Pythonベースイメージの事前pullによるDockerイメージのビルド時間の短縮

加藤 健吾[1]　遠藤 睦実[1]　串田 高幸[1]

概要：Dockerfile を記述する際，ベースイメージを指定する．ローカル内に，指定したベースイメージが存在しない場合，Docker イメージをビルドする際にベースイメージの pull が行われる．そのため，この pull 時間の分だけ，ビルド時間が増加してしまう．イメージの容量が pull 時間に与える影響を調べるために，基礎実験を行った．その結果，イメージの容量の増加に伴って，pull 時間が増加することが分かった．そのため，Docker 及び Python を用いた開発において Docker イメージをビルドする際に，Python ベースイメージを pull する時間が長いことによって作業が停滞してしまい，開発が円滑に進まなくなってしまうという課題がある．本稿ではこの課題を解決するために，Python ベースイメージを，開発者が Python のソースコードを記述している時に事前に pull するソフトウェアである，Pullior を提案する．Pullior によってビルド時に行われるベースイメージの pull を省略し，ビルド時間を短縮することができる．本提案を評価するために，Github から QBittorrentBot というリポジトリを clone し，これに対して提案を適用することで，提案適用前と適用後のビルド中の pull 時間及びビルド時間を比較した．その結果，提案適用前は 11.1 秒と 22.4 秒，提案適用後は約 0.0 秒と 8.7 秒であり，提案適用後のビルド時間を，提案適用前のビルド時間に比べて約 61%短縮することができた．

1. はじめに

背景

Docker コンテナを作成するために，Docker イメージがテンプレートとして使用される [1]．Docker イメージは，レイヤ構造で構成されている [2]．まずベースイメージという，イメージのベースとなるイメージがあり，その上に差分としてアプリケーションの追加やファイルの追加のレイヤが積み重なっている．Docker イメージは，Dockerfile と呼ばれる一連のセットアップ命令によって作成される [3]．ビルド時に，ローカル内に Dockerfile で指定されたベースイメージが存在しない場合，そのベースイメージの pull が行われる．しかし，ローカル内に既に指定されたベースイメージが存在している場合，そのベースイメージが利用されるため pull は省略される [4]．そのため，ローカル内に，ベースイメージに指定されたイメージがない場合と比べ，Docker イメージのビルド時間が短縮される．

イメージを pull する際にタグを指定することができ，タグ機能の一部として，そのイメージのバージョンを指定することができる [5]．latest と指定すると，そのイメージの最新バージョンが指定される．バージョンの他にも，指定できるタグは存在するが，イメージによって異なる．例として Python を取り上げる．Python イメージには，bookworm や bullseye，buster といった Debian のコードネームや，slim や alpine といったそのイメージの軽量版をタグをして指定することができる[*1]．イメージの例として，python:3.9-bookworm のようなイメージがある．タグを指定しないこともできるが，その場合自動的に最新バージョンが pull される [6]．

課題

Python ベースイメージを pull する時間が長い．これによって，Docker 及び Python を用いた開発において Docker イメージをビルドする際に作業が停滞してしまい，開発が円滑に進まなくなってしまうという課題がある [7]．原因として，ローカル内に Dockerfile で指定されたベースイメージが存在しない場合，ビルド中にベースイメージの pull が行われることが挙げられる．これを図 1 に示している．図 1 の左側の状況では，Dockerfile で指定されている Python ベースイメージがローカル内に存在しているため，利用可能である．一方で図 1 の右側の状況では，Dockerfile で指定されている Python ベースイメージがローカル内に存在

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1

[*1] https://hub.docker.com/_/Python/tags

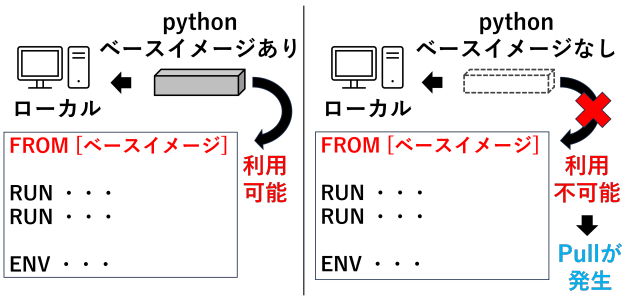

図 1 ベースイメージの pull の発生

していないため，利用不可能であることから，pull が発生する．この時，ベースイメージの容量が大きいほど pull 時間は長くなる．そのため容量の大きいベースイメージを用いる場合，Docker イメージのビルド時間が長くなってしまい，開発が円滑に進まなくなってしまう．

基礎実験

基礎実験として，イメージの容量が pull 時間に与える影響を調査した．Python イメージを用いて実験を行った．実験内容は，python:3.9 イメージと python:3.9-slim イメージの pull 時間の計測をそれぞれ 10 回行い，それぞれの平均値を求める．表 1 はその結果である．python:3.9 イメージと python:3.9-slim イメージの pull 時間の平均は，それぞれ 45.67 秒と 12.57 秒であった．また容量は，997MB と 126MB であった．この結果から，イメージの容量の増加に伴って，pull 時間が増加することが言える．

表 1 Python イメージの容量と pull 時間の比較

イメージ名	容量 (MB)	pull 時間 (s)
Python:3.9	997	45.67
Python:3.9-slim	126	12.57

各章の概要

本稿は以下のように構成される．第 2 章では，本稿の関連研究について述べる．第 3 章では，本稿で挙げた課題を解決するための提案について述べる．第 4 章では，提案した手法の実装について述べる．第 5 章では，提案手法に対しての実験内容と，その評価について述べる．第 6 章では，提案手法の議論を述べる．第 7 章は，本稿のまとめである．

2. 関連研究

入力ファイルをキャッシュし，ローカルのイメージデータを利用することで，イメージ構築中のリモートファイル取得時間を大幅に削減する FastBuild を提案した研究がある [4]．イメージ構築中に複数回利用されるファイルが多数存在するため，FastBuild ではローカルバッファを使用して最近アクセスしたファイルをキャッシュすることで，システムの構築速度を最大 10 倍向上させた．しかし，本研究では入力ファイルの取得を最小限にするために，イメージを pull するのではなく，Dockerfile をダウンロードしてビルドしている．そのため，ビルドに時間がかかる Dockerfile である場合に，かえって全体の構築時間が増加してしまう．

コンテナの起動に必要不可欠なファイルのみを先にダウンロードし，コンテナデプロイを早い段階で行い，コンテナ起動後にその他のファイルをダウンロードすることで，コンテナが機能を提供できるようになるまでの時間を大幅に短縮した研究がある [8]．コンテナの起動に必要不可欠なファイルの特定には，fatrace を利用したコンテナのプロファイリングを用いている．しかし，本研究ではコンテナのデプロイ時にコンテナが有用な機能を実行できるようになるまでの時間を短縮することが目的であるため，イメージのビルド時間は短縮しない．

Docker Distribution をベースにした FID という P2P ベースの大規模イメージ配信システムを実装し，Docker レジストリの負担を分散することで，イメージ配布時間を短縮した研究がある [9]．本研究では，大規模なコンテナデプロイメントにおける，イメージ配布時間の短縮，Docker レジストリのネットワークトラフィックの削減を目的としている．しかし本研究では，ローカル内にベースイメージに指定されたイメージが存在しないときに，イメージのビルド中に pull が発生してしまうことを防ぐことができない．

3. 提案

提案方式

本稿では，Docker イメージのビルド時に行われるベースイメージの pull を省略し，ビルド時間を短縮することを目的とした．この目的を達成するために，Dockerfile で指定されるベースイメージとなる Python イメージを，開発者が Python のソースコードを記述している間に事前に pull するソフトウェアを提案する．本提案は，開発者は Visual Studio Code(以下 VSCode) を用いて開発を行っており，記述している Python のソースコードは VSCode の自動保存機能によって定期的に保存されるという状況を想定している．図 2 は本提案のアルゴリズムである．まず Python のソースコードが保存されたタイミングで，仮想環境内の Python バージョンを取得する．そして取得したバージョンをタグとして用い，python:[取得したバージョン]-bookworm という形でバージョンを固定したイメージを Docker Hub から pull する．その後，pull したイメージをベースイメージとして指定するコマンドのみを記述した Dockerfile を作成する．

本提案では，ベースイメージの事前 pull を行うタイミングとして，Python ソースコードが保存されたタイミングを用いた．ソースコードの記述中は，VSCode の自動保存機能によって定期的に保存が行われる．これにより，開発者がソースコードを記述している時間を使って事前 pull を

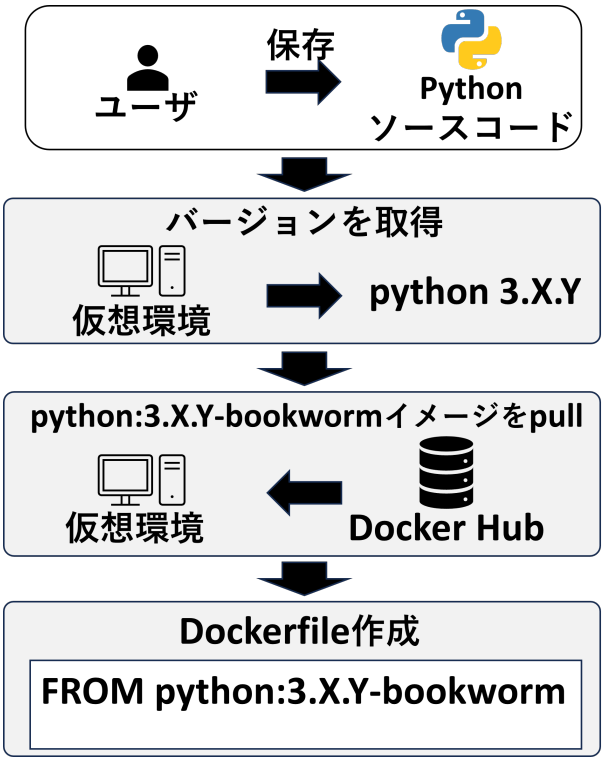

図 2　提案の流れの図

行うことができるので，全体としてビルド時間を短縮することができる．

ユースケース・シナリオ

本稿のユースケース・シナリオでは，機械学習を研究している研究者が研究で用いるソフトウェアを，言語は Python，開発環境は VSCode を用いて，venv によって開発用の仮想環境を作成し，その仮想環境内で開発する状況を想定している．VSCode では自動保存機能を使うことによって，ソースコードが定期的に保存されるようになっている．

提案を適用する前のユースケースを，図 3 に示す．まず

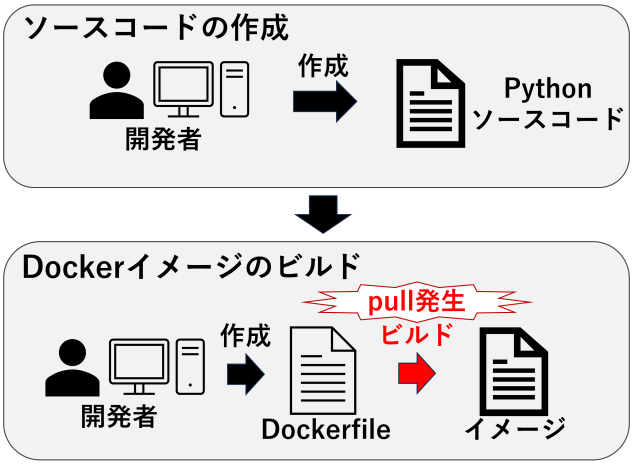

図 3　提案適用前のユースケース

開発者が Python のソースコードを作成する．その後，開発者は Dockerfile を作成し，それを用いて Docker イメージのビルドを行う．ビルドを行う際，ローカル内にベースイメージとして指定されているイメージが存在していなかったため，pull が発生している．

次に，提案を適用した後のユースケースを図 4 に示す．まず，開発者が Python のソースコードを作成している間に

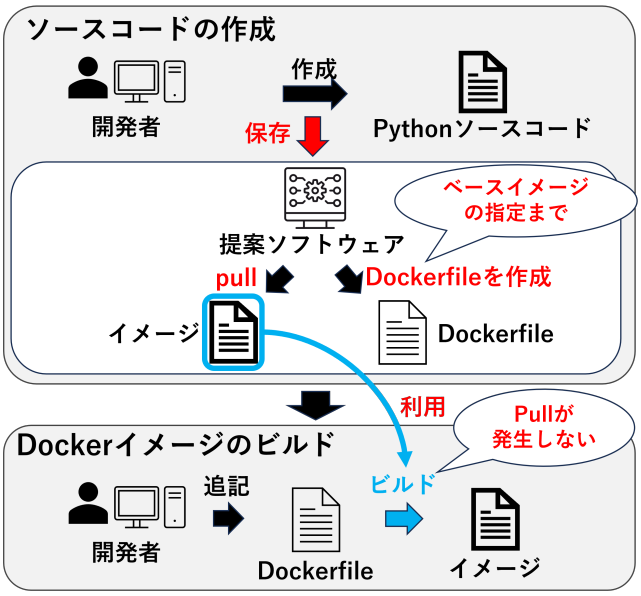

図 4　提案適用後のユースケース

ソースコードが保存されると，そのタイミングで提案ソフトウェアによってイメージの pull が行われる．また，ベースイメージを指定するコマンドのみが記述された Dockerfile を生成する．そこでは，pull してきたイメージを指定する．これによって，開発者は生成された Dockerfile に必要なコマンドを追記していくだけで良くなり，ビルド時には pull が発生しないため，ビルド時間中の待機時間が減少する．

4.　実装

本提案を実装した，Pullior というソフトウェアを作成した．Pullior は，図 5 のような構成になっている．開発言語には Python を用いた．また，Pullior は，開発者の開発用仮想環境内に配置する． Pullior は以下の①から⑤のような流れで処理を行う．

① modi_watcher.py にて，仮想環境内の Python ファイルを監視する．

② VSCode の自動保存機能によって，.py ファイルが保存される．

③ modi_watcher.py が，watchdog モジュールの Pattern-MatchingEventHandler によって②を検知する．

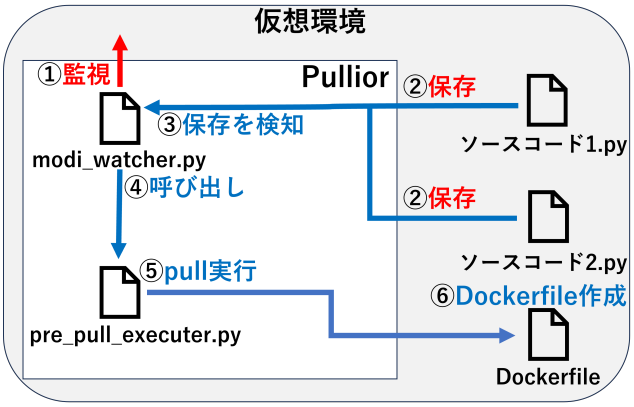

図 5 Pullior の構成図

④ modi_watcher.py が pre_pull_executer.py を呼び出す．

⑤ pre_pull_executer.py が，os モジュールを用いて docker pull コマンドを実行し，ベースイメージの事前 pull を行う．

⑥ pre_pull_executer.py が，リポジトリ直下に Dockerfile が存在するかどうかを判断し，存在しない場合は Dockerfile を作成する．

modi_watcher.py での監視には，Python の watchdog モジュールを用いた．監視対象の拡張子を.py に設定することで，Python ファイルを監視するようにした．監視位置は，Pullior の親フォルダを指定することで，仮想環境内を監視するようにした．

図 6 は，modi_watcher.py によって呼び出される pre_pull_executer.py の動作を表している．pre_pull_executer.py では，処理中に get_pyversion.py，

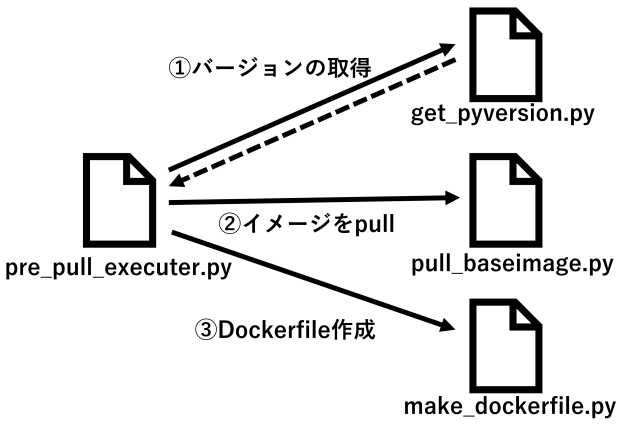

図 6 pre_pull_executer.py の動作

pull_baseimage.py，make_dockerfile.py の 3 つの Python ファイルを呼び出す．get_pyversion.py では，仮想環境内の Python バージョンを取得し，pre_pull_executer.py にバージョンを返す．pull_baseimage.py では，仮引数として Python バージョンを受け取り，そのバージョンを用いてイメージの pull を行う．make_dockerfile.py では，Dockerfile の作成を行う．まず，get_pyversion.py を呼び出して，仮想環境内での Python バージョンを取得する．バージョンの取得には，subprocess モジュールを使って，python -V コマンドを実行した．次に，pull_baseimage.py を呼び出し，イメージの pull を行う．取得した Python バージョンを 3.X.Y として，python:3.X.Y-bookworm イメージを pull する．pull の実行には，os モジュールを用いて docker pull コマンドを実行した．次に，make_dockerfile.py を呼び出し，Dockerfile の生成を行う．まず，os モジュールを用いて，リポジトリ直下に Dockerfile が存在するかどうかを確認する．Dockerfile が存在しなければ，Dockerfile を生成し，pull したイメージをベースイメージとして指定するコマンドのみ記述する．

5. 評価実験

評価実験として，Dockerfile を用いてそのままビルドした場合と，Pullior によって事前にベースイメージを pull しておいた状態でビルドした場合の，ビルド中の pull 時間とビルド時間を測定し，比較した．pull 時間及びビルド時間の測定では，提案適用前と適用後それぞれの場合において 10 回のビルドを行い，それぞれの平均値を求めた．ビルドの実行と実行の間では，docker builder prune -f コマンドでキャッシュを削除することによって，キャッシュの利用によるビルド時間への影響を排除した．実験は以下の①から④のような手順を 10 回ずつ行った．なお，提案適用後の pull 時間及びビルド時間の計測には，最初に Pullior によって事前にベースイメージを pull しておいた後に以下の手順を実行した．

① Dockerfile を用いて Docker イメージをビルドする．

② ビルド中の pull 時間及びビルド時間を計測する．

③ ビルドした Docker イメージの削除を行う．

④ docker builder prune -f コマンドでキャッシュを削除する．

実験環境

実験を行うリポジトリとして，VM 内に evaluation_ex フォルダを作成し，このリポジトリに仮想環境を作成した．仮想環境内の Python バージョンは 3.11.7 に固定した．図 7 に実験環境のディレクトリ構造を示す．リポジトリには，

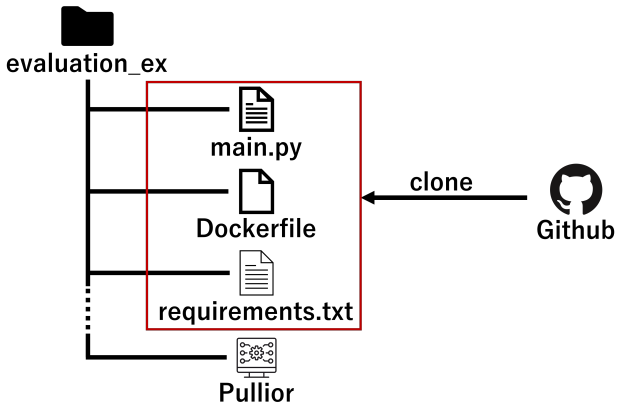

図 7 実験環境のディレクトリ構造

Github から QBittorrentBot[*2] を clone し，QBittorrentBot フォルダ以下の全てのフォルダやファイルをリポジトリ直下に配置し，main.py を，開発者が作成しているソースコードと想定した．QBittorrentBot を使用した理由は以下の通りである．QBittorrentBot の Dockerfile において，指定されているベースイメージが python イメージである．タグとして slim や alpine などの bookworm 以外のタグを使用していない．バージョンが 3.11 で，保守されていないバージョンを用いていない．これらの理由から，QBittorrentBot が Pullior の動作に影響を与えないため，実験に使用した．また，Pullior はリポジトリ直下に配置した．また，VM の構成を以下に示す．

・VM構成
　OS:Ubuntu-22.04
　vCPU 2 コア
　RAM 2GB
　HDD 25GB

実験結果と分析

図 8 は実験結果を棒グラフにしたものである．縦軸は

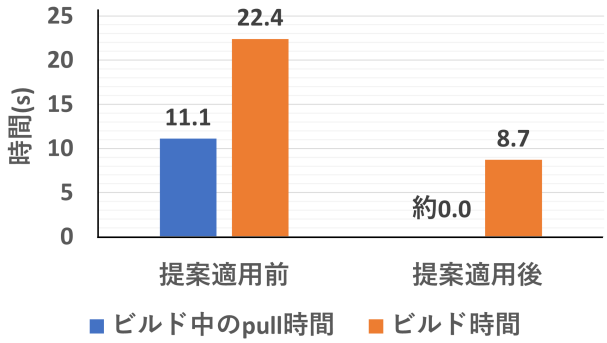

図 8 提案適用前と提案適用後の比較

pull やビルドにかかった時間を表している．横軸は提案適用前か適用後かを表している．提案適用前のビルド中の pull 時間は 11.1 秒，ビルド時間は 22.4 秒であった．一方で，提案適用後のビルド中の pull 時間は約 0.0 秒，ビルド時間は 8.7 秒であった．提案適用前と後のビルド時間を比較すると，提案適用後の方が 13.7 秒早くなっている．また，提案適用後のビルド中の pull 時間が約 0.0 秒になっている．これは，ローカル内にあるイメージが利用されたことで，ベースイメージの pull が省略されているためである．また，約 0.0 秒という表現を使用しているのは，小数第二位において丸め処理が行われているからである．結果を要約すると，提案の適用によってビルド時間を約 61%短縮することができた．

6. 議論

本提案では，ベースイメージを事前に pull することによってビルド時間の短縮を行った．この方法に加え，マルチステージビルドを使用することで，よりビルド時間の短縮が可能である．マルチステージビルドは，ベースイメージを指定するコマンドである FROM コマンドを複数記述することで，それぞれ指定されたベースイメージを使用した別々のビルドとして扱われるというものである[*3]．これにより，ビルドの並列化が可能となり，ビルド時間を短縮できる．マルチステージビルドによって，複数のベースイメージが pull される場合，それぞれを事前に pull することでビルド時間を短縮できる．

本提案では，言語を Python に限定したため，Python 以外の言語を用いた開発を行う場合，本提案を適用できない．本提案を別の言語にも適用するためには，まずユーザが開発に用いる言語を特定する必要がある．特定を行うために，リポジトリ内に src ディレクトリを作成し，ソースコードをそのディレクトリに配置する．そして，src ディレクトリ内の拡張子の数を参照することで特定することができる [10]．src ディレクトリにソースコードを配置することで，目的とする言語以外のファイルの参照を減らすことができる．

本提案では，Python イメージのタグとして，bookworm を用いたが，bullseye や buster のような過去の Debian コードネームを用いた開発を行う場合，本提案を適用できない．本提案をこのようなケースにも適用するためには，開発者がどの Debian コードネームを用いるのかを特定する必要がある．Debian のソフトウェアパッケージとして，Python パッケージが含まれている[*4]．Debian のバージョンによって，サポートされている Python パッケージのバージョンが異なる．そのため，開発者の仮想環境内の Python バージョンが，どの Debian コードネームでサポートされているのかを特定することで，Debian コードネームを特定す

[*2] https://github.com/ch3p4ll3/QBittorrentBot
[*3] https://matsuand.github.io/docs.docker.jp.onthefly/develop/develop-images/multistage-build/
[*4] https://packages.debian.org/ja/bookworm/Python3

ることができる．

本提案では，.py ファイルの保存が実行されるたびに，pull が実行される．そのため，pull 時に出力されるログが保存の回数だけ出力されてしまう．それによって，開発中のログと pull 時のログが混ざってしまう．.py ファイルの保存が実行されるたびに pull が実行されてしまう問題は，ローカル内にあるイメージを docker image ls によって確認して，現在 pull しようとしているイメージが既にローカルに存在する場合は pull を行わないようにすることで解決可能である．しかし，これだけでは pull 時に出力されるログが開発中のログと混ざってしまうという問題が残ってしまう．これは，pull の実行をバックグラウンドで行うことで，ログが出力されないようにすることで解決可能である．

本提案では，pull するイメージを Python:3.X.Y-bookworm のように，バージョン以外の情報を定めてしまっているため，Python イメージのタグとして slim や alpine を指定することができない．そのため，軽量版の Python イメージから，独自でライブラリをインストールして Python 環境を構築したい場合には，本提案を適用することはできない．そこで，仮想環境内にインストールされているライブラリを確認し，タグ無しの Python イメージにも含まれているライブラリが存在するかどうかを確認する．存在する場合，slim または alpine であると判断できる．また，Python の slim イメージの方にも含まれているライブラリが存在する場合，alpine であると判断できる．これによって，Python イメージのタグとして slim や alpine を指定することができるようになる．

7. おわりに

課題は，Docker 及び Python を用いた開発において，Docker イメージをビルドする際に，Python ベースイメージを pull する時間が長いことによって作業が停滞してしまい，開発が円滑に進まなくなってしまう点である．課題を解決するために，Python ベースイメージを，開発者が Python のソースコードを記述している時に事前に pull するソフトウェアを提案した．提案を評価するために，提案適用前と提案適用後のビルド中の pull 時間及びビルド時間を比較した．その結果，提案適用前は 11.1 秒と 22.4 秒，提案適用後は約 0.0 秒と 8.7 秒であり，提案適用後のビルド時間を，提案適用前のビルド時間に比べて約 61%短縮することができた．

参考文献

[1] Ayed, A. B., Subercaze, J., Laforest, F., Chaari, T., Louati, W. and Kacem, A. H.: Docker2RDF: Lifting the Docker Registry Hub into RDF, *2017 IEEE World Congress on Services (SERVICES)*, pp. 36–39 (online), DOI: 10.1109/SERVICES.2017.15 (2017).

[2] Lu, Z., Xu, J., Wu, Y., Wang, T. and Huang, T.: An Empirical Case Study on the Temporary File Smell in Dockerfiles, *IEEE Access*, Vol. 7, pp. 63650–63659 (online), DOI: 10.1109/ACCESS.2019.2905424 (2019).

[3] Henkel, J., Silva, D., Teixeira, L., Marcelod' Amorim, Reps, T.: Shipwright: A Human-in-the-Loop System for Dockerfile Repair, *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1148–1160 (online), DOI: 10.1109/ICSE43902.2021.00106 (2021).

[4] Huang, Z., Wu, S., Jiang, S. and Jin, H.: FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container, *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 28–37 (online), DOI: 10.1109/MSST.2019.00-18 (2019).

[5] Lin, C., Nadi, S. and Khazaei, H.: A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub, *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 371–381 (online), DOI: 10.1109/ICSME46990.2020.00043 (2020).

[6] Lane-Walsh, S., Stillerman, J., Santoro, F. and Fredian, T.: Introduction to MDSplus using Docker, *Fusion Engineering and Design*, Vol. 165, p. 112121 (2021).

[7] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Slacker: Fast distribution with lazy docker containers, *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 181–195 (2016).

[8] Civolani, L., Pierre, G. and Bellavista, P.: FogDocker: Start container now, fetch image later, *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 51–59 (2019).

[9] Kangjin, W., Yong, Y., Ying, L., Hanmei, L. and Lin, M.: FID: A Faster Image Distribution System for Docker Platform, *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 191–198 (online), DOI: 10.1109/FAS-W.2017.147 (2017).

[10] Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L. and Teal, T. K.: Good enough practices in scientific computing, *PLoS computational biology*, Vol. 13, No. 6, p. e1005510 (2017).

