockerfile の割合（継承の深さ別）

継承の深さ	Dockerfile 数	ビルドエラーが発生した Dockerfile の数	割合
1	3,349	631	18.8%
2	1,517	236	15.5%
3	158	33	20.8%
4	57	11	19.2%
5	309	45	14.5%
6	8	0	0.0%

の依存関係が原因で発生しているビルドエラーと比べて半数以上が修正されていない．これは，ベースイメージレイヤにまたがって存在する，破損した依存関係の修正は困難であることが原因である可能性が考えられる．そのため，イメージレベルの依存関係を修正する自動修正手法を構築することが期待される．

4.4 妥当性への脅威

4.4.1 内的妥当性

本研究では先行研究で作成されたデータセットを用いている．データセットに含まれる Dockerfile やイメージは先行研究が実施された当時のものであり，現在，Dockerfile をビルドした場合の結果とは異なる可能性がある．

また，本研究では第 1 著者が目視によってビルドエラーの分類をおこなったため，分類結果は著者の判断に依存する．ただし，3.2 節に記載した基準にしたがって分類したため，第三者が分類をおこなったとしても，結論に影響を与えるほどの個人差は発生しないと考えている．

4.5 外的妥当性

本研究では，Shipwright で使用された 20,526 件の GitHub リポジトリからフィルタリングをおこない，16,099 件の GitHub リポジトリとそれに対応する Dockerfile を対象に分析をおこなっている．本論文でおこなったフィルタリングによって，データの性質に偏りが発生している可能性がある．また，コンテナレジストリは DockerHub 以外にも，Amazon Elastic Container Registory(Amazon ECR)[5]や Azure Container Registory[6]，Google Container Registory[7]などが存在する．Docker で使用されるパブリックコンテナレジストリのデファクトスタンダードは Docker Hub であり，最も多く利用されているコンテナレジストリサービスの 1 つである [9].

5. 関連研究

5.1 Dockerfile の品質問題

近年，Dockerfile の品質に関する研究が多くされているが，特にソフトウェア保守の観点から，品質に影響を与えるようなソースコードや実装に関する研究が盛んにおこなわれている．Dockerfile の品質に影響を与えるようなソースコードや実装は将来的にビルドエラーを引き起こす可能性があり，これらを事前に検出し修正するこ

[5]https://aws.amazon.com/jp/ecr/
[6]https://azure.microsoft.com/ja-jp/products/container-registry/
[7]https://cloud.google.com/container-registry

表 6. ビルドエラーが発生した “latest” タグを使用している Dockerfile 数

依存関係の種類	Dockerfile 数	ビルドエラーが発生した Dockerfile	割合
レイヤレベル	621	43	6.90%
イメージレベル	2,546	275	10.80%

表 7. Dockerfile が修正されていないリポジトリの件数

依存関係の種類	Dockerfile 数	修正されていない Dockerfile	データセット全体に対する割合
レイヤレベル	621 件	461 件	22.4%
イメージレベル	2,546 件	2,053 件	46.2%

とでビルドエラーを未然に防ぐことができる．

Wu ら [10] は，GitHub 上に存在する Dockerfile のうち約 84%に非推奨とされる命令記述（コードスメル）が存在することを示した．また，プロジェクトが作成された年数やプロジェクトで使用されるプログラミング言語によっても，コードスメルの数や種類に違いがあることを発見した．一方，Eng ら [11] は，7 年間にわたる Dockerfile の変更履歴から Dockerfile の記述方法の変化についての知見を提供している．近年では，コードスメルを持つ Dockerfile の数が減少しており，開発者は Docker 社が提供する Dockerfile 記述のためのベストプラクティスに従う傾向が強い可能性があることを示した．Azuma ら [12] は，開発者がコメント文でアドホックに記載したコードの改善策（Self-Admitted Technical Debt: SATD）が Dockerfile 内にどの程度存在しているかを調査した．その結果，Dockerfile に含まれるコメントのうち，約 3.0%が SATD の存在を示唆していることが明らかになった．また，Dockerfile 中の SATD は 5 つのクラスと 11 のサブクラスに分類可能であり，Dockerfile 特有の SATD が存在することも明らかになった．

Dockerfile においてビルドエラーが発生した際には，コードスメルや SATD の種類からビルドエラーの原因を理解することで，Dockerfile の自動修正手法の性能向上への手掛かりにできる可能性がある．

5.2 ビルドエラーの自動修正

Hassan ら [13] は，ビルドスクリプトを自動修正するための手法として HireBuild を提案している．HireBuild では，既存のビルドスクリプトの修正から自動生成した修正パターンとビルドログの類似性に基づいて，ビルドエラーが発生しているビルドスクリプトに修正パッチを自動生成している．その結果，175 件のビルドエラーから，135 件の修正パターンを抽出した．この修正パターンは，再現性のあるビルドエラーのうち，45%を手動修正と同程度の時間で自動修正することができた．

Dockerfile の自動修正に関する研究は，2.2 節で紹介した Shipwright のみである．Shipwright では，Dockerfile のビルドエラーが発生した際に得られるエラーメッセージをルールベースで分類し，その分類を基に修正パッチを手動で生成している．それにより，13 個の修正パターンと 50 個の修正の提案を作成している．その結果，自動で修正をおこなうことができた Dockerfile は 18.9%である．また，ビルドエラーが発生している Dockerfile を持つプロジェクトに対して，修正パッチの提案をおこなった結果，45 件中 19 件の修正パッチが受け入れられた．

本論文では，Dockerfile が持つ依存関係を考慮した Dockerfile の自動修正手法の構築へ向けた事前分析をおこなった．依存関係を考慮した自動修正手法と，[13] や [2] を組み合わせることで，自動修正手法の性能向上が期待できる．

6. おわりに

本論文では，Dockerfile の自動修正手法の性能向上に向けたビルドエラーを引き起こす Dockerfile の依存関係について分析をおこなった．RQ1 では，イメージレベルの依存関係が Dockerfile のビルドエラーの大半を占めていることが明らかになった．RQ2 では，依存関係が原因のビルドエラーを修正するのに要する時間を調査した．その結果，レイヤレベルの依存関係によるビルドエラーの修正時間は約 5.8 時間，イメージレベルの依存関係によるビルドエラーの修正時間は約 6.3 時間であることが分かったが，それぞれのビルドエラーの修正時間に統計

的な有意差は見られなかった．一方，ビルドエラーが発生しているDockerfileのうち修正がおこなわれていないDockerfileが多く存在していた．

RQ1とRQ2の結果を合わせると，Dockerfileのビルドエラーは，ベースイメージレイヤとDockerfile内の命令の互換性によるものが半数以上を占め，そのようなビルドエラーは修正が困難である可能性があることが明らかになった．今後は，修正が困難なDockerfileのビルドエラーが多数存在する可能性があり，今後依存関係ごとのビルドエラーの内容分析を通じてビルドエラーを体系化し，自動修正手法を改善する予定である．

謝辞

本研究の一部は，文部科学省科学研究補助金（基盤（C）：22K11974）による助成を受けた．

参考文献

[1] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An Empirical Study of Build Failures in the Docker Context. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR ' 20)*, pp. 76–80, 2020.

[2] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps. Shipwright: A human-in-the-loop system for dockerfile repair. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE '21)*, pp. 198–199, 2021.

[3] Michael Ferranti. 2017 annual container adoption survey: Huge growth in containers. `https://portworx.com/blog/2017-container-adoption-survey/`. Accessed on Febrary 22, 2023.

[4] Microsoft. コンテナー イメージのタグ付けとバージョン管理に関する推奨事項. `https://learn.microsoft.com/ja-jp/azure/container-registry/container-registry-image-tag-version`. Accessed on March 6, 2023.

[5] Jurgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, pp. 323–333, 2017.

[6] Hasan Ibrahim, Mohammed Sayagh, and Ahmed E.Hassan. Too many images on dockerhub! how different are images for the same system? *Empirical Software Engineering*, Vol. 25, No. 5, pp. 4250–4281, 2020.

[7] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. A Dataset of Dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pp. 528–532, 2020.

[8] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 724–734, 2014.

[9] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER' 19)*, pp. 491–501, 2019.

[10] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. *IEEE Access*, Vol. 8, pp. 34127–34139, 2020.

[11] Kalvin Eng and Abram Hindle. Revisiting dockerfiles in open source software over time. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR '21)*, pp. 449–459, 2021.

[12] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering*, Vol. 27, No. 2, 2022.

[13] Foyzul Hassan and Xiaoyin Wang. HireBuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pp. 1078–1089, 2018.

[14] Inc. Docker. Docker official images. `https://docs.docker.com/docker-hub/official_images/`. Accessed on Febrary 22, 2023.

[15] Yinyuan Zhang, Yang Zhang, Xinjun Mao, Yiwen Wu, Bo Lin, and Shangwen Wang. Recommending Base Image for Docker Containers based on Deep Configuration Comprehension. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '22)*, pp. 449–453, 2022.

[16] Chris Tomy, Tingmao Wang, Earl T. Barr, and Sergey Mechtaev. Modus: A Datalog dialect for building container images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE '22)*, pp. 595–606, 2022.