修士学位論文

題目

版管理リポジトリ上の設定ファイルに対する命名の現状調査とガイドラインの提案

指導教員

楠本 真二

報告者

忠谷 晃佑

令和 8 年 2 月 2 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 7 年度 修士学位論文

版管理リポジトリ上の設定ファイルに対する命名の現状調査とガイドラインの提案

忠谷 晃佑

内容梗概

近年の大規模チームによって遂行されるソフトウェア開発において，版管理リポジトリは中心的な役割を果たす．また近年では様々なプロセスの自動化もツールによって行われており，プロジェクトの規模が大きくなるにつれて多数のツールが導入されると，版管理リポジトリにはそれぞれのツールに対する設定ファイルが配置される．ツールを多く導入すると，開発プロセスにおいて自動化による恩恵を受けられる一方で，それぞれのツールが必要とする設定ファイルもその数が増加し，版管理リポジトリ上のファイルそれぞれの意図やプロジェクト構造が把握しにくくなる．この原因として，単純な設定ファイル数の増加だけはなく，設定ファイルに対する命名規則が様々であり，単一リポジトリ内でも統一されていないという現状が挙げられる．そこで本研究では設定ファイルの命名に対して共通の認識をガイドライン形式で与える．そのために OSS リポジトリ上の設定ファイル名に対する現状調査を実施する．調査においては 140 個のソフトウェア開発プロジェクトのリポジトリから 618 個の設定ファイル名を収集しそれらを分析する．調査の結果，命名における 2 つのプラクティスを確認した．また，調査の結果に基づきプラクティスがリポジトリの構造理解に与える影響を考察し，ガイドラインを提案した．

主な用語

設定ファイル，命名規則，ガイドライン，リポジトリマイニング

目次

図目次

表目次

1　はじめに

大規模チームによるソフトウェア開発において，版管理リポジトリは中心的役割を担っている．そこでは開発において得られたあらゆる成果物を管理するだけでなく，開発者の貢献度を推定できたり，プロジェクトの進捗を管理したりもできる [1]．近年のソフトウェアリポジトリには開発対象プロダクトであるソースコードだけではなく，それに対するテストコードやビルドプロセスを自動化するためのビルドスクリプト，またその他自動化ツールの設定ファイルやエンドユーザ向けのドキュメントが管理される [2]．開発に係るあらゆるファイルを中央的に保管し，プロジェクトのさまざまなステークホルダーら（例えば開発者やメンテナ，ユーザーなど）に一元的なリポジトリ構造の理解をもたらす．

近年では開発プロセス自動化のために様々なツールが導入されており，例えばファジングツールを用いたテストの自動化 [3] や Docker などのコンテナプラットフォームを用いた環境構築の自動再現 [4, 5]，静的解析ツールによるソースコードの問題点の捜索 [6, 7] のほかにもパッケージマネージャーを用いた依存関係の管理 [8] がある．これらのツールを導入しその設定を共有するとチームの貢献者らが同一の開発環境を用いたり，同じコーディング規約を採用したりできる [9, 10, 11]．

ツールの導入による開発の効率化が可能 [12] な一方で，それぞれのツールが必要とする設定ファイルが版管理リポジトリ上で問題になるとも考えられる．ツールが必要とする独自の設定ファイルがリポジトリ上に増加すると，リポジトリ全体の構造が複雑になるためである．例えば継続的インテグレーション（CI）環境を提供するシステムにはワークフローの記述が必須である．他にも Docker は Dockerfile でコンテナのビルド手順を定義するし，パッケージマネージャーの一つである npm も package.json というファイルにその依存関係を記述する．これらのファイルはほとんどがリポジトリのルートディレクトリに配置されており，事前調査で GitHub の OSS 開発リポジトリのルートディレクトリには平均で 34.3 個のファイルが存在すると判明した．数の増加に加えて，設定ファイルには命名規則が存在しないという点もそのファイルが果たす役割の把握を難しくしている．設定ファイルが果たす役割上，そのファイルがどのツールに対応する設定を記述しているかに加えて，どのような書式で内容を記述しているかが名前から把握可能であるべきと考えられる．しかし現状，すべての設定ファイル名がこれらを満たすとはいえないうえ，設定ファイル名はツール制作者によって事前定義されている場合が多いためリポジトリ管理者が独自に改善するのは困難である．

そこで本研究では版管理リポジトリで管理するべき設定ファイルの命名に対して一定の共通認識の形成を試みる．その形成手段として，設定ファイルの命名に対してガイドラインを提案する．

ガイドラインの提案においては，現状の設定ファイルに対する命名を調査する．ガイドラインをより受け入れられやすいものにするには，現状の多数派の把握が必要なためである．調査に当たっては実際の版管理リポジトリ上の設定ファイル名を調査する，マイニング調査を実施した．GitHub 上のソフト

ウェア開発プロジェクトのリポジトリ 140 個から設定ファイル名 618 個を収集し，設定ファイル名に対してそれらが持つ意味的な観点で命名プラクティスを調査した．

結果として，設定ファイル名の多くはツールの開発者によって定められているという事実と，設定ファイル命名における 2 つのプラクティスを発見した．また，命名のプラクティスについてそれらがリポジトリの構造理解にもたらす恩恵を考察し，これに基づいてガイドラインを提案した．

2　版管理リポジトリにおける設定ファイル命名の現状

本節では実際の版管理リポジトリを題材として設定ファイル名がプロジェクト構造の理解に及ぼす影響を述べる．例として，図 1 に広く用いられている Python パッケージの一つである NumPy の

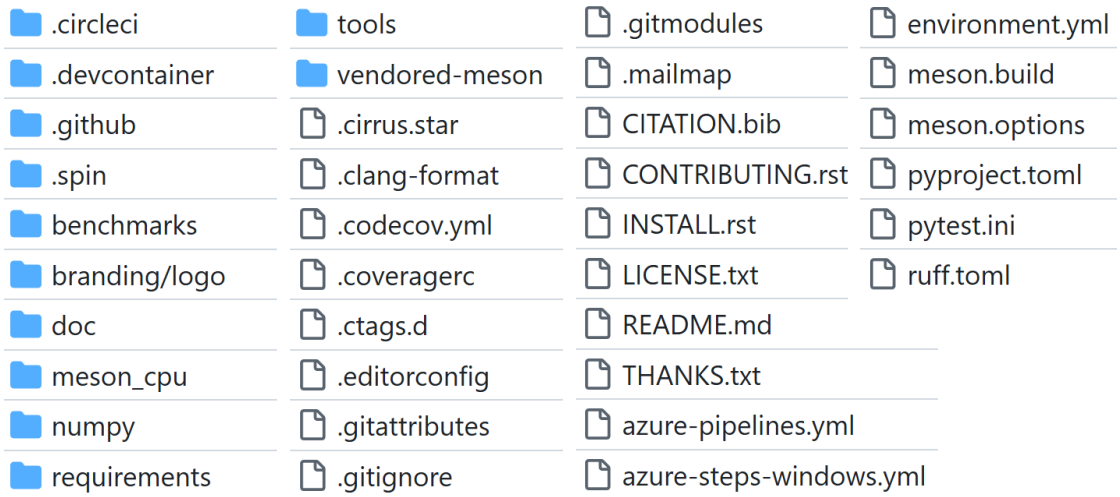

図 1: NumPy リポジトリ（numpy/numpy）のルートディレクトリ

GitHub リポジトリ[*1]を示す．NumPy は C 言語と Python を用いて実装されており，ルートには 37 個のファイルが存在している．後述するデータセットを用いた事前調査の結果，GitHub 上の OSS 開発リポジトリには平均で 34.3 個のファイルが存在すると判明している．そのため，例示したプロジェクトのルートに存在するファイル数は極端に大きな数ではないといえる．

Numpy プロジェクトの貢献者らは様々な目的を達成するために，リポジトリ上のファイルを閲覧または編集する．このとき対象とするファイルは，貢献者らの目的によってそれぞれ異なる．以下にプロジェクトに対して様々な目的をもった貢献者がアクセスするファイルを探すシナリオ例とともに，設定ファイル名がプロジェクト理解にもたらす影響を述べる．

2.1　シナリオ例：CI のメンテナンス

NumPy プロジェクトで用いられている CI 環境のメンテナンスをする場合について考える．メンテナはまず初めに CI 環境プラットフォームに関連するファイルを洗い出す．NumPy では GitHub Actions や CircleCI を利用しており，それぞれ `.github/` と `.circleci/` ディレクトリ内にワークフローを定義したファイルが配置されている．このプロジェクトで他に用いられている CI プラットフォー

[*1] `https://github.com/numpy/numpy/tree/2b2dac4` Accessed at 2026/2/2

ムとして Cirrus CI と Azure Pipelines が存在する．これらのワークフローはそれぞれ `.cirrus.star` と `azure-pipelines.yml` で定義されている．CI に関連した設定ファイルに絞って注目しても，サブディレクトリを採用したプラットフォームとそうでないプラットフォームで一貫性がないとわかる．さらにはドットファイルか非ドットファイルかという視点でも一貫性がない．

2.2 シナリオ例：コーディング規約を更新する

次に，プロジェクトで採用するコーディング規約を更新する場合を考える．この場合，貢献者はリンタやフォーマッタの設定を更新する必要がある．NumPy プロジェクトでは，自動化ツールを用いてコーディング規約に違反した記述を開発者に通知しており，変更後の規約にツールの設定を適合する必要があるためである．

関連するファイルとしてまず `.editorconfig` が挙げられる．これは IDE やテキストエディタに対して横断的にコーディング規約を定義するツールである EditorConfig の設定が INI 形式で記述されたファイルである．他にも C ソースファイルのフォーマッタである ClangFormat に対して，C ソースコード記述規約を YAML 形式で記した `.clang-format` がある．これら 2 つのファイルには拡張子が付加されておらず，ファイル名からその書式を特定できない．一方で Python ソースファイルのリンタである Ruff に対してその設定を TOML 形式で与える `ruff.toml` にはその書式に対応する拡張子が付加されている．これも設定ファイル名に存在する一つの一貫性がない点である．

2.3 シナリオ例：依存関係を整理する

プロジェクトが依存している外部ライブラリの整理をする場合に必要なファイルをリポジトリ内から洗い出す．Python を用いたプロジェクトでは Python Enhancement Proposal（PEP）518 および 621[13, 14] で定められたファイルである `pyproject.toml` に記述されている．ところが NumPy プロジェクトは Conda を用いた仮想環境を利用しており，この環境における依存関係は `enviroment.yml` に記述されている．このファイル名から，Conda が提供する仮想環境に関する設定ファイルである，と類推するのは難しく，開発者によっては役割の把握が困難なファイルであるといえる．他にもこのプロジェクトでは依存関係を `requirements/` 以下のファイルに設置された `*-requirements.txt` にも記述している．これは Python の標準的なパッケージマネージャである pip で慣習的に用いられるファイル名ではある．このファイル名から pip を導くのは難しいうえに，`*-requirements.txt` はユーザに定義されたファイル名であって pip により厳密に定められたファイル名ではない．以上より依存関係に関連したファイルはファイル名から依存関係が記述されているとの推測が難しい場合が多いといえる．さらに，ツールが厳密にファイル名を事前定義しているか否かにも一貫性はない．

3　本研究で取り組む課題と提案

3.1　設定ファイル名の非一貫性

本研究で取り組む課題は，リポジトリ上の設定ファイルの増加とそれらの名前にある非一貫性により版管理リポジトリ上の構造理解が妨げられているという現状である．2 節に示したシナリオ例よりリポジトリ上に存在する様々な設定ファイル名は以下の点で一貫していない

- 設定ファイルは一つのファイルかサブディレクトリ以下に存在するか
- 設定ファイルはドットファイルであるか
- ファイル名から対応するツールを特定できるか
  - ファイル名にツール名が含まれているか
  - ファイルの持つ役割はファイル名から特定できるか
- ファイルの書式は拡張子と一致するか

さらに，設定ファイル名が必ずしもツールの利用者によって自由に決定できるとは限らない点も設定ファイル名管理の難しさにつながっている．2 節で取り上げたファイル中でもプロジェクト側でファイル名を設定できるのは `azure-pipelines.yml` と `requirements.txt` のみである．

ファイル名はプロジェクトの構造を理解するために用いられる最も基本的で簡素な情報である．その為，命名規則の不一致は開発の様々な場面でリポジトリ構造の理解を妨げる．

3.2　設定ファイルの命名に対する提案

以上の課題に対して，本研究では設定ファイルの命名に対して一定の共通認識の形成を考える．その手段として，設定ファイルの命名に対するガイドラインを提案する．厳格な規則としてではなくガイドラインとしての提案とする理由は，版管理リポジトリを用いるプロジェクトにはそれぞれ異なる様々な命名規則が採用されているためである．3.1 節で述べたように，リポジトリ上の設定ファイル名の間にある非一貫性が版管理リポジトリの構造理解を妨げていると考えられる．そのためプロジェクトごとに異なる命名規則に対して柔軟に対応できる提案がより広く受け入れられやすいといえる．

提案内容の考案にあたり，現状の版管理リポジトリにおける設定ファイルの命名規則の把握が必要である．ガイドラインをより受け入れられやすいものにするためには，現状と大きくかけ離れた提案を避ける必要がある．また，現状の問題点や一貫性がない点を洗い出し，その解決を図るためにも現状の把握は必須である．

3.3 現状把握を目的とした Research Questions

本研究では設定ファイルの命名に対するガイドライン提案をするにあたって，現状の設定ファイルに対する命名プラクティスを調査する．この調査の目的は，実際の版管理リポジトリのマイニング調査によって設定ファイルに対する命名の現状を把握する点にある．調査に当たっては 2 つの ResearchQuestion を設定する．

RQ1: 設定ファイル名はどれほど事前定義されているか

RQ1 では設定ファイル名そのものの詳しい観察を実施する前段階として，その名前は誰によって決められたかを調査する．3.1 節で述べた通り，設定ファイルの名前は必ずしもツールのエンドユーザが自由に設定できるとは限らない．ツール開発者によってあらかじめ設定ファイルの名前が定められているツールも数多く存在する．そのためガイドラインを提案する対象を把握するためにも命名者を特定する必要がある．

RQ2: 設定ファイルの命名には現状どのようなプラクティスが存在するか

RQ2 では設定ファイル名を調査し，現状どのようなプラクティスが存在するかを調査する．3.2 節で述べた通り，設定ファイルの命名に対してガイドラインを提案するにあたりより受け入れられやすい内容を目指すには現状の把握が必要である．実際にソフトウェア開発者らに対して行動の指針を示唆する研究においては現状の開発者らの行動などに基づいた提案をするために現状を調査する場合がある [15, 16]．本研究も同様に現状の設定ファイルに対する命名プラクティスを調査する．RQ2 に対する回答として得られた現状を基に，それらが行われている理由やそれによって受けられるリポジトリ構造理解の上での恩恵を考察する．

また，本 RQ への回答を通して現状の設定ファイルに対する命名においてファイルそれぞれで一貫性のない点を洗い出す．これを基に，ガイドラインにて現状一貫していない命名プラクティスに対して一定の示唆を試みる．

4 調査

4.1 調査対象の収集

4.1.1 対象リポジトリの選定

設定ファイルの調査においては OSS リポジトリのマイニング調査を実施する．調査対象リポジトリの選定においてはそれぞれ異なるプログラミング言語を開発に用いるリポジトリを収集する．各言語が形成するエコシステムの様々なツールに関連するファイルを取得するためである．本調査では以下の言語を主として開発を行うプロジェクトのリポジトリを調査の対象とする．選定においては GitHub プラットフォームがサポートを提供している言語を選択した．GitHub の利用者が比較的，採用しやすい言語であると考えられるためである．

- C
- C++
- C#
- Go
- Java
- JavaScript
- Kotlin
- PHP
- Python
- Scala
- Swift
- TypeScript
- Ruby
- Rust

以上の 14 言語を主とするリポジトリをスターの多い順に 10 個ずつ取得し計 140 個のリポジトリをデータセットとして用いる．ただし，その中に教育や技術の習得を目的としたリポジトリやサンプルコードの集積リポジトリが存在した場合は対象とせず，次にスターが多いリポジトリを対象としている．本調査ではソフトウェア開発リポジトリにおいてのプラクティスに集中しているためである．スターは GitHub 上のリポジトリの人気度を示す一つの指標であると知られている [17]．人気があり注目されているリポジトリは開発環境が整備されており様々な設定ファイルの取得が期待される．スターを選定のメトリクスとして用いたのはこのためである．

4.1.2 設定ファイル名の収集

次に，4.1.1 節で収集したリポジトリのルートディレクトリから設定ファイルを収集する．ここで，本稿における設定ファイルの定義を述べる．

> リポジトリで管理されているテキストファイルのうちツールの動作をビルドなしで定義もしくは変更できるもの．もしくはそのようなファイルが配下に存在するディレクトリ．

この定義は Siegmund らが提案した**ソフトウェア設定の次元** (*Dimentions of Software configuration*)[18] に基づいている．Siegmund らの提案した次元のうち，設定が現れる成果物の次元に着目し，

リポジトリで管理されうるファイルに注目した．

ファイル名の収集においては対象リポジトリのルートディレクトリに存在するファイルを全て収集し，それらから定義に基づく設定ファイルを収集する．まず，拡張子に基づき定義に明らかに一致しないファイルを除外する．例えばコンパイルが必要なファイル（ `*.c` , `*.rs` ）やプロジェクトの様々なステークホルダーに提供されたドキュメント（ `*.md` ），またバイナリファイルがある．残りのファイルに関してはファイル名やコミットメッセージに基づいて設定ファイルであるかを目視で判断する．最終的に 1,929 個の設定ファイルを収集し，ユニークな 618 個のファイル名を取得した．最終的に取得したデータセットについて 表 1 に示す．

表 1: GitHub から取得したデータセット

収集対象とするプログラミング言語の数	14
各言語で収集したリポジトリの数	10
合計リポジトリ数	140
収集した設定ファイルの数	1,929
収集した設定ファイル名の数	618

4.2 設定ファイル名の調査

収集した設定ファイル名を分析する．分析においてはいくつかの異なる視点に着目する．それぞれの調査項目について，調査の意図と調査方法を以下の節で述べる．

4.2.1 ファイル名は事前定義されているか

はじめに，それらの名前が事前定義された名前であるかに着目する．本調査における事前定義された名前は，対応する各ツールのドキュメントで言及された名前とする．

自動化ツールは自身の設定ファイルをあらかじめ定義された名前，つまり開発者による事前定義名によって探索する場合が多い．すると，設定ファイル名の各プロジェクトや組織が用いる命名規則への適合が難しくなる．これはリポジトリ上のファイルに対する命名から一貫性が失われる原因の一つと考えられる．本調査において，どれほどの設定ファイルの名前の決定権が，ツールの開発者側またはツールのエンドユーザ側にあるかを調査する．

4.2.2 ドットファイルであるか

ファイル名がそれぞれドットファイルであるかを調査する．本調査におけるドットファイルは名前の先頭がドット（.）であるようなファイルまたはディレクトリであるとする．

ドットファイルであるか否かに注目する理由は，ドットファイルが隠しファイルであるという共通認識の存在にある．UNIX 系 OS において `ls` コマンドを用いてファイルを一覧表示すると，ドットファイルはデフォルトでは表示されない．この特性を用いて，普段ユーザが直接読み書きする機会は少ないがアプリケーションなどによって読み込まれる場合が多いファイルがドットファイルとして管理され，`ls` コマンドの出力を整理したり，ユーザによる不用意な編集を防止したりしている．

以上のような特性から UNIX 系 OS においてソフトウェアの設定はドットファイルである場合があり，この事実も共通認識として存在する [19]．したがって，ファイル名の先頭に存在するドットは他の文字より多くの意図を示唆する場合が多い．設定ファイルがドットファイルであるかを調査し，版管理リポジトリ上でのドットファイルがどのように扱われているかプラクティスを把握する．

4.2.3　ファイル名に含まれる語はどのような意味を持つか

ファイル名に含まれる語がどのような意味を持つか，それぞれについて調査を行う．Anquetil と Lethbridge はソースコードのファイル名に注目し，それらがいくつかの意味ある語に分割できると主張した [20]．また，それらの語がソースコードが実現する機能やそれらが参照するデータを表しているとも主張した．設定ファイル名についても，そのファイル名が関連する情報を表しているべきであると考える．本研究では以下の要素に注目して考える．

ツール名　対応するツール名が設定ファイル名に含まれているべきである．

書式　設定ファイルは構造化されたテキストファイルである場合が多い．そのため書式が判断できるファイル名が望ましい．

調査においては，辞書に基づいてファイル名を意味的な部分に分割しそれぞれの語を抽出する．この辞書には一般的な英単語に加えて，ソフトウェア開発においてよく用いられる略語[*2]やツール名，またそれぞれのプロジェクト名を用いる．それぞれの単語がツール名，もしくは書式に一致するかを目視によって調査する．

4.2.4　ファイルの拡張子と書式は一致するか

次に設定ファイルの拡張子に着目する．本調査における拡張子は，ファイル名の先頭にあるものを除いたドットでファイル名を分割し得られる部分文字列のうち，最後の一つを表す（図 2）．

拡張子は多くの場合そのファイルの書式を表現する．OS やアプリケーションによるファイルの判別も同じく拡張子によって行われる場合が多い．そのため設定ファイルについても拡張子を適切に付加してシンタックスハイライトや補完機能などの恩恵を受けられるようにすべきである．

[*2] `https://github.com/abbrcode/abbreviations-in-code/tree/4dddb19` Accessed at 2026/2/2

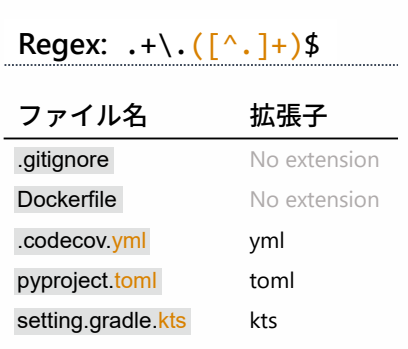

図 2: 拡張子の例：正規表現にマッチするファイル名のうちオレンジで示したグループにマッチする文字列を拡張子とする

4.2.5 ファイル名における要素のセパレータは何か

命名においては構文的な取り決めも必要である．ファイル名に含まれる要素について 4.2.3 節では意味的な側面に注目した．しかし，それらの要素をどのように分割するかについてもガイドラインの提案には必要である．ソフトウェア開発において発生する命名は多くの場合，構文的な規則が定められている [21]．広く知られている規則としては，各要素の先頭をキャピタライズして分割する CamelCase や，要素間をアンダースコア（_）で分割する snake_case が知られている．設定ファイルの命名に一定の共通認識を形成するという目的を達成するうえで，ソースコード上の識別子に対してされてきたような取り決めが必要である．

調査においては 4.2.3 節で得られたファイル名の意味的分割を用いる．この分割がファイル名の上ではどのように表現されているかを調査する．ただし，この調査においては 図 2 に示す拡張子を取り除いたファイル名を調査する．拡張子はファイル名に対してドット区切りで付加するという共通認識はすでに広く知られているうえに，OS や多くのアプリケーションの実装としても存在するためである．

5　調査結果

5.1　ファイル名は事前定義されているか

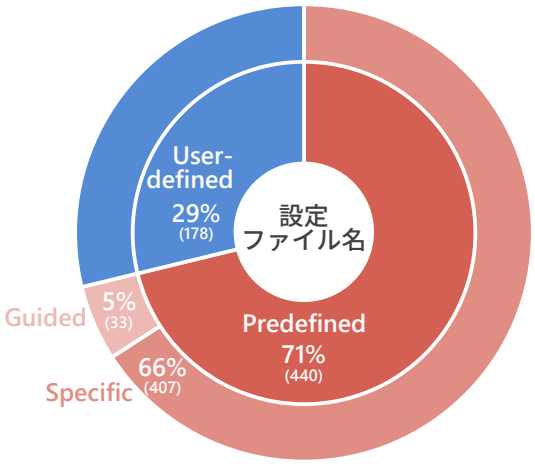

図 3: 事前定義されている設定ファイル名とユーザ独自の設定ファイル名それぞれの割合

図 3 に設定ファイル名が事前定義されたものであるかを調査した結果を示す．調査の結果，全体の 71% の設定ファイル名が事前定義された名前であると判明した．また，全体の 66% は特定の名前（Specific）である．つまりツール開発者がファイル名やディレクトリ構造をすべて定義しておりエンドユーザが全くコントロールできないファイル名である．一方で全体の 5% はファイル名が完全に指定されているわけではないものの，そのコントロールが難しいファイル名（Guided）である．例えば Visual Studio のプロジェクト管理に用いられるソリューションファイル（`*.sln`）はツール開発者によって決まりきった名前が存在するわけではない．しかしプロジェクト名である必要があり，その他の名前にはできない．これらのファイルについてもエンドユーザに決定権があるとはいえない．

71% のファイル名について，それらがエンドユーザではなくツールの開発者によって決定されているとわかった．この事実は，OSS プロジェクトがそれぞれ異なる命名規則やディレクトリ構造を採用しているという現状に反しており，プロジェクト側の工夫のみではリポジトリの構造理解を促進できない．したがって，設定ファイル名に対するガイドラインはツール開発者に対しても受け入れられやすくなくてはならない．

_RQ1_ の回答

71% の設定ファイル名は事前定義されており，ツールのエンドユーザが名前を自由に決定できる設定ファイルは少ない．

5.2　ドットファイルであるか

ファイル名に対する調査についてそれぞれの観点に基づいて調査した結果，確認されたそれぞれのファイル名の個数とその割合を 図 4 に示す．それぞれのファイルがドットファイルであるかを 図 4 の

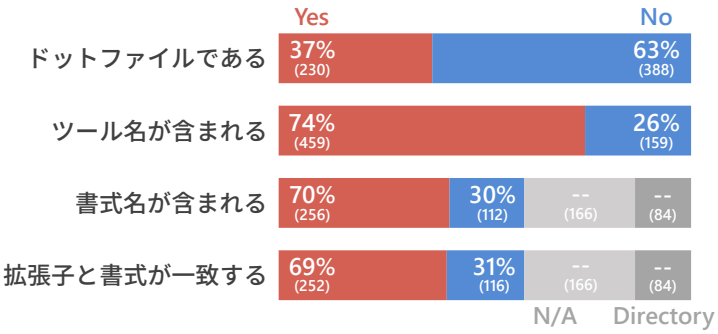

図 4: それぞれ異なる観点でのファイル名の分析結果

一番上のグラフに示す．調査の結果，設定ファイル名全体のうちドットファイルであるような名前は 37% であることがわかった．これよりさまざまなエコシステムを横断的に考えたとき，ツールの設定ファイル名は UNIX 系 OS 上で用いられる文化とは異なる命名をされているとわかる．

5.3　ファイル名に含まれる語はどのような意味を持つか

図 4 の上から 2 番目と 3 番目のグラフにファイル名を意味的な語に分割して分析した結果を示す．上から 2 番目のグラフは設定ファイル名にそのファイルが対応するツールの名前が含まれているかを示す．設定ファイル名全体のうち 74% のファイル名に対応するツール名が含まれているとわかった．

一方で上から 3 番目のグラフはファイル名にそのファイルの書式名が含まれているかを調査した結果を示す．この調査にあたっては，一部のファイル名を調査対象から除外している．まず，設定ディレクトリの名前を除外する．ディレクトリ自体が書式を持たないためである．次に，よく知られた書式が存在しないファイルの名前を除外する．設定ファイルが用いる書式にはよく知られた名前が存在しないものがある．それらの書式を分類し 表 2 に示す．調査では以上のいずれかに当てはまるファイル名をそれぞれ 図 4 の “Directory” と “N/A” として除外し記述している．結果，対象のファイル名のうち 70% のファイル名にそのファイルの書式が挿入されているとわかった．

5.4　ファイルの拡張子と書式は一致するか

図 4 の一番下のグラフにそれぞれのファイル名に付加されたファイル名の書式がファイルの記述書式と一致している数とその割合を示す．またファイル名に対応するファイルの書式ごとに拡張子が一致

表 2: 設定ファイルにみられる書式の分類

分類名	説明	#
List	一行に一つの値もしくは Key-Value を記述したテキストファイル	80
DSL	当該設定ファイルの記述のみに用いられる言語で記述されたファイル	50
Unique	当該設定ファイルに独自の構造を持ったテキストファイル	16
Other	その他のファイル	20
合計		166

表 3: 書式ごとの拡張子一致割合とその数

書式	拡張子	拡張子と書式が一致	拡張子と書式が不一致	合計
JSON	`*.json` , `*.jsonc`	73 (78%)	20 (22%)	93
YAML	`*.yaml` , `*.yml`	80 (88%)	11 (12%)	91
JavaScript	`*.js` , `*.mjs`	46 (100%)	0 (0%)	46
XML	`*.xml`	9 (25%)	27 (75%)	36
TOML	`*.toml`	20 (91%)	2 (9%)	22
INI	`*.ini`	6 (35%)	11 (65%)	17
その他		22 (35%)	41 (65%)	63

している割合を 表 3 に示す.

よく知られた拡張子が存在する書式を用いる設定ファイルのうち 69%（252/368）のファイル名に拡張子が付加されていた．なお，拡張子がと書式が不一致であるとした 31% のファイル名には全て拡張子が付加されていなかった．つまり，例えば JSON 形式のファイルであるにも関わらず `.yml` のような他の書式の拡張子として知られるものが付加されているファイルは存在しなかった.

5.3 節に示したファイル名に書式が含まれる割合 70% に対して，ファイル名を拡張子として持つファイルは 69% と少なくなっている．これは書式名を末尾ではない部分に含んでいるファイル名が存在したためである．実際に確認した例として `phpunit.xml.dist` がある．これは PHP 言語に対するテストランナーの一つである PHPUnit の動作を設定するファイルの事前定義名である．`*.dist` は一般的に，“distribution” を意味する語であり，ここでは公開リポジトリにプッシュして配布するファイルであるという意図をファイル名に付加している．しかしこのファイル名では拡張子が XML ではないため，書式に対応したツールの支援を受けられない可能性がある．`phpunit.xml` に “distribution” の意図を含ませる場合は，同じく事前定義名である `phpunit.dist.xml` がより適切であると考えられる.

5.5 ファイル名における要素のセパレータは何か

表 4: 要素の分割方法とその数

分割方法名	概要	#ファイル名
flatcase	要素を記号等用いずに接続する名前	132
dot.notation	要素間をドット（.）で接続する名前	91
chain-case	要素間をハイフン（-）で接続する名前	60
snake_case	要素間をアンダースコア（_）で接続する名前	24
CamelCase	要素の初めの文字をキャピタライズして接続する名前	9
複合型	要素の接続に以上 5 つのうち 2 つ以上の方法を用いている名前	32
なし	拡張子を除き一つの要素で構成された名前	270

現状のファイル名における要素の分割方法を 表 4 に示す．現状もっとも使われている書式は要素をセパレータやキャピタライズなしに連結する flatcase である．この書式は設定ファイルによく用いられる `.<ツール名>rc` や `.<ツール名>ignore` という名前が採用している．したがって，伝統的な理由でこの命名を採用している場合が多いともいえる．

デリミタを用いたファイル名の書式としてよく用いられるのは dot.anotation と chain-case である．これらはソースコードやテストコードの名前として扱われにくい．ドットやハイフンはソースコード中で演算子として扱われる場合が多いためである．この 2 種類の書式について比較する．dot.notation はデフォルトや広く用いられている名前に対して利用環境や目的を限定するために情報を付け加えるときに用いられる場合が多い．chain-case も同じく一般的な名前に情報を付加するために利用されるほか，一般的な文章での利用方法と同じく単語と単語の合成にも利用される．一方で snake_case や CamelCase は，ソースコード中でも用いられる形式であるが設定ファイルの命名規則としては一般的ではないと分かる．

RQ2 **の回答**

現状の設定ファイル名が採用しているプラクティスは 2 つ存在する．対応するツール名の挿入と，記述内容の書式に一致する拡張子の付加である．

6　考察

6.1　各特徴の組み合わせの分析

本節では 5 節で報告したそれぞれのファイル名の特徴について，他の特徴との関係に注目してファイル名を整理する．5 節では設定ファイル名に対していくつかの観点で独立した調査を実施した結果を示した．そこで，ファイル名を一定の基準によって細分化しそれぞれのプラクティスを把握する．

6.1.1　ドットファイルと非ドットファイルの比較

設定ファイルそれぞれが，ドットファイルであるか否かに注目しそれぞれの命名に対する特徴を分析する．ドットファイルであることは他の特徴に比べて，より示唆に富む命名である．したがってドットファイルとして設定ファイルを命名する場合，一定の意図が存在すると考えらえる．しかし 5.2 節で述べた通り，版管理リポジトリにおいては設定ファイルをドットファイルとするプラクティスが認識されているとは言えない．

ドットファイルと非ドットファイルである設定ファイルのあいだに異なる共通認識がある場合，ガイドラインの提案にあたりそれらを考慮する必要がある．そこで，ファイル名をドットファイルと非ドットファイルという 2 つに細分化し，それぞれについて注目した分析をする．その結果を 図 5 から 図 7 にそれぞれ示す．

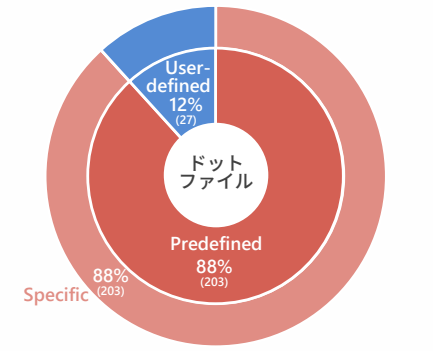

図 5: ドットファイルが事前定義されている割合

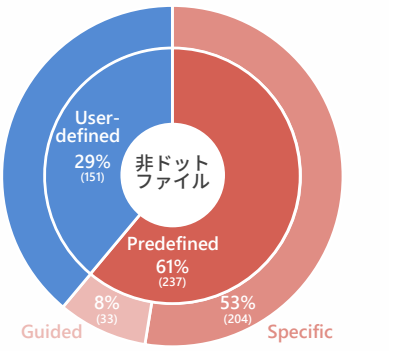

図 6: 非ドットファイルが事前定義されている割合

図 5 と 図 6 は 5.1 節に対応する分析を示す．図 5 に示すように，ドットファイルであるような設定ファイルの名前は 88% と大部分がツール制作者によって事前定義されたファイル名であり，ユーザが定めたファイル名は全体の 12% にとどまる．一方で 図 6 に示すように，非ドットファイルである設定ファイル名はユーザが定めたファイル名が 29% にも上る．

非ドットファイル中にユーザ定義名のファイルが増えた原因として，同様の領域に対して設定を行うファイルが一つのプロジェクトに複数存在し，それらが非ドットファイルである場合が多いと

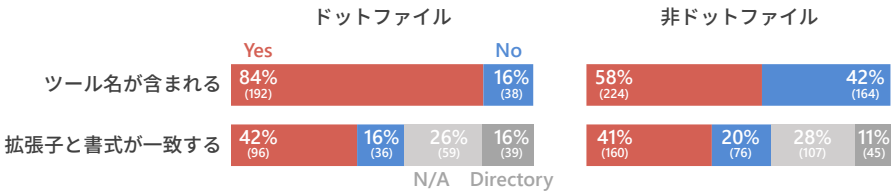

図 7: 図 4 をドットファイルと非ドットファイルそれぞれで分析した結果

いう点が挙げられる．ひとつの例として Python のパッケージ管理に用いる `requirements.txt` がある．TensorFlow[*3] の開発プロジェクトでは，開発者らが開発に用いる環境にインストールされている Python バージョンに対応するためにそれぞれ異なる `requirements_lock_3_*.txt` を用いている．いずれのファイルも Python パッケージを管理するために利用するファイルであるが，それぞれ異なる開発環境で用いるファイルである．一方でドットファイルであるような設定ファイルには同じツールの同じ機能に対して複数の設定ファイルが存在するというケースは少なかった．これはドットファイルが非ドットファイルに対して，一定の設定を提供するファイルの名前に使われる場合が多いためであると考えられる．例えば版管理の設定（`.gitignore`）やフォーマッタの設定（`.clang-format`）はプロジェクトで一様に設定されるべき事項であり，異なる環境に対してそれぞれ異なる設定を提供する場面は少ないといえる．ドットファイルは伝統的に UNIX 系 OS における設定ファイルであるといえるが，これらはユーザ個人用の設定するファイルが多く [19]，この認識が版管理リポジトリ上の設定ファイルにも存在すると考えられる．

次にそれぞれのファイル名にツール名が挿入されている割合をドットファイルと非ドットファイルで比較する．すると，ドットファイルにはツール名が挿入されているファイル名が全体の 84% であったのに対して，非ドットファイルでは 58% であった．非ドットファイル名にツール名の挿入が少ない理由も，非ドットファイルが環境に対する設定である場合が多いためであるといえる．そのため，ツールよりもどのような環境で利用されるファイルかに注目した命名をする場合が多く，ツール名の挿入が少なくなったと考えられる．

最後に拡張子と書式が一致しているファイル名の割合を比較する．図 7 の下段に図示したグラフにそれぞれのファイル名全体に対して，拡張子が一致したファイル名と不一致のファイル名に加えて，一般的に知られた拡張子が存在しないファイルとディレクトリの割合も図示している．これらを比較すると，すべての分類についてドットファイルと非ドットファイルいずれも割合がほとんど一致する．したがって，拡張子や書式に対する開発者らのファイル名に対する傾向はドットファイルと非ドットファイルで同一であるとわかる．したがって，ドットファイルであるか否かに関わらず書式に対応する拡張子を付加するべきである．

[*3] `https://github.com/tensorflow/tensorflow/tree/5563e93` Accessed at 2025/1/10

6.1.2　事前定義された名前とユーザ定義名の比較

次に設定ファイル名それぞれが，ツール開発者によって事前定義された名前かユーザによる命名かに注目して設定ファイル名を分析する．命名者の立場がツールの開発者とツールのエンドユーザのようにそれぞれ異なるため，ファイル名の意図についても異なると予想される．そのため，命名者のコミュニティに基づいてファイル名を細分化し，それぞれに存在する命名に対しての共通認識を洗い出す．

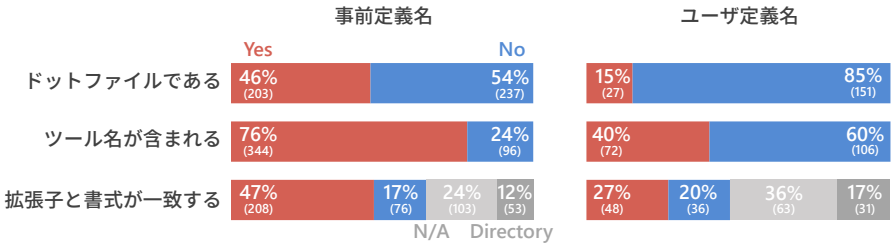

図 8:　図 4 を事前定義名とユーザ定義名それぞれで分析した結果

分析結果を 図 8 に示す．はじめにドットファイルの割合を調査する．事前定義名のうちドットファイルであるようなファイル名は全体の 46% であったが，ユーザ定義の名前でドットファイルであるようなファイル名は全体の 15% であった．6.1.1 節で述べた通りドットファイルの多くは事前定義名であるといえる．

ツール名の挿入割合についても，事前定義されたファイル名に多く見られ，全体の 76% に，一方でユーザ定義名には 40% のファイル名に見られた．これは，ツール開発者らが自らのツールが探索するファイル名を，他のシステムが利用する設定ファイルと衝突しない一意な名前にしようとした結果であると考えられる．設定ファイルに対して，例えば `config.json` など汎用的に用いられる言葉のみを用いると他ツールとの衝突が十分に考えられる．ファイル名に挿入されたツール名は一種の ID の役割を果たせる．

最後に拡張子と書式に注目する．書式と拡張子が一致するファイル名に注目すると，事前定義名では全体の 47% であったがユーザ定義名は 27% に留まった．事前定義名に拡張子を付加した名前が多い理由として，開発者らの間で広く知られた拡張子が存在するのであればそれを明示的に付加する文化が形成されているためであると考えられる．一方で広く知られた拡張子が存在しない書式を採用したファイルやディレクトリの名前はユーザ定義の場合が多く，ユーザ定義名のそれぞれ 36%，そして 17% であった．広く知られた拡張子が存在しない書式を用いるファイル名がユーザ定義名を付けられる場合が多い理由は，環境ごとに異なる設定を利用するファイルがドメイン固有言語を用いている場合が多いためである．6.1.1 節で述べたように，ユーザ定義名のファイルは同じ領域に対して環境に合わせた設定をするために用意されている場合が多い．ユーザが独自に名前やパスを指定でき，固有言語を利用し

ている設定ファイルとしてよく知られるものに，Docker コンテナイメージのビルド手法を固有のコマンドを用いて宣言的に定義する Dockerfile やビルドやテストランナーのひとつである Bazel の動作を Starlark で定義する *.bzl ファイルがある．このようなツールの設定においては，より表現の幅が広い固有言語を用いて設定ファイルを記述している場合がある．

6.2 現状存在するプラクティス

本節では 5 節および 6.1.1 節で述べた結果を基に現状の OSS リポジトリ上に存在する設定ファイル命名のプラクティスを整理し，それによって版管理リポジトリの構造を理解するうえで受けられる恩恵を考察する．より受け入れられやすいガイドラインの提案には現状の考慮が必要なためである．

6.2.1 ツール名の挿入

設定ファイル名には対応するツールの名前を挿入するプラクティスが存在する．5.3 節にて述べた通り，設定ファイル名の 74% には対応するツールの名前が挿入されている．これは，すでに共通の認識として形成されたプラクティスともいえる．

ツール名を設定ファイル名に付加すると得られる恩恵は複数存在する．まず，ツール名が ID としての役割を担える点である．ツール名が名前に挿入されていれば，各ファイルの利用目的が明確になるうえ，他のツールが用いる設定ファイルとの衝突や混在も回避できる．

また設定ファイルのグループ化に役立つ点も挙げられる．たとえば，Git に対する .gitignore や .gitattributes のように単一のツールに対してそれぞれ異なる領域の設定を与える複数の設定ファイルが版管理リポジトリ上で管理される場合がある．複数ファイルが同一のツールに対して利用されるという意味的要素を挿入し，一つの構造として役立たせられる．

さらに，ツール名からファイルの利用タイミングを推測できる場合がある．例えば，CircleCI や ClangFormat の設定ファイルはそれぞれ，.circleci/ と .clang-format である．これらのファイル名から，それぞれ CI 環境やフォーマッタに関わるファイルであると類推が可能である．

6.2.2 書式に対応する拡張子の付加

よく知られた拡張子が存在する書式で記述されたファイルには対応する拡張子を付加するプラクティスも確認できる．設定ファイルの多くは，構造化されたテキストファイルである．その構造化には JSON や YAML などの標準化された書式 [22, 23] が利用されている場合が多く，これらの書式には標準の拡張子が定められている場合もある．

OS やアプリケーションはテキストファイルの書式を拡張子によって判別し，これによって利用者はシンタクスハイライトや補完などの恩恵を受けられる．UNIX 系 OS においては拡張子の付加がその

ファイルの利用において任意であったほか，アプリケーションの設定ファイルによく用いられる名前として .<ツール名>rc と拡張子がない名前が使われるため，一部その命名方式を引き継ぐファイルが存在している．しかしモダンな開発環境では拡張子の付加によって得られる恩恵が大きいため伝統的な命名の採用には議論の余地がある．

6.2.3 ignore ファイルへの命名

本節では，設定ファイルに一般的な命名プラクティスではなく一部のファイルに対して見られたプラクティスを述べる．ツールの中には，リポジトリ上のファイルの一部をそのツールの監視や影響下から例外的に外すための設定を行うファイルが存在する．ここではこれらのファイルを ignore ファイルと呼ぶ．版管理リポジトリにおいて最もよく見られる ignore ファイルとして Git の追跡から一部ファイルを外す .gitignore がある．他にも，Docker のイメージビルド時にコンテキストから除外するファイルを記述する .dockerignore や ESLint に対して一部ファイルに警告を出さないよう設定する .eslintignore が存在する．これら ignore ファイルには対応するツールはそれぞれ異なるものの命名やその書式に共通点が存在する．

設定ファイル名に対する規則 収集した ignore ファイルの名前は .<ツール名>ignore の名前であった．ツール名の付加というプラクティスが存在するが，後述の書式にはよく知られた拡張子は存在しないためファイル名に拡張子は付加されない．

同一の書式 収集した ignore ファイルはすべて，表 2 の List に分類される．記述内容についても同様であり，1 行に 1 つの glob パターンに否定等の拡張を許すパス指定もしくはコメントであった．これらの書式はコミュニティ上で*.gitignore-style* と呼ばれる場合もある．

これらは，設定ファイル全体というよりも，ignore ファイルに対して形成されたプラクティスである．しかし，広く知られた共通認識である以上，ignore ファイルに向けてその他の設定ファイルと画一的なガイドラインを提案しても受け入れられにくいと考えられる．

6.3 ガイドラインの提案に当たっての議論

5 節と 6.2 節では，マイニング調査の結果を基に現状確認できるプラクティス，つまり多数派を観察しそれらからガイドラインとしての提案事項を考察した．しかし，本研究で提案する内容がガイドラインである以上，現状の命名においてプラクティスと呼べるような共通認識が存在しない観点についても考察が必要である．

6.3.1 ドットファイルを利用すべきか

設定ファイルをドットファイルとして版管理リポジトリ上で管理すべきかを考察する．図 4 に示したように，ドットファイルであるような設定ファイル名は全体の 37% と少数派ではあるが，無視できる量ではない．そこで，本節ではガイドラインの提案に向けた一定の指針を示唆する．

本稿ではドットファイルでの管理を積極的に採用するべきであると提案する．ドットファイルでの管理で得られる恩恵として設定ファイルとその他のファイルの分離が挙げられる．本調査で用いたリポジトリのルートに存在するドットファイル名を調査したところ設定ファイルではないファイルの名前はドットファイル名全体のうち，11% (28/258) しか存在しなかった．つまり，ドットファイルであれば設定ファイルであるという認識が存在すると考えられる．この認識はドットファイルが持つ隠しファイルとしての特性に由来すると考えられる．設定ファイルのもつ役割上，意図しない編集を避ける必要があるほか開発者の多くが編集するプロダクトと関心を分離する必要もある．

また非ドットファイルでの管理の採用が消極的な理由であると考えられる点からもドットファイルでの採用がよいと考えられる．本研究における調査はすべてファイル名の集合に基づいて実施した．そのためプロジェクトで採用されているツール数や命名規則に対して数が増えているファイルの存在も考えられる．例えば，図 6 に示した Guided 名がある．これらの名前にはそれぞれの開発プロジェクト名が挿入されており，これらはほとんどの場合において唯一の名前となる．そのため，非ドットファイル名が増加していると考えらえる．

非ドットファイルの採用が増えている原因として，事前定義名に対して情報を付加した命名の採用もある．6.1.1 節で議論したように，同じ領域に対する設定を環境や用途ごとに用意する場合がある．このとき，ファイルの命名は標準である事前定義名に対してユーザが独自に情報を付加するという方法で命名している場合が多い．実際の例として `Dockerfile` がある．ファイルの自動同期ツールである Syncthing の開発リポジトリ[*4]では，ビルド環境として Docker を利用し，そのイメージを記述した Dockerfile を管理している．このとき，それぞれ異なる機能のビルドを実施するためにそれぞれに対する Dockerfile を用意しており，`Dockerfile.ursrv` や `Dockerfile.stupgrades` など 7 つのファイルがある．いずれも `Dockerfile.*` という標準ファイル名に対して情報を付加するという規則に従っているため非ドットファイルでの命名になっているといえる．

6.3.2 要素のセパレータや記述形式は何が適切か

命名に対して構文的な取り決めも議論する．現状のプラクティスとして 表 4 に示す通り，デリミタを用いたファイル名の書式としてよく用いられるのは dot.anotation と chain-case である．これに加え

[*4] `https://github.com/syncthing/syncthing/tree/6a3a28f` Accessed at 2026/01/26

て多くの場合，名前すべてを小文字で記述するファイル名がほとんどである．これは，UNIX 系 OS のファイルシステムが case sensitive であるのに加えて，サーチエンジンに対する最適化の文化が用いられていると考えられる [24].

一方で snake_case や CamelCase を採用する設定ファイル名は少ないが，プロダクトに関連するファイルでは命名規約により用いられる場合が多い．具体的には Python のパッケージ名，つまりファイル名は snake_case であり [21]，C#では CamelCase である．プロジェクト内のファイルの命名規則が一貫していることの重要性はこれまで述べたとおりである．しかし，設定ファイル名がプロダクトに用いられるファイル名の命名規則を同様に採用する点には議論の余地がある．理由としては，開発プロジェクトで用いられるプログラミング言語が単一とは限らない点や開発言語に対して横断的に利用されるツールの存在がある．すると，設定ファイルに対して適用すべき命名規則がプロダクトファイルの命名規則からは一つに限定できない．また，設定ファイルとプロダクトファイルの分離という面でもそれぞれ異なる命名規則を用いる恩恵がある．

ファイル名における記法以外にも，ディレクトリを用いていくつかの設定ファイルをグループ化する行為も一種の命名であると考えられる．この場合，設定ファイルのフルパスを考えるとディレクトリ名が要素に対応し，セパレータとしてスラッシュ（/）を用いているとも言える．この方法は設定ファイルのパスをエンドユーザが決定できるツールで有効である．ルートディレクトリ上のファイル数を削減できるためである．しかし，サブディレクトリに設定ファイル名の要素を閉じ込めるため一覧性が損なわれる場合もある．

6.3.3 ガイドライン提案の方針

ここでガイドラインの提案にあたる基本的な方針として定める．

単一リポジトリ内の設定ファイル間では単一の命名規則を採用する ソフトウェア開発にあたっては様々な対象に対して命名が必要になる．多くの場合において対象ごとにどのように命名するかを定めた命名規則が定められており，可読性や記述のしやすさの点からただ一つに定められている場合がほとんどである．リポジトリ上の設定ファイルに対しても同様にただ一つの命名規則を定める．本稿でも述べた通り，設定ファイルの命名に対する一貫性のなさが版管理リポジトリの構造理解を妨げる一因であると考えられるためである．

設定ファイルとそうでないファイルを分別可能にする 版管理リポジトリには様々なファイルが存在するが，設定ファイルではないファイル，例えばソースコードやドキュメントはプロダクトの成果物であり明らかに設定ファイルとは版管理リポジトリに設置される目的や意図が異なる．したがって両者は分別されるべきである．

リポジトリルート上のファイル数を削減する プロジェクト構造の大まかな理解は，初めにリポジトリ

ルートに現れるファイル名によって行われる．構造の理解という観点では情報量を絞る工夫が求められる．

設定ファイルが対応するツールがわかるようにする　設定ファイルの特徴として，開発環境で用いられる何らかの自動化ツールと結びつく点がある．構造把握という観点から設定ファイルの利用目的が理解できる名前が望ましく，ファイルが結びつくツール名はその最たる例であるといえる．また，調査の結果からもすでに開発者らの間で共通認識として存在するプラクティスである．

設定ファイルが用いる書式がわかるようにする　設定ファイルは多くの場合，構造化されたテキストファイルである．内容の把握や編集の容易さの観点から書式を簡単に把握できるようにする命名が必要である．書式の付加には拡張子を用いるべきである．すでにプラクティスとして存在するうえに，アプリケーションから支援を受けるためにも必要な場合が多いためである．

7　ガイドラインの提案

本章では調査によって判明した設定ファイルに対する命名プラクティスと，前章で議論した内容に基づいて設定ファイルの命名に対して一定のガイドラインを提案する．

7.1　ドットファイルとしての管理

設定ファイルを版管理リポジトリ上で管理する場合，ドットファイルによる管理を提案する．版管理リポジトリにおいてソースコードやテストコード，ドキュメントなどのプロダクトは非ドットファイルで管理される場合がほとんどである．そのため，設定ファイルのドットファイルでの管理によりドットファイルとそうでないファイルの分離によって設定ファイルとそうでないファイルの 2 つの関心を分離できる．

7.2　ツール名の挿入

6.2.1 節で考察した現状のプラクティスに基づき設定ファイルの命名に当たっては，対応するツールの名前を挿入すべきである．ツール名の挿入はファイルの利用目的の明確化のほか，複数ファイルのグループ化に役立つ．このためにツール名の挿入はファイル名の先頭にするべきである．また，ツール名によってはファイルが利用される場面が名前から類推出来るようになる．

7.3　書式に一致する拡張子の付加

設定ファイルの記述用いる書式に一般的に知られている拡張子や標準で定められている拡張子が存在するならば，それを設定ファイル名にも付加すべきである．拡張子の付加によって開発者らにとって内容の把握が容易になるだけでなく，アプリケーションによるシンタクスハイライトや補完などの恩恵を受けられる場合がある．また，設定ファイルの書式は構造化されたデータ記述形式である場合が多いため，書式がわかればそれらをソースコードなどのプロダクトから分離できる可能性もある．

7.4　セパレータの選択

設定ファイル名の形式をどのように選択するかを示唆する．第一にそのファイルが ignore ファイルであるならば，他のファイルの命名規則に従って `.<ツール名>ignore` とすべきである．この命名についてはすでに特殊なファイル群として共通認識が形成されているためである．

それ以外の場合，小文字を用いた chain-case の採用を提案する．これは，他のスタイルガイドでも言及されている [24] ように，case sensitive であるシステム上での簡単のためである．flatcase と比較してファイル名の意図が把握しやすい．また dot.notation が用いるドットは拡張子としての利用がすでに

共通認識やアプリケーションの実装として存在するため，その明確化の観点から他の部分では別のセパレータを利用するべきである．

また設定ファイルが一つのリポジトリに複数存在し，かつファイルの設置ディレクトリを自由に決定できる場合，ディレクトリを利用したグループ化を提案する．ルートディレクトリ上のファイル数を削減し構造を明確化するためである．

以上の提案に基づいて 図 1 の設定ファイル名またはパスを変更した様子を 図 9 に示す．

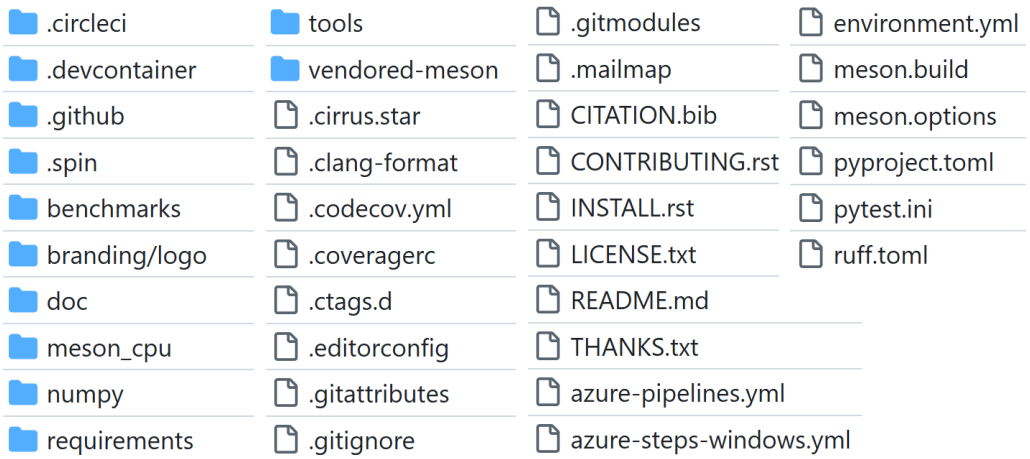

図 1（再掲）NumPy リポジトリ（numpy/numpy）のルートディレクトリ

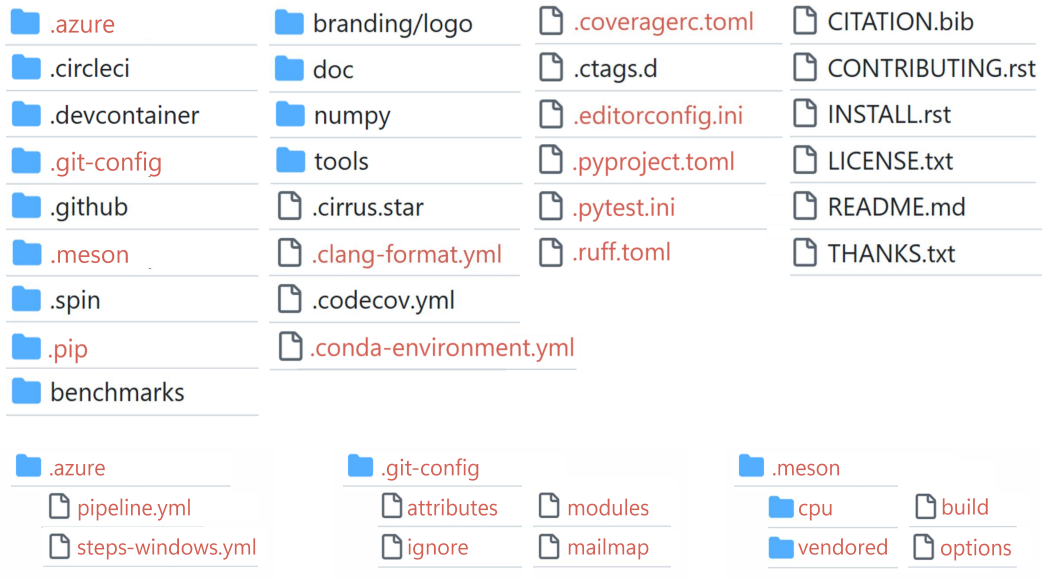

図 9:　図 1 にガイドラインの命名を適用した図

8 関連研究

8.1 開発プロセスにおける命名に関する研究

開発プロセスで発生する様々な命名は様々な観点で調査されている．命名の対象は様々であるがその中でも特に，ソースコード上に出現する変数やクラス，メソッドの名前である識別子をマイニングし，調査する研究は数多く遂行されている．Bulter らは Java ソースコード中のクラス名に対する命名規則をマイニングによって調査した [25]．Singer と Kirkham は Java クラス名をマイニングし，それらを意味的に解析した [26]．Newman らは C/C++，Java などの言語から識別子をマイニングし，それらに対して意味論に基づいた構造を分類した [27]．Malik らは JavaScript の関数名に対し，意味的解析を実施してその識別子名に対して実態がどのようであるかを示唆する手法を提案した [28]．これらの先行研究 [25, 26, 27, 28] が用いた調査手法の共通点として，識別子を構成する要素を意味的に分割し，それらを解析している点である．本研究はこの手法を設定ファイル名に対して適用した研究である．

また，命名における規則の存在やそれらの順守が開発にもたらす影響についても調査が実施されている [9, 10, 11]．Binkley らはソースコードの識別子に対して適用された命名規則が開発者らのソースコード理解に与える影響を調査した [9]．Allamanis らはコーディングスタイルの一貫性を高めるためのフレームワーク NATURALIZE を提案した [10]．He らは機械学習によってソースコード中の命名に関する問題を検出するシステムである Namer を提案した [11]．

本研究では版管理リポジトリにおける設定ファイルに対する命名に注目し，リポジトリマイニング調査を実施した．本研究の目的はソースコードやプロダクトであるソフトウェアのアーキテクチャに対する理解促進ではなく，開発環境に着目したリポジトリ全体の理解促進である．先行研究として，ソフトウェア構造をソースファイル名やディレクトリ構造から理解しようとする研究が存在する [29, 30] が，本研究が注目した設定ファイルはこれまでその命名や構造理解のためには注目されていなかった対象である [2]．

8.2 設定ファイルを対象とした研究

ソフトウェアに対する設定を調査する研究も数多く行われている．マイニング対象としてよく調査されている設定ファイル群として Dockerfile が挙げられる [31, 32, 33, 34, 35]．Henkel らは GitHub 上の Dockerfile を収集する手法を提案し，そのデータセットを提供した [31]．Zhou らは Dockerfile を解析し，暗黙的な Dockerfile 記述上のルールとそれらに対する違反を検出するツールを提案した [32]．Ksontini らは Dockerfile に対するリファクタリングに注目し，それらをマイニングおよび分析する手法を提案した [33] また，Dockerfile のリファクタリングに対して現状を調査し，その自動化可能性を考察した [34]．Rosa らはリンタによって検出された Dockerfile 上のコードスメルを調査した [35]．

Dockerfile のほかにも，同じく宣言的にツールの動作を記述したファイルのひとつである Makefile も調査されている [36]．Douglas と James は Makefile をマイニングしそれらを解析後，ファイルの保守に必要な労力を推定した [36]．これらの研究は特定の設定ファイルに対し，従来ソースコードに対して用いられてきた古典的な SE 解析手法を適用した研究である．

特定のツールだけでなく，ソフトウェア開発において同じプロセスを自動化するツール群の設定を一元的に調査や比較をする研究が実施されている [7, 37]．Beller らは複数のソースコード静的解析ツールに対する設定を調査し，その利用の現状を明らかにした [7]．Mazrae らは継続的インテグレーションおよびデプロイメント環境を提供する複数のプラットフォームの設定を調査し，開発者らが用いる機能を調査した [37]．

本研究は特定のツールやプロセスに限らず，版管理リポジトリ上のすべての設定ファイルに注目している．一方で，本研究の調査においては設定ファイルの記述内容は詳細に解析していない．研究の目的が，各設定を考察しそれらが与える影響を調査する点ではなく，リポジトリ構造の理解にあるためである．

9　妥当性への脅威

本研究の 4.1.1 節には妥当性への脅威が存在する．本研究で実施した調査はリポジトリマイニング調査であり，データセットとして用いるリポジトリの選定によって調査結果が変化する可能性がある．リポジトリ選定におけるリポジトリの成熟具合を測るメトリクスとして，本調査ではリポジトリが GitHub 上で得たスターを用いた．しかしスター数はプロジェクトに対するプロモーションが影響する可能性が指摘されており [38]，必ずしもリポジトリの成熟と強く相関しない可能性がある．他にリポジトリの成熟を示すメトリクスとして当該リポジトリのフォーク数がある．Jiang らの調査によれば，OSS の開発者らはリポジトリに対して機能追加などのプルリクエストを送信するためにフォークを作成する場合が多い [39]．したがって，フォーク数の多いリポジトリは多くの貢献を受けておりより成熟した一般的なプラクティスを観察できる可能性がある．

10　おわりに

本研究では，版管理リポジトリ構造の整理を目的として設定ファイル名にガイドラインを提案した．またガイドラインの提案にあたっては，マイニング調査によって現状存在する設定ファイルに対する命名プラクティスを調査した．RQ1 では設定ファイル名それぞれに対して対応するドキュメントの目視調査を行い，設定ファイル名の多くがツール開発者によって事前定義されていると明らかにした．RQ2 では設定ファイル名をいくつかの要素で分析し，設定ファイルの命名に対するプラクティスとして対応するツール名の挿入と拡張子の付加の 2 つの存在を確認した．さらに調査によって発見した命名のプラクティスについてそれぞれが版管理リポジトリの構造理解にもたらす恩恵を考察しガイドラインを提案した．

今後の課題として調査の細分化が挙げられる．本文中でも指摘した通り，版管理リポジトリが採用するディレクトリ構造や命名規則はそのプロジェクトで用いられるプログラミング言語やフレームワークによってさまざまである．本研究においては一つのプログラミング言語を一つのエコシステムとして扱い，版管理リポジトリを収集した．しかし，近年のソフトウェアや Web 開発においては多様なフレームワークが用いられておりそれぞれにおいて命名規則が異なる可能性がある．したがって，版管理リポジトリの収集において利用しているフレームワークなどのエコシステムに注目した細分化が必要になる．

他にも，ツールに注目した設定ファイル名の調査も考えられる．本研究の調査では，エンドユーザであるプロジェクトに注目した調査を実施した．しかし，様々な言語やエコシステムに横断的なツールの存在や複数プログラミング言語を用いた開発の広まりを考慮すると，ツール開発者らの文化も設定ファイル名に多大な影響を及ぼしていると考えられる．

また，提案したガイドラインの評価も課題として考えられる．版管理リポジトリにおけるディレクトリ構造の整頓や命名規則の提案は日々行われている．そのため本研究で提案したガイドラインに基づく版管理リポジトリの整理を示唆し受け入れられるかを評価し適切であるかを確認する必要がある．

謝辞

本研究の遂行においては多くの方からご助力を賜りました．この場を借りてお礼申し上げます．

楠本真二 教授には学部での研究室配属から 3 年間にもわたり研究室での輪講や研究活動など多くの学びの場を提供していただきました．また，研究においては中間報告の場などで研究やその論述の不足をご指摘やご指導をいただきました．深く感謝いたします．

柗本真佑 准教授には学部の卒業研究から続けて 3 年間にわたって熱心なご指導を賜りました．研究の遂行に当たって必要な議論や相談，論文の執筆にお時間をいただいたきました．先生からのご指導を賜り，それをきっかけに研究テーマに対して熟考できました．研究活動のほかにも共同作業に必要なコミュニケーションや自分の考えの言語化とそれに必要な文章力など社会人として必須のスキルもご指導いただきました．心より感謝いたします．

事務補佐員の橋本美砂子 様には出張やアルバイトなどに必要な事務手続きのサポートをいただきました．特に，私が多忙だった時期には学内の様々な方に連絡を取るお気遣いをいただきました．他にも，私を含めた学生に様々な差し入れや催しの実施などさまざまなお気遣いをいただきました．深く感謝いたします．

肥後研究室の肥後芳樹 教授と Raula Gaikovina Kula 教授，そして Olivier Nourry 助教にはセミナーや中間報告の場で発表に対してのご指摘やご質問をいただきました．自力では注目できなかった視点で研究を見つめなおし，よりよい研究や論述につなげられました．お礼申し上げます．また，Olivier Nourry 助教には国際会議に向けた論文の執筆においてご助力をいただきました．改めてお礼申し上げます．

楠本研究室で共に研究生活を送った学生の皆様には様々な刺激をいただきました．皆様が書く文章や研究を遂行する上で用いる技術は私自身の研究活動を見つめなおすきっかけになりました．特に同期の皆様には，研究以外でも学びや楽しみの場をともにでき研究生活でより多くのものを得られたと思っています．深く感謝いたします．

最後に，2 年間の博士前期課程の生活を支えてくれた家族に改めて感謝いたします．

参考文献

[1] Gousios, G., Kalliamvakou, E. and Spinellis, D.: Measuring developer contribution from software repository data, in *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pp. 129–132 (2008).

[2] Ma, Y., Fakhoury, S., Christensen, M., Arnaoudova, V., Zogaan, W. and Mirakhorli, M.: Automatic classification of software artifacts in open-source applications, in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pp. 414–425 (2018).

[3] Ding, Z. Y. and Le Goues, C.: An Empirical Study of OSS-Fuzz Bugs, in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pp. 131–142 (2021).

[4] Cito, J. and Gall, H. C.: Using docker containers to improve reproducibility in software engineering research, in *Proceedings of the International Conference on Software Engineering Companion (ICSE)*, pp. 906–907 (2016).

[5] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and Gall, H. C.: An empirical analysis of the Docker container ecosystem on GitHub, in *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 323–333 (2017).

[6] Tómasdóttir, K. F., Aniche, M. and Deursen, van A.: Why and how JavaScript developers use linters, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 578–589 (2017).

[7] Beller, M., Bholanath, R., McIntosh, S. and Zaidman, A.: Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software, in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 470–481 (2016).

[8] Abdalkareem, R., Oda, V., Mujahid, S. and Shihab, E.: On the impact of using trivial packages: An empirical case study on npm and pypi, *Journal on Empirical Software Engineering*, Vol. 25, No. 2, pp. 1168–1204 (2020).

[9] Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C. and Sharif, B.: The impact of identifier style on effort and comprehension, *Journal on Empirical Software Engineering*, Vol. 18, No. 2, pp. 219–276 (2013).

[10] Allamanis, M., Barr, E. T., Bird, C. and Sutton, C.: Learning natural coding conventions, in *Proceedings of International Symposium on Foundations of Software Engineering (FSE)*, pp. 281–293 (2014).

[11] He, J., Lee, C.-C., Raychev, V. and Vechev, M.: Learning to find naming issues with big code and small supervision, in *Proceedings of International Conference on Programming Language Design and Implementation (PLDI)*, pp. 296–311 (2021).

[12] Wessel, M., Mens, T., Decan, A. and Mazrae, P. R.: *The GitHub Development Workflow Automation Ecosystems*, pp. 183–214, Springer International Publishing (2023).

[13] Cannon, B., Stufft, D. and Kluyver, T.: PEP 518 – Specifying Minimum Build System Requirements for Python Projects, `https://peps.python.org/pep-0518/` (2016), Python Enhancement Proposal 518.

[14] Cannon, B., Kluyver, T., Moore, P., Stufft, D. and Warsaw, B.: PEP 621 – Storing Project Metadata in pyproject.toml, `https://peps.python.org/pep-0621/` (2020), Python Enhancement Proposal 621.

[15] Palomba, F., Zaidman, A., Oliveto, R. and De Lucia, A.: An Exploratory Study on the Relationship between Changes and Refactoring, in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 176–185 (2017).

[16] Robredo, M., Esposito, M., Palomba, F., Peñaloza, R. and Lenarduzzi, V.: In Search of Metrics to Guide Developer-Based Refactoring Recommendations, `https://arxiv.org/abs/2407.18169` (2024).

[17] Borges, H., Hora, A. and Valente, M. T.: Understanding the Factors That Impact the Popularity of GitHub Repositories, in *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*, pp. 334–344 (2016).

[18] Siegmund, N., Ruckel, N. and Siegmund, J.: Dimensions of software configuration: on the configuration context in modern software development, in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 338–349 (2020).

[19] Jungwirth, G., Saha, A., Schröder, M., Fiebig, T., Lindorfer, M. and Cito, J.: Connecting the. dotfiles: Checked-in secret exposure with extra (lateral movement) steps, in *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 322–333 (2023).

[20] Anquetil, N. and Lethbridge, T.: Recovering software architecture from the names of source files, *Journal of Software Maintenance*, Vol. 11, No. 3, pp. 201–221 (1999).

[21] Rossum, G. v., Warsaw, B. and Coghlan, A.: PEP8 – Style Guide for Python Code, `https://peps.python.org/pep-0008` Accessed at 2026/01/20 (2001), Python Enhancement Proposal 8.

[22] Crockford, D.: Request for Comments: 4627, https://www.ietf.org/rfc/rfc4627.txt Accessed at 2026/01/19 (2006).

[23] YAML Language Development Team, : YAML Ain't Markup Language (YAML™) version 1.2.2, https://yaml.org/spec/1.2.2 Accessed at 2026/01/24 (2021).

[24] Google, : Google developer ocumentation style guide Filenames and file types, https://developers.google.com/style/filenames Accessed at 2026/1/30 (2025).

[25] Butler, S., Wermelinger, M., Yu, Y. and Sharp, H.: Mining Java class naming conventions, in *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 93–102 (2011).

[26] Singer, J. and Kirkham, C.: Exploiting the Correspondence between Micro Patterns and Class Names, in *Proceedings of International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 67–76 (2008).

[27] Newman, C., AlSuhaibani, R., Decker, M., Peruma, A., Kaushik, D., Mohamed, M. and Hill, E.: On the generation, structure, and semantics of grammar patterns in source code identifiers, *Journal on Systems and Software*, Vol. 170, p. 110740 (2020).

[28] Malik, R., Patra, J. and Pradel, M.: NL2Type: Inferring JavaScript function types from natural language information, in *Proceedings of International Conference on Software Engineering (ICSE)*, pp. 304–315 (2019).

[29] Preschern, C.: Patterns for Organizing Files in Modular C Programs, in *Proceedings of European Conference on Pattern Languages of Programs (EuroPLoP)*, No. 1, pp. 1–15 (2020).

[30] Iddon, C., Giacaman, N. and Terragni, V.: GradeStyle: GitHub-Integrated and Automated Assessment of Java Code Style, in *Proceedings of International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 192–197 (2023).

[31] Henkel, J., Bird, C., Lahiri, S. and Reps, T.: A dataset of Dockerfiles, in *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 528–532 (2020).

[32] Zhou, Y., Zhan, W., Li, Z., Han, T., Chen, T. and Gall, H.: Drive: Dockerfile rule mining and violation detection, *Transactions on Software Engineering and Methodology*, Vol. 33, No. 2, pp. 1–23 (2023).

[33] Ksontini, E., Abid, A., Khalsi, R. and Kessentini, M.: Drminer: A tool for identifying and analyzing refactorings in Dockerfile, in *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 584–594 (2024).

[34] Ksontini, E., Mastouri, M., Khalsi, R. and Kessentini, W.: Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential, in *Proceedings of International*

Conference on Mining Software Repositories (MSR), pp. 788–800 (2025).

[35] Rosa, G., Scalabrino, S., Robles, G. and Oliveto, R.: Not all Dockerfile smells are the same: An empirical evaluation of hadolint writing practices by experts, in *Proceedings of International Conference on Mining Software Repositories (MSR)*, pp. 231–241 (2024).

[36] Douglas, M. and James, C.: On the maintenance complexity of Makefiles, in *Proceedings of International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pp. 50–56 (2016).

[37] Mazrae, P., Mens, T., Golzadeh, M. and Decan, A.: On the usage, co-usage and migration of CI/CD tools: A qualitative analysis, *Journal on Empirical Software Engineering*, Vol. 28, No. 2, p. 52 (2023).

[38] Hudson, B. and Marco, T. V.: What' s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform, *Journal of Systems and Software*, Vol. 146, pp. 112–129 (2018).

[39] Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S. and Zhang, L.: Why and how developers fork what from whom in GitHub, *Journal on Empirical Software Engineering*, Vol. 22, No. 1, pp. 547–578 (2017).