Automated Identification of Flaky Builds using Knowledge Graphs*

Florent Moriconi[1,2], Raphael Troncy[2], Aurélien Francillon[2] and Jihane Zouaoui[1]

[1]*AMADEUS IT Group, Sophia Antipolis, France*
[2]*EURECOM, Sophia Antipolis, France*

Abstract
DevOps' mindset encourages the use of Continuous Integration to quickly identify regressions introduced on code changes by running a set of checks (i.e., a build) on each commit or pull-request. This approach leads to a very large number of builds every day: in the order of tens of thousands at Amadeus. Builds can either fail because of a regression which has been introduced (a legitimate failure) or because of infrastructure instabilities (an illegitimate failure). Investigating the cause of failure is generally a manual process done by developers. If the failure is due to the infrastructure, the developers have wasted time investigating a failure they cannot directly fix. Therefore, there is a strong interest in automating identification of builds broken by infrastructure instabilities. Considering the large amount of heterogeneous data available about a build, we propose a method based on knowledge graph embeddings to automatically identify whether the failure is legitimate or not. We first propose an ontology to model the continuous integration process widely used in software engineering. To this end, we follow the best practices from the knowledge engineering community using competency questions and we also re-use well known ontologies. Next, we propose a method to build a labelled dataset and we experiment with knowledge graph embedding techniques. We empirically demonstrate that we can predict whether a failure is due to the infrastructure or not with an accuracy of 94% on a balanced validation dataset.

Keywords
Ontology modeling, Knowledge Graphs, Software engineering, Continuous Integration, Root cause analysis,

1. Introduction

Modern software development practices like DevOps try to shorten the System-Development Life Cycle (SDLC) to allow frequent and safe updates. The DevOps mindset encourages the use of Continuous Integration (CI) to reduce the feedback loop by ensuring on every code change that the code still meets quality requirements. This allows developers to be alerted quickly when problems with code changes occur. In practice, a CI system is often composed of a build step (compilation of the source code), static code analysis (e.g., code linting tools, vulnerability scanners), peer-review, binary scan, tests (e.g., unit-testing, end-to-end testing, regression testing, functional validation) and deployment of produced artifacts (e.g., binaries,

EKAW-C 2022: Companion Proceedings of the 23rd International Conference on Knowledge Engineering and Knowledge Management, September 26–29, 2022, Bozen-Bolzano, IT
florent.moriconi@amadeus.com (F. Moriconi); raphael.troncy@eurecom.fr (R. Troncy); aurelien.francillon@eurecom.fr (A. Francillon); jihane.zouaoui@amadeus.com (J. Zouaoui)

docker images, tarball archives, pip packages). The CI is orchestrated by a CI orchestrator where the most important ones are Jenkins, Gitlab CI, Github Actions, Travis CI, Circle CI and Drone.

This paper is structured as follows. In Section 1.1, we describe the vocabulary, main components and processes used in continuous integration systems. In Section 1.2, we present approaches explored in the literature to perform automated root cause analysis of CI failures. In Section 2, we present the modeling of an ontology to represent the continuous integration process. This ontology is then used to populate a knowledge graph that represents heterogeneous information about successful and failed builds together with information about external factors. In Section 3, we propose a simple method to create a labelled dataset and we experiment with knowledge graph embedding methods for predicting when build failures are illegitimate. We empirically demonstrate that our approach can reliably predict which build failures are due to instabilities. In Section 4, we conclude and outline some future works.

1.1. Continuous Integration

A run of CI steps is called a build. A build can be triggered on code change (Pull-Request, branch event), by a timer or by a user. A build failure may be legitimate (a requirement for success is not met, e.g., the code is not linted correctly), or illegitimate (e.g., due to a temporary network failure, no space left on the device). Finding why a build has failed may be complex and it often requires manual work by software developers. While build failures are not an issue by design, illegitimate failures take time to be investigated while they are often temporary or out of the scope of a developer mission.

Builds that failed because of infrastructure instabilities are similar to flaky tests [1] since a build fails if at least one of its tests fails. However, builds tend to have much more dependencies and source of issues than tests: network access, local storage, use of remote APIs, environment updates, tool configurations, etc. Even large companies such as Google and Microsoft have flaky tests [2]: 41% and 26% of their tests, respectively, have some level of flakiness. Infrastructure instabilities are often limited in time and non-deterministic: restarting the build can fix the issue. This solution is often chosen by developers, sometimes without even looking for the root cause, in particular if developers are facing numerous instabilities. However, blindly restarting all broken builds consumes computing power, which leads to higher energy consumption and infrastructure load. Thus, there is a strong interest in solutions that allow automated root cause analysis of build failures to restart only builds that failed because of infrastructure instabilities.

1.2. Related work

Attempts have been made in the literature to provide automated root cause analysis of CI build failures [3][4]. Root cause analysis can be provided as: (i) locating the root cause message in the log; (ii) classifying the whole log into categorical root causes; (iii) classifying log lines as normal or abnormal lines; (iv) providing a fix for the root cause (e.g., a code patch). In this work, we focus on the binary classification task to identify builds broken by infrastructure instabilities. A large number of methods rely on log analysis, using either rule-based methods (e.g., regular expressions, metrics extraction) [5] or natural language processing approaches [6, 7]. However, methods that rely on expert knowledge are difficult to use in practice considering the number of possible root causes in large companies and the continuous evolution of software: new software versions, log format evolution, evolving build environment, etc. Furthermore, CI logs are composed of various software stacks and in-house tooling written in different languages (shell scripting, package managers, test orchestration tools, code analyzers, etc.). Therefore, CI logs tend to be highly heterogeneous compared to production logs where only one software is running per log stream. Thus, log parsing approaches such as Drain [8] or Logram [9] tend to poorly perform to identify log events in CI build logs.

Haben et al. [10] work on detecting flaky tests in CI builds of the Chromium project using a curated list of metrics such as run duration and failure history. While they achieved a F1-score of 94% for predicting flaky tests, this method is highly tailored for Chromium's CI setup and practices. Furthermore, they only focus on test failures, while in this work, we focus on build failures, which include additional root causes such as network issues, node issues or program mis-configuration. Amar et al. [11] present a fault localisation tool based on historical test logs. The proposed solution flags less than 1% of log lines for manual inspection (which may be huge considering the size of the logs as input). However, the authors concede that a system based solely on logs will not be able to detect all faults because not all errors are found in the logs, which motivates our work to propose a method that could easily integrate multiple data sources.

2. An Ontology for Representing Builds in a Continuous Integration Process

We hypothesize that Knowledge Graphs (KG) are best suited for representing builds in the continuous integration process considering their ability to easily integrate heterogeneous information such as series of events of variable length and categorical data which are typical of the traces produced by CI systems. A knowledge graph allows to decompose the process into elements (e.g., a build machine, a build trigger) represented as nodes in the graph. We expect to walk the graph for detecting failure patterns that would be much more difficult to detect independently.

We first design an ontology to represent builds in continuous integration systems. The ontology is available at https://purl.org/cibuilds. The ontology contains 14 classes, 13 object properties and 10 data properties. Figure 1 illustrates the representation of a CI build in the knowledge graph. The represented build has 2 stages (Compile and Upload). It was triggered because of a code change in a source file (src/main.py), and it has failed because the stage Upload has failed.

In order to populate the knowledge graph, we first leverage the API of the CI orchestrator which provides information such as why the build was triggered, on which build machine it ran, the different steps of the build, etc. Graphs are automatically created for new builds by a Python program in near real time: as soon as a build is finished, build metadata are retrieved from the HTTP REST API of the CI orchestrator. However, there is much more information available: insights from the build log, the state of the environment (e.g., outages), network connections between the build and its environment (e.g., API calls, dependencies download) or other metrics created by other systems (e.g., infrastructure load, command metrics). It is also possible to enrich available data using simple heuristics. This heterogeneity of information

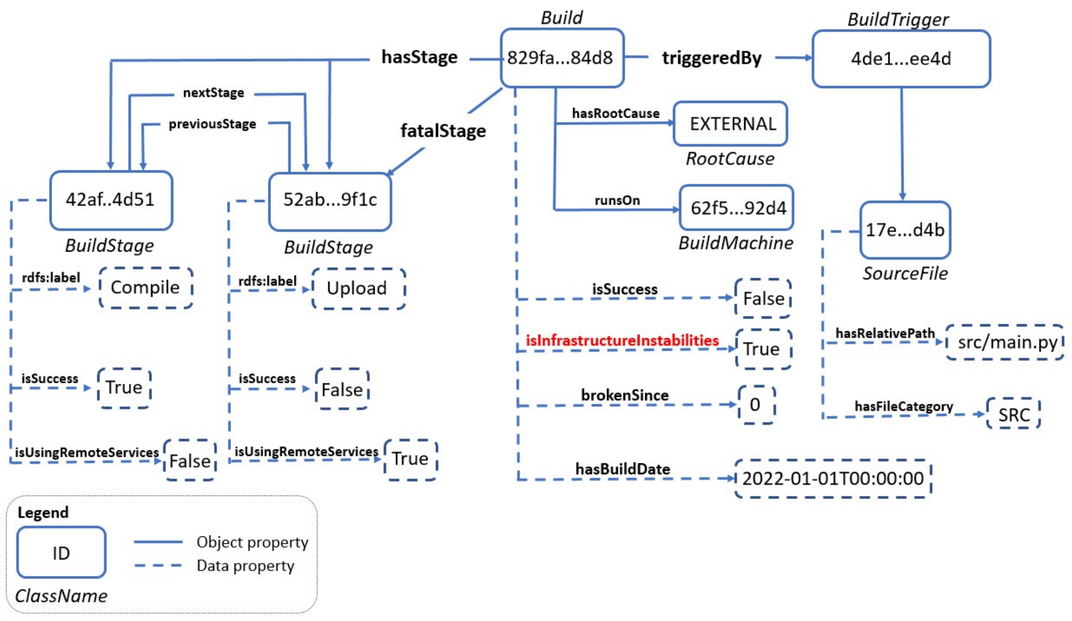

Figure 1: Excerpt of the Knowledge Graph representing a specific Build together with its stages and states. The red arrow corresponds to the edge we expect to predict.

coming from multiple systems justifies a data integration and enrichment approach which is best conducted using knowledge graphs.

We generate the value of the object property `isUsingRemoteServices` using an heuristic. If the `rdfs:label` value of the build stage (i.e., the stage name) contains at least one of the words *publish, deploy, download, push, remote, upload, download, checkout* (case-insensitive) then we consider that this stage is using remote services. Most of the builds have stages that rely on external dependencies to work properly: download dependencies from a remote server, use web APIs, push compilation artifacts to remote storage, etc. Broken builds because of infrastructures instabilities are often the consequence of failure to use remote services. Therefore, it is relevant to document whether a stage is using remote services or not. Network traffic analysis can help to precisely identify the remote resources in use as well as transport failure (e.g., TCP timeout) or application failure (e.g., HTTP status code >= 400) but it is harder to use in practice. However, nowadays, most of the traffic is encrypted using TLS.

Integrating service status is also interesting to model in the graph if remote services are available or not during the build execution. While a broken remote service can lead to build failure, services can have a partial outage: they fail to process only a small part of total requests. Therefore, service status can be green even when the root cause is related to this service.

Given the proposed ontology to explicitly represent knowledge about CI builds, the following step is to automatically identify builds broken because of infrastructure instabilities using root cause analysis methodologies.

3. Root Cause Analysis

In this work, we focus on the binary classification of builds between REPO class (root cause is related to the code repository) and EXTERNAL class (root cause is related to the infrastructure). This classification problem is equivalent to a link prediction problem for the `isInfrastructureInstabilities` edge (in red in Figure 1) between a Build object and a boolean value: this is a binary representation of `hasRootCause` property: this is equal to true if `hasRootCause` is connected to `EXTERNAL` root clause class, or equal to false if connected to another class. It aims to simplify (reduce from multi-class to binary) the prediction task.

3.1. Creating a Labelled Dataset

Labelled data is needed for training a supervised learning method, but also for performance validation in unsupervised settings. We propose a method to automatically label a part of historical builds (i.e., not the latest build). We assign a label to a failed build depending on code changes between the failed build and the next build, if and only if the following build is successful. In other terms, our method is available for all builds that met the following condition:

$$\forall n \text{ such as } build_n \text{ failed and } build_{n+1} \text{ is successful}$$

If there is no code change between these builds (i.e., the build was restarted without code change), we consider that the root cause is related to the infrastructure; otherwise, we consider that the root cause is related to the code repository. In other words, if a broken build is restarted without any modification and it passed, then it has failed because of infrastructure instabilities. Detecting code changes between builds may be hard: during our experiment, the CI orchestrator returned wrong results in few cases. It notably happened when git history was rewritten and force pushed or when config files were injected at build time (without code commit). To handle these cases, we compare git commit hashes to detect code changes. If not available, builds were ignored. In practice, this method helps to label around 10% of historical failed builds. We made the hypothesis that these builds are representative of all failed builds.

3.2. Baseline

We developed a baseline model based on natural language processing methods. This model is based on TF-IDF for representing build log and Random Forest to predict if the root cause is related to the infrastructure. TF-IDF is composed of two terms: Term Frequency (TF) and Inverse Document Frequency (IDF). The main idea is that words that appear in a low number of documents are more significant to classify documents, as well as words that appear often in a document. TF-IDF is part of *Bag of Words* (BoW) approaches: the context of log lines is lost. Random Forest is an evolution of Decision Tree classifier that aims to provide better performance while requiring only little configuration. On a balanced cross-validation dataset (CV=5), this approach has a mean accuracy of 67% and a mean recall of 60% which leads to a F1-score of 63%.

3.3. Knowledge Graph Embeddings

We propose to use Knowledge Graph Embedding (KGE) methods on the large and heterogeneous graph about CI builds presented in Section 2. KGE aim to represent entities as vectors that can be compared with a distance function and similarity metrics. In our experiments, we tried TransE [12], DistMult [13], HolE [14] and TorusE [15]. We did not use literal values to create the embeddings. For instance, TransE models relations as a translation from head to tail entities.

$$\mathrm{e}_h + \mathrm{e}_r \approx \mathrm{e}_t$$

Reworking the expression above and applying the p-norm leads to the TransE interaction function:

$$f(h,r,t) = -\|\mathrm{e}_h + \mathrm{e}_r - \mathrm{e}_t\|_p$$

The objective of the learning phase is to maximize the TransE interaction function. When the algorithm is trained, similar entities should have similar vectors. In the context of this work, the head entity is a `Build`, the relation is `isInfrastructureInstabilities` and the tail entity is a Boolean.

3.4. Experiment

In our experiment, we use a dataset composed of 829,638 triples that represent 8,586 builds where we know the real root cause. The builds were scraped from Amadeus' continuous integration systems and therefore cannot be released publicly. We labelled them following the methodology described in Section 3.1. The dataset was split into training (90%) and testing (10%) sets. For the embedding generation, we used the PyKeen framework. Our code is available at https://purl.org/cibuilds/code-v1 to support reproducibility. Table 1 provides the performance results. We define the balanced accuracy score as

$$\frac{TP + TN}{TP + TN + FP + FN}$$

We define F1-score as

$$\frac{2 * TP}{2 * TP + FP + FN}$$

Where TP stands for True Positive (count), TN for True Negative, FP for False Positive, and FN for False Negative. We used an artificially balanced dataset to validate the performance where we vary the embedding sizes and embedding methods. We choose to limit the hyperparameter optimisation to embedding size only to reduce search space. We used PyTorch's ExponentialLR learning rate (PyKeen's default). Low dimensional spaces result in poorer classification accuracy. However, with an embedding dimension of 50 we outperform the baseline approach (TF-IDF using the Random Forest classifier) for 3 over 4 methods, with a simpler and easier to explain algorithm. The highest classification accuracy and F1-score is achieved for TorusE embedding method with an embedding dimension of 300. When a build is predicted to have failed because of infrastructure, restarting it could fix it. With an accuracy score of more than 94%, less then 6% of restarted builds were restarted while it was not relevant considering the real root cause.

Embedding method	Embedding dimension	balanced accuracy score	F1 score
TransE	50	0.875	0.75
TransE	100	0.898	0.795
TransE	300	0.839	0.679
DistMult	50	0.799	0.597
DistMult	100	0.895	0.79
DistMult	300	0.875	0.75
HolE	50	0.615	0.231
HolE	100	0.867	0.733
HolE	300	0.772	0.545
TorusE	50	0.837	0.674
TorusE	100	0.908	0.816
TorusE	**300**	**0.948**	**0.897**

Table 1
Accuracy and F1 scores for the prediction of the failed build due to instabilities while varying the embedding dimension and embedding algorithms. The highest F1-score is achieved for TorusE with an embedding dimension of 300.

4. Conclusion and Future Work

Analysing continuous integration builds raises new challenges: log analysis is significantly more complex than application logs, builds are composed of steps, build dependencies to remote services are not always well defined, builds can be scheduled on different nodes, etc. While our results for flaky builds identification outperformed our baseline, using graph embedding should allow to provide finer-grained automated root cause analysis of build failures by allowing easier data integration (network traffic analysis, insights from logs, commands, infrastructure metrics, etc.). Fine-grained root classification will also help to develop methodologies for automated fixes: rollback the source commit that caused the build to fail, generate a code patch, rerun the build only when the root cause is fixed, etc. Organising knowledge about builds and making it queryable will also help to advocate best practices for continuous integration by detecting bad patterns (i.e., patterns that lead to more failures): one-stage build, rarely executed builds, dependencies on remote services for unit-test stage, etc.

References

[1] M. Gruber, Tackling Flaky Tests: Understanding the Problem and Providing Practical Solutions, in: 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 1–3. doi:10.1109/ASE51524.2021.9678701.

[2] O. Parry, G. M. Kapfhammer, M. Hilton, P. McMinn, A Survey of Flaky Tests, ACM Transactions on Software Engineering and Methodology 31 (2022) 1–74. URL: https://dl.acm.org/doi/10.1145/3476105. doi:10.1145/3476105.

[3] J. K. Bastida, Applying Machine Learning to Root Cause Analysis in Agile CI/CD Software Testing Environments, phdthesis, Aalto University, Espoo, Finland, 2018.

[4] F. Hassan, Tackling build failures in continuous integration, in: 2019 34th IEEE/ACM

International Conference on Automated Software Engineering (ASE), 2019, pp. 1242–1245. doi:10.1109/ASE.2019.00150, ISSN: 2643-1572.

[5] J. Kahles, J. Torronen, T. Huuhtanen, A. Jung, Automating Root Cause Analysis via Machine Learning in Agile Software Testing Environments, in: 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE, 2019, pp. 379–390. URL: https://ieeexplore.ieee.org/document/8730163/. doi:10.1109/ICST.2019.00047.

[6] C. Bertero, M. Roy, C. Sauvanaud, G. Tredan, Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection, in: 28th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2017, pp. 351–360. URL: http://ieeexplore.ieee.org/document/8109100/. doi:10.1109/ISSRE.2017.43.

[7] N. Aussel, Y. Petetin, S. Chabridon, Improving Performances of Log Mining for Anomaly Prediction Through NLP-Based Log Parsing, in: 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2018, pp. 237–243. URL: https://ieeexplore.ieee.org/document/8526889/. doi:10.1109/MASCOTS.2018.00031.

[8] P. He, J. Zhu, Z. Zheng, M. R. Lyu, Drain: An Online Log Parsing Approach with Fixed Depth Tree, in: International Conference on Web Services (ICWS), IEEE, 2017, pp. 33–40. URL: http://ieeexplore.ieee.org/document/8029742/. doi:10.1109/ICWS.2017.13.

[9] H. Dai, H. Li, C.-S. Chen, W. Shang, T.-H. Chen, Logram: Efficient Log Parsing Using *n*n-Gram Dictionaries, IEEE Transactions on Software Engineering 48 (2022) 879–892. doi:10.1109/TSE.2020.3007554.

[10] G. Haben, S. Habchi, M. Papadakis, M. Cordy, Y. L. Traon, Discerning Legitimate Failures From False Alerts: A Study of Chromium's Continuous Integration, arXiv 2111.03382, 2021. URL: https://arxiv.org/abs/2111.03382.

[11] A. Amar, P. C. Rigby, Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs, in: 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 140–151. URL: https://ieeexplore.ieee.org/document/8812113/. doi:10.1109/ICSE.2019.00031.

[12] A. Bordes, N. Usunier, A. Garcia-Durán, J. Weston, O. Yakhnenko, Translating Embeddings for Modeling Multi-Relational Data, in: 26th International Conference on Neural Information Processing Systems (NIPS), Curran Associates Inc., 2013, pp. 2787––2795.

[13] B. Yang, W.-t. Yih, X. He, J. Gao, L. Deng, Embedding entities and relations for learning and inference in knowledge bases, 2014. URL: https://arxiv.org/abs/1412.6575. doi:10.48550/ARXIV.1412.6575.

[14] M. Nickel, L. Rosasco, T. A. Poggio, Holographic embeddings of knowledge graphs, in: D. Schuurmans, M. P. Wellman (Eds.), Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA, AAAI Press, 2016, pp. 1955–1961. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12484.

[15] T. Ebisu, R. Ichise, Toruse: Knowledge graph embedding on a lie group, in: S. A. McIlraith, K. Q. Weinberger (Eds.), Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, AAAI Press, 2018, pp. 1819–1826. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16227.