V2: Fast Detection of Configuration Drift in Python

Eric Horton, Chris Parnin
North Carolina State University
Raleigh, NC, USA
{ewhorton, cjparnin}@ncsu.edu

***Abstract*—Code snippets are prevalent, but are hard to reuse because they often lack an accompanying environment configuration. Most are not actively maintained, allowing for drift between the most recent possible configuration and the code snippet as the snippet becomes out-of-date over time. Recent work has identified the problem of validating and detecting out-of-date code snippets as the most important consideration for code reuse. However, determining if a snippet is correct, but simply out-of-date, is a non-trivial task. In the best case, breaking changes are well documented, allowing developers to manually determine when a code snippet contains an out-of-date API usage. In the worst case, determining if and when a breaking change was made requires an exhaustive search through previous dependency versions.**

We present V2, a strategy for determining if a code snippet is out-of-date by detecting discrete instances of configuration drift, where the snippet uses an API which has since undergone a breaking change. Each instance of configuration drift is classified by a failure encountered during validation and a configuration patch, consisting of dependency version changes, which fixes the underlying fault. V2 uses feedback-directed search to explore the possible configuration space for a code snippet, reducing the number of potential environment configurations that need to be validated. When run on a corpus of public Python snippets from prior research, V2 identifies 248 instances of configuration drift.

***Index Terms*—Configuration Management, Configuration Repair, Configuration Drift, Environment Inference, Dependencies.**

I. Introduction

Code snippets are commonly used by developers to provide API documentation [1] and act as examples for learning and reuse [2], [3], [4]. Stack Overflow, a question and answer site for developers, has over 1M questions relating to Python, a popular and fast growing programming language [2]. On GitHub's gist system, developers have shared over 300K public Python code snippets [3]. Both feature prominently in search results for API documentation [5], and a study by Yang et al. found over 4M code blocks from Stack Overflow snippets that had been reused in public GitHub projects [6]. Recently, code snippets in Jupyter notebooks have become a standard for sharing and replicating scientific work and more [4].

Unfortunately, many code snippets require a non-trivial environment configuration in order to execute successfully [7], [8], and are not accompanied by sufficient information for developers to easily recreate that configuration [3], [9], [10]. This leads to the problem of configuration drift, where a code snippet goes out-of-date because the APIs that it depends on experience breaking changes over time. Developers struggle to determine if a code snippet has experienced configuration drift, or is simply incorrect [3]. In some instances, a breaking change made to an API may be highlighted in release documents made available to developers. In the worst case, determining if and when a breaking change was made requires an exhaustive search through previous dependency versions.

The ability to validate and detect out-of-date code snippets is the most important consideration for quality of code reuse according to a recent survey of 183 software developers [11]. Pimental et al. found that only 24% of Jupyter notebooks could be executed, and only 4% had reproducible results [10]. They note that reproducibility suffers because the notebook format does not encode dependencies or dependency versions. For example, the recent release of Tensorflow version 2.0 introduced many breaking changes. As a result, many Jupyter notebooks are not runnable with the latest version of the framework. This proves detrimental, as developers have remarked that documentation for the Tensorflow platform is largely based on examples.[1] Overall, there is an unmet need for checking if code examples are up-to-date and runnable.

This work presents V2, available at **https://github.com/v2-project/v2**, a tool that determines if a code snippet is out-of-date by detecting configuration drift. V2 is based on the observation that an instance of configuration drift often manifests as a crash during execution. It can therefore be categorized by the illuminating failure (crash) paired with a configuration patch sufficient to enable execution. Patches consist of changes to dependency versions, and act both as a certificate of correctness for the snippet and enable execution with an older configuration if desired.

Unlike other work in configuration repair [12], [13], V2 automatically infers up-to-date candidate environment configurations from a code snippet without the need of developer input or pre-existing build scripts. If the code snippet experiences a crash when executed in its candidate environments, V2 searches for configuration drift by employing *feedback-directed search*, a search strategy that incorporates feedback from code snippet execution and knowledge about prior API breakage events to prune and prioritize configuration patches from the space of all possible environment configurations.

We show that V2 is able to discover configuration drift in open source Python code snippets from the Gistable dataset [3] and from a wide array of Jupyter notebooks [4]. For both datasets, we show that incorporating execution feedback and knowledge of previous API breakages enables V2 to more quickly decide when configuration drift is present in a snippet.

[1]https://news.ycombinator.com/item?id=18445225

```python
import math
from functools import wraps

from theano import tensor as T
from theano.sandbox.rng_mrg import
    ↪ MRG_RandomStreams as RandomStreams

from lasagne import init
from lasagne.random import get_rng

...
```

(a) https://gist.github.com/a003ace716c278ab87669f2fbd37727b

```dockerfile
FROM python:3.7
ADD bayes.py /bayes.py
RUN ["pip", "install", "Theano==1.0.4"]
RUN ["pip", "install", "Lasagne=0.1"]
CMD python /bayes.py
```

(b) Dockerfile

```
DecrementSemverMajor(Theano==1.0.4) -> 0.9.0
DecrementSemverMinor(Theano==0.9.0) -> 0.8.2
```

(c) Mutations

Fig. 1: (a) Import statements extracted from a GitHub gist using the Lasagne and Theano APIs. (b) A base environment configuration, using the Docker container system Dockerfile format, containing the gist and the latest versions of Theano and Lasagne. Due to configuration drift, the gist does not execute successfully using the configured versions. (c) Environment mutations made while searching for the correct version of Theano.

II. Motivation

Consider the code snippet fragment presented in Figure 1a. The fragment contains the import statements for Lasagne and Theano from a public Python gist found on GitHub. Both libraries can be found on and installed from the Python Package Index (PyPI). However, after installing version `0.1` and `1.0.4` (the latest versions), respectively, executing the code fragment with Python 3 results in the following exception:

```
...
File "/usr/local/lib/python3.7/site-packages/⌋
    ↪ lasagne/layers/pool.py", line 6, in
    ↪ <module>
  from theano.tensor.signal import downsample
ImportError: cannot import name 'downsample' from
    ↪ 'theano.tensor.signal'
    ↪ (/usr/local/lib/python3.7/site-packages/⌋
    ↪ theano/tensor/signal/__init__.py)
```

The exception indicates Lasagne has attempted to import `downsample` from Theano and failed because the current version of Theano does not provide it in the expected location. The Lasagne installation documentation[2] indicates that the library is tightly coupled with Theano, and that a very recent version of Theano is often required to run correctly:

> Lasagne has a couple of prerequisites that need to be installed first, but it is not very picky about versions. The single exception is Theano: Due to its tight coupling to Theano, you will have to install a recent version of Theano (usually more recent than the latest official release!) fitting the version of Lasagne you choose to install.

However, the current version of Lasagne was released before the current version of Theano. While the documentation indicates that the correct configuration requires installing a pre-release version of Theano, it may actually be the case that Lasagne has drifted and is now out-of-date. The Theano versioning scheme uses semantic versioning[3] (semver), which tells us the current version of Theano is a major version with some patches. We start searching for the correct version by going back to the latest release of the previous major version, `Theano==0.9.0`. When that fails, we go back an additional minor version, `Theano==0.8.2`. Using this version allows the gist to be executed successfully, although with a warning that the "downsample module has been moved to the theano.tensor.signal.pool module," confirming that execution originally failed because the code had fallen out-of-date.

[2]https://lasagne.readthedocs.io/en/latest/user/installation.html
[3]https://semver.org/

III. V2

V2, *"Version 2"*, is an extension of DockerizeMe [9] that adds support for validating multiple environments and reasoning over dependency versions. V2 determines if a code snippet or its dependencies is out-of-date by discovering instances of configuration drift. Each instance of configuration drift is identified by two things: 1) a runtime failure and 2) a configuration patch consisting of version changes.

To detect configuration drift, V2 first generates candidate environments that are potentially correct. That is, the environment is configured with the dependencies that V2 infers as being potentially necessary for executing a snippet correctly. It then executes the snippet in this inferred environment, recording any execution failures as they are encountered. For each new execution failure, V2 applies mutations to installed dependency versions to generate a new candidate configuration until the fault is either resolved, or no new candidate configurations can be made. V2 uses information about snippet execution and prior API breakage events to prune and prioritize candidate configurations. We now detail each stage of the repair algorithm in detail.

A. Candidate Environment Generation

In order to find instances where configuration drift has caused a failure, V2 must first have one or more candidate environment configurations for a code snippet. These should contain the set of required dependencies, although they may not have the correct versions. V2 uses the Docker container system to specify configurations for isolated environments. The use of Docker guarantees a clean and consistent base environment for V2 to work in, and allows each candidate environment to be based off of a standard image.

Although the scheduled end-of-life for Python 2 is 2020 [14], many older scripts will not run in Python 3, as they use syntax that was removed from the language. Candidate generation must first determine which language runtime to use: either Python 2 or Python 3. The correct language runtime is detected by attempting to parse the code snippet using Python's built in AST module for each language runtime (2 or 3). If the code snippet parses in both language runtimes, then two potential candidate environments are generated.

V2 then uses the dependency resolution algorithm and knowledge base provided by DockerizeMe [9] to populate each candidate environment with an initial set of dependencies in the following manner. V2 extracts fully qualified names of resources appearing in import nodes of the AST. Imported resources not in the Python standard library are mapped to a set of packages from the Python Package Index (PyPI) that potentially provide them, although the set may be an overestimate. These packages are considered the direct dependencies of the code snippet. V2 also resolves all transitive dependencies, i.e., packages depended on by a direct dependency or by another transitive dependency. Each dependency is initially pinned at the latest version available on PyPI, because DockerizeMe does not consider versions, and a final installation order is generated such that, for each dependency, it is installed only after all packages that it depends on.

A $(runtime, dependencies)$ tuple defines a candidate environment specification containing the correct language runtime level and dependencies listed in a valid installation order.

B. Environment Validation

The V2 validation phase accepts as input a single code snippet and a candidate environment specification in the $(runtime, dependencies)$ format. It returns a status code indicating the result of executing the code snippet in the candidate environment, plus any metadata about the execution or encountered failures. Validation consists of two phases: 1) environment configuration and 2) snippet execution.

During the configuration step, V2 creates a new execution environment using a Docker container based on the language runtime. This guarantees a consistent starting environment. It then configures the execution environment according to the candidate environment specification by installing each dependency in order. Installation failures are recorded as part of the validation metadata, but do not halt validation. If a dependency which is not required fails to install, it will not affect the result of execution. Conversely, if a required dependency fails to install, it will produce an execution failure.

Finally, the snippet is executed in the configured environment. If the snippet runs to completion without experiencing a failure, the execution result is coded as `Success`. If an exception is encountered during execution, the snippet is considered to have failed validation and the execution result is coded as `Exception`. In this case, the exception name, message, and stack trace are provided in the validation metadata.

In certain cases, a code snippet will run indefinitely instead of exiting. For example, a snippet which waits for user input will block forever. If a snippet has not exited after a time limit, execution is stopped and the validation is coded as `Timeout`. A timeout is considered neither a success nor a failure, as it is generally undecidable if the snippet would have eventually exited. Regular Python snippets are run with a time limit of one minute. Because they generally involve computation and require a longer runtime, Jupyter notebooks are run with a base time limit of two minutes, plus an additional minute for each cell in the notebook.

C. Environment Mutation

Configuration drift is classified by a validation failure and a patch. Each patch consists of a sequence of configuration changes (mutations) to the environment configuration which are sufficient to resolve the fault. V2 supports two classes of mutation operators for dependency versions. The first class is based on the semantic versioning scheme, and the second class is based on pre-existing knowledge about version changes.

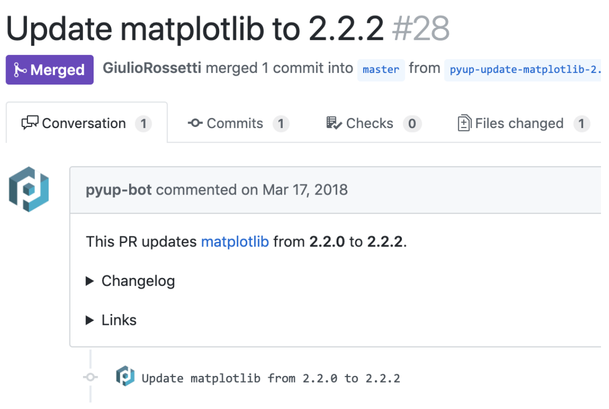
1) Semantic Versioning: Semantic versioning (semver) is a versioning scheme which defines major, minor, patch, and prerelease changes to a package. Breaking API changes must be accompanied by a major version change, while backwards compatible additions are accompanied by a minor version change. However, before version `1.0.0`, semver defines that an API should not be considered stable, and that anything may change at any time. Many Python dependencies follow a versioning scheme that is, or can be interpreted as, semver.[4]

V2 defines two mutation operators for semantic versions: 1) `DecrementSemverMajor` and 2) `DecrementSemverMinor`, which mutate the major and minor version level of a dependency in an environment configuration as seen in Figure 1c. Both mutation operators will decrement the specified version level by one if possible, but will choose the latest release at that version level. For example, if a package has versions `2.1.0`, `1.2.0`, and `1.1.0`, then `DecrementSemverMajor(2.1.0) -> 1.2.0`. `DecrementSemverMinor` only operates within a major version, it will not roll back to a previous minor version if doing so also requires decrementing the major version.

2) Upgrade Matrix: In the worst case, all versions of a dependency may be enumerated by combining the semver mutation operators described in Section III-C1. However, such a brute force approach is generally not feasible due to the large number of potential candidate environments in the version configuration space. A more efficient approach is to prioritize or prune environment configurations based on how likely they are to fix a validation failure.

To make this estimation, we extrapolate from TravisCI build statuses of Python projects that experience version upgrade events. An upgrade event occurs when a commit is made that upgrades the version of a single project dependency, triggering a TravisCI build for the project. Figure 2 shows an upgrade event initiated by `pyup-bot`, an upgrade bot for Python projects.

[4]https://www.python.org/dev/peps/pep-0440/


Update matplotlib to 2.2.2 #28

Merged GiulioRossetti merged 1 commit into `master` from `pyup-update-matplotlib-2.`

pyup-bot commented on Mar 17, 2018

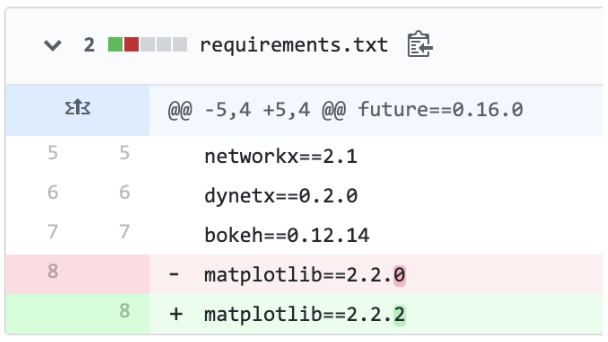
This PR updates matplotlib from 2.2.0 to 2.2.2.

▶ Changelog

▶ Links

`Update matplotlib from 2.2.0 to 2.2.2`

(a) Pull request to upgrade matplotlib from 2.2.0 to 2.2.2.


```
2 requirements.txt
@@ -5,4 +5,4 @@ future==0.16.0
5   5      networkx==2.1
6   6      dynetx==0.2.0
7   7      bokeh==0.12.14
8        - matplotlib==2.2.0
    8    + matplotlib==2.2.2
```

(b) Git diff.

Fig. 2: An upgrade event initiated by the upgrade bot PyUp.

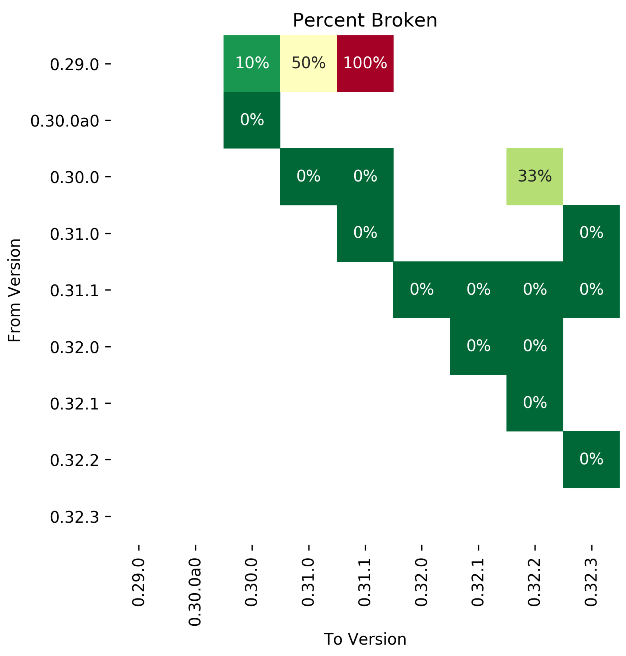


Fig. 3: Version upgrade matrix for Wheel.

All upgrade events were discovered by mining Google Big-Query for GitHub pull requests from January 2014 though January 2019. We limit our search to pull requests which change the version for only a single Python dependency and were submitted by an account with the name `dependabot[bot]`, `dependabot`, `snyk-bot`, or `pyup-bot`. We then extract the project build statuses for the original and new dependency versions from Travis CI, excluding build histories where one of the statuses was either canceled or errored. In total, this provided 7,104 upgrade events for 193 distinct packages that experienced at least one failure.

Taken together, the build statuses for a single package form a version upgrade matrix for that package, where each row and column indicates upgrading from one version to another. Each cell of an upgrade matrix contains the percent of builds that were broken by performing the version upgrade, where a breakage is determined by a build status that changes from passing to failing as a result of the upgrade. Figure 3 shows the version upgrade matrix for the Python package Wheel. An incompatibility cluster in the top row indicates that upgrading from version `0.29.0` becomes more likely to cause a failure as the distance between versions increases.

Intuitively, we apply the heuristic that upgrades which break a larger number of builds are more likely to indicate a backwards incompatible change. Conversely, given a failed validation, downgrading to a version which experienced a large number of upgrade failures is more likely to fix the fault. When V2 finds a validation failure caused by a dependency with an upgrade matrix, it can leverage information about the incompatible upgrades to test only those versions likely to result in a fix, pruning the rest of the version space.

D. Feedback-Directed Search

The problem of finding an instance of configuration drift can be solved by conducting a search through the full configuration space for a code snippet. We model the configuration space as a configuration graph $G = (V, E)$, where each vertex $v \in V$ is a potential environment configuration and each directed edge $(u, v, m) \in E$ is a pair of configurations (u, v) and a mutation m such that applying mutation m to configuration u results in configuration v. We now formally define the problem, DriftSearch, of finding patched configurations in G that fix faults caused by configuration drift. If some configuration fixes all faults, we say that it is a working environment configuration.

DRIFTSEARCH

Input:	A configuration graph $G = (V, E)$ and a starting environment v.
Problem:	Find a working environment w.
Objectives:	Minimize the distance from v to w.
	Minimize the number of vertices explored.

We implement a search strategy, *feedback-directed search*, that generates a configuration space from a candidate environment configuration by applying the mutation operators defined in Section III-C. Feedback-directed search incorporates information about the executability of a code snippet within candidate environments to intelligently drive exploration of the configuration space, reducing the number of environments in the configuration space explored and allowing V2 to quickly converge to a working configuration.

```
Procedure FeedbackDirectedSearch(environment)
    checkpoint = null
    do
        // Validation
        validation = validate(environment)
        if fixed(checkpoint, validation) then
            record_drift(checkpoint)
            checkpoint = validation
            if not fixable(checkpoint) then
                return environment
            end
        end

        // Fault Localization
        if updated(checkpoint) then
            dependency = localize(checkpoint)
            if dependency is not null then
                if has_matrix(dependency) then
                    mutator = matrix(dependency)
                end
                else
                    mutator = IDDFS(dependency)
                end
            end
            else
                mutator = IDDFS(environment)
            end
        end

        // Exploration
        environment = next(environment, mutator)
    while not successful(checkpoint)
    return environment
```

Algorithm 1: Implementation of feedback-directed search for a single candidate environment. Exploration is performed by either using a version upgrade matrix or iterative-deepening depth-first search (IDDFS).

V2 begins feedback-directed search with a list of candidate environment configurations generated as described in Section III-A. If more than one candidate environment was generated, feedback-directed search operates on the environment specifications round-robin, exploring the configuration space for each candidate in tandem. Doing so allows feedback-directed search to satisfy the objective of minimizing the distance, or number of mutations, from starting environment v to working environment w in the case that one candidate environment is easily fixable, but the other is not. The search algorithm is structured in three distinct phases: validation checkpointing and pruning, fault localization, and exploration. A high-level implementation of feedback-directed search for a single environment is outlined in Algorithm 1. We now highlight each phase in detail.

1) Validation Checkpointing and Pruning: Given a code snippet s and candidate environment c, feedback-directed search first performs validation of s in c using the procedure outlined in Section III-B. If the candidate environment produces an execution failure, the validation result is saved as a checkpoint, and c will be mutated to fix the fault. After each environment mutation, s is again validated in c.

Conceptually, at each stage of the search process, a validation checkpoint represents the latest unfixed validation failure. We consider the fault indicated by the validation checkpoint to have been fixed by a sequence of mutations to c if a newer validation results in an execution which covers more of s. That is, the mutations made to c allow execution of s to proceed past the line which previously caused a failure. Whenever such a sequence is discovered by feedback-directed search, it, and the validation checkpoint, are recorded as an instance of configuration drift, and the checkpoint is updated to the newer validation. If a validation indicates that a sequence of mutations has not changed the validation result, or that execution exits on or before the line reached by the checkpoint, it is discarded and the search process continues.

Whenever a new validation checkpoint is discovered, V2 determines if it is potentially fixable via mutations to dependency versions. If it is not potentially fixable, search halts and returns the current environment and instances of configuration drift reported. A validation checkpoint is considered potentially fixable if the execution exception does not indicate a failure due to the local file system and satisfies one of:

1) It is caused by an installed dependency.
2) It is an import error related to an installed dependency.
3) It is one of `TypeError` or `AttributeError`, which can indicate breaking changes in a public API.

2) Fault Localization: During search, V2 prunes the local configuration search space by performing fault localization to map validation checkpoint failures to a single dependency. If the execution failure was caused by code not belonging to the code snippet or the Python standard library, V2 extracts the last dependency from the stack trace that was installed as part of the environment configuration. If the exception is an import error related to a single installed dependency, V2 indicates that dependency. In cases where fault localization fails to highlight a single installed dependency related to the failure, V2 continues exploring the local search space.

3) Exploration: Whenever a validation result indicates that a code snippet s does not execute successfully in a candidate environment c, feedback-directed search searches the local configuration space for a fix by applying mutations to c. There are three exploration strategies, based on the success of fault localization and whether V2 can leverage one or more version upgrade matrices.

If V2 succeeds in localizing a fault to a single installed dependency d, and that dependency has a version upgrade matrix, feedback-directed search proceeds by querying all pairs of versions $(v_{i,1}, v_{i,2})$ such that the version matrix indicates a nonzero percent of builds were broken by upgrading from $v_{i,2}$ to $v_{i,1}$, and $v_{i,1}$ is at most the current version of d. Selection in this manner implicitly disregards the versions which experienced no breaking upgrades. An ordering of versions is generated by sorting all pairs in descending order by $v_{i,1}$. Then, for each $v_{i,1}$, all $v_{j,2}$ not already in the ordering are sorted in descending order by percentage of builds broken and appended to the order. While validation indicates that the checkpoint has not been fixed, d is mutated to be the next version in the ordering and validated again.

When V2 is capable of localizing a fault to a single installed dependency d, but that dependency does not have a version upgrade matrix, feedback-directed search explores the local configuration space by performing iterative-deepening depth-first search (IDDFS) over the versions of d using the semver mutation operators from Section III-C1. In all cases, the operators are strongly ordered such that all instances of `DecrementSemverMajor` are applied prior to instances of `DecrementSemverMinor`. This, coupled with the fact that `DecrementSemverMinor` does not cross major version boundaries, guarantees that the same configuration is not reached at two different depths. If V2 is unable to localize a fault to a single installed dependency, feedback-directed search performs iterative-deepening depth-first search over the versions of all dependencies, making use of upgrade matrices for individual dependencies where applicable. In either case, while performing iterative-deepening depth-first search, candidate environments are only validated when they are generated.

Because of the size of the search space, tracking all candidate environments on the frontier quickly becomes intractable, limiting search algorithms which require doing so. Depth-first exploration allows feedback-directed search to generate and validate new candidate environments with smaller memory requirements. Note that, because mutation operators result in decreasing dependency versions, exploration will never undo mutations made prior to the current checkpoint. Search is halted if no sequence of mutations will lead to an as-of-yet unexplored environment.

IV. Evaluation

We evaluate the ability of V2 to find instances of configuration drift by analyzing open source Python code snippets.

A. Datasets

Horton and Parnin previously presented Gistable, a dataset of 10,259 Python code snippets from GitHub's snippet sharing service [3]. They identified a subset of approximately 2,891 "hard" gists that experienced a failure even after a straightforward approach to inferring an environment configuration. We exclude 795 gists that are known to have over 10 direct dependencies, because the large configuration space is intractable for a baseline. For example, an environment with 10 dependencies, each with 3 versions, would have 3^{10} = nearly 60K possible unique configurations. This leaves 2,096 gists that we refer to as the Gist dataset. Prior work was unable to make 1,999 of these gists execute successfully after candidate environment generation using the latest versions of all dependencies [9].

TABLE I: Summary of code snippets from datasets.

	Gist	**Jupyter**
Total	2,096	6,529
With a Candidate Environment	2,096	5,423
With a Successful Candidate Environment	119	758
No Successful Candidate Environment	1,977	4,665

Second, we collect larger Python code snippets in the form of Python notebooks. Rule et al. previously collected 1.25M open source Jupyter notebooks from GitHub, making the dataset publicly available [4]. As part of their release, they provide a random sample of 6,529 notebooks, many of which are written in Python. We exclude 1,106 notebooks from the sample for which no candidate environment could be generated, leaving 5,423 notebooks we refer to as the Jupyter dataset. There were several main causes for why candidate generation failed, all related to being unable to parse the snippet. Some notebook files were completely empty, or identified as using Jupyter kernels for other languages, such as R, meaning V2 could not parse the notebook source code as Python. Other notebooks contained IPython magics,[5] which are defined as statements which are syntactically invalid Python, causing difficulty in parsing.

For simplicity, and to disambiguate between the origin of code snippets, we refer to code snippets from the Gist dataset as "gists" and code snippets from the Jupyter dataset as "notebooks." Table I shows a summary of both. Together, these snippets represent a wide range of behaviors and API uses, and their generated candidate environments have over 2K unique installable packages.

B. Methodology

We use iterative-deepening depth-first search to establish a baseline for identifying configuration drift in each dataset. This baseline enumerates all states in the configuration space and represents a worst-case scenario where it is assumed a repair can be made by modifying a dependency version, but no information is available to guide configuration. For each code snippet, we record whether search finished or timed out. If search finished, we record whether or not it resulted in a successful environment configuration, and the total number of environments validated.

After running the baseline, we analyze both datasets using V2 with feedback-directed search, recording the same metrics as used in the baseline evaluation. In all cases where search

[5]https://ipython.readthedocs.io/en/stable/interactive/magics.html

finished within the timeout, we also record every instance of configuration drift identified by V2, along with the number of environments in the configuration space that were validated while identifying the instance, as evaluations are the dominating factor in cost for validation based approaches [15]. How far the final configuration is from the starting configuration is also recorded. If search did not find a fully working environment configuration, we record metadata about why the search process terminated, and which portion of the configuration space it was operating in.

We performed evaluation on a cluster of 8 virtual compute nodes running Ubuntu 16.04.5 LTS. Each compute node was configured with 8 CPU cores and 12 GB memory. Evaluations were run using the HashiCorp Nomad job scheduler, with at most 6 jobs running on a single compute node at one time. All evaluations were run with a timeout of 1 hour, after which they were given the chance to gracefully exit and record results. Evaluations which lasted longer than 2 hours were terminated. This could happen for snippets where validating the last environment took longer than an hour, as we allowed each validation to fully complete before timing out.

V. Results

We evaluate V2 with three metrics related to its ability to detect configuration drift: improvement over the baseline, discovery of configuration drift, and search performance.

A. Improvement Over the Baseline

The iterative-depening depth-first search baseline approach searched for a working environment configuration using only the mutation operators `DecrementSemverMajor` and `DecrementSemverMinor`. We found that the most common baseline result was a timeout at the one hour mark without finding a working environment configuration. Search failed to complete for 978 out of 2,096 gists, nearly 50%. 3,921 out of 5,423 notebooks likewise failed to complete within an hour. Comparatively, feedback-directed search only timed out on 248 gists, a reduction of 730 from the baseline analysis. This is mirrored by Jupyter notebooks, only 488 of which timed out versus the baseline of 3,903, a reduction of 3,415.

A simple search strategy is insufficient for fully exploring the configuration space of most code snippets. Feedback-directed search aids exploration by allowing V2 to focus only on a relevant subset of the configuration space.

B. Discovered Configuration Drift

V2 discovered 175 instances of configuration drift present in 143 gists and 73 instances present in 63 notebooks. Search terminated without finding a working environment configuration for 1,882 gists and 5,017 notebooks, only 29 and 24 of which timed out, respectively (although 1,839 notebooks experienced a Jupyter error related to network issues). For 928 gists and 1,710 notebooks, search was terminated because the validation failure was heuristically determined to not be fixable by mutating installed dependency versions according the criteria

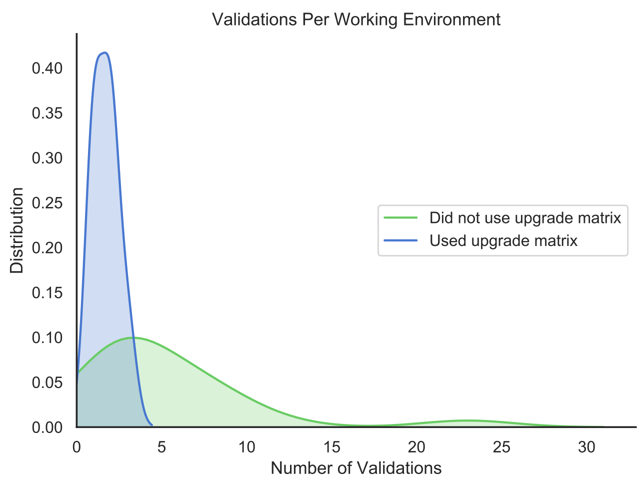


Fig. 4: Number of validations performed on gists for which a working environment was ultimately found. If V2 is able to make use of data in a version upgrade matrix, it can converge to a working environment with fewer validations.

in Section III-D1. In the other cases, search was terminated because V2 had explored all environment configurations in the portion of the configuration space determined to be relevant to the validation failure without finding a patch.

V2 is capable of finding instances of configuration drift in both gists and notebooks. It can potentially find more than one instance in a single snippet.

C. Search Performance

We record two metrics important to search performance, corresponding to the objectives of DriftSearch (Section III-D). The first is the total number of environment configurations validated before finding a patch. This indicates the overall performance of feedback-directed search, since validation dominates runtime performance. The second metric is the number of mutations made to a candidate environment to find a patch. Both metrics reflect on V2's ability to prune and prioritize within the search space.

Feedback-directed search shows an improvement over the baseline in the average number of validations needed to find a working environment configuration. Gists see a large improvement, dropping from 18 validations on average in the baseline to 8 when using feedback-directed search. Jupyter notebooks see a more modest improvement from 7 to 6. For gists, V2 was able to take advantage of a version upgrade matrix for 7 of the 27 (26%) that were made fully executable, bringing the average number of validations needed down from 5 to 2. Even when a working environment was not found, V2 was able to use the version upgrade matrices for about 28% of code snippets, decreasing the number of validations needed before deciding no working environment existed.

Figure 4 shows the distribution of number of validations required by feedback-directed search to find a working environment configuration for gists. When able to prioritize using a version upgrade matrix, V2 requires a third as many validations on average, with substantially better savings in the worst case. Time savings are also seen for individual instances of configuration drift, regardless of whether a fully working environment was eventually discovered.

A relatively small number of mutations (about 2) is required to find a configuration patch for each instance of drift on average. However, incorporating information from a version upgrade matrix results in an improvement in the worst case. For gists, the maximum number of mutations made to create a configuration patch without a version upgrade matrix was 14, while the maximum with one was 7. For notebooks, the maximum number of mutations dropped from 11 to 4. This indicates that V2 is able to correctly skip over certain versions while searching.

Feedback-directed search is able to find configuration drift by only searching a portion of the entire configuration space. Whenever a version upgrade matrix is available, V2 can leverage the build information to prioritize dependency versions and further improve search performance.

D. *Configuration Drift Found in Code Snippets*

We now detail V2's behavior on two code snippets from our corpus, detailing how feedback-directed search actually found and fixed instances of configuration drift.

1) Prioritization of Dependency Versions: Consider the gist `moby.py`,[6] which contains the following imports from the Python library `Sphinx`.

```
from sphinx import addnodes
from sphinx.builders.html import
  ↪  StandaloneHTMLBuilder
from sphinx.util.osutil import EEXIST, make_filename
from sphinx.util.smartypants import
  ↪  sphinx_smarty_pants as ssp
```

Executing `moby.py` with the latest version of Sphinx (`2.0.1`), results in an import error for the module `sphinx_smarty_pants`. A developer might assume that the configuration is broken and look for a missing dependency that provides an extension to Sphinx. However, Sphinx extensions are provided from the `sphinx.ext` module, not the `sphinx.util` module. Another assumption is that a breaking change was made to the `smartypants` module.

To validate this assumption, the developer must find a previous version of Sphinx for which `moby.py` executes successfully. They start by reverting from `Sphinx==2.0.1` to the previous major version `Sphinx==1.8.5`. However, executing with this version results in the same error. They change the major version again to `Sphinx==0.6.7`, a change which also requires using Python 2, but doing so results in an import error for `osutil` on the line before, indicating that the latest version change has actually broken a part of the configuration which previously worked.

If a correct working version exists, it must be one of the other untested minor or patch level versions. Without any other information, discovering this version requires performing an exhaustive search though the `1.x` and `0.x` versions. Using a depth-first strategy, such as iterative-deepening depth-first search, the developer might additionally validate versions `1.7.9`, `0.5.2`, `1.6.7`, and `0.4.3` before finding a working configuration with version `1.5.6`.

V2, by comparison, starts by validating Sphinx version `2.0.1` and encounters the original error, localizing it to the Sphinx module. Using the available upgrade matrix, it sees that the most recent version with upgrades that resulted in a broken build was `1.7.5`, and the only broken upgrade was from version `1.4.5`. It then mutates the environment, resulting in a configuration in which the gist successfully validates.

2) Uncovering Multiple Instances of Configuration Drift: V2 is capable of uncovering more than one instance of configuration drift from the code snippets that it analyzes. For example, by using feedback-directed search, V2 demonstrates two instances of configuration drift in the gist `guess_candidate_model.py`,[7] which the following excerpt is from.

```
from sklearn.cross_validation import train_test_split
from keras.preprocessing import sequence, text
from keras.models import Sequential
from keras.layers import (Dense, Dropout, Activation,
  ↪  Embedding, LSTM, Convolution1D, MaxPooling1D)
...
X = [x[1] for x in labeled_sample]
y = [x[0] for x in labeled_sample]
```

This gist relies on two installable packages: `scikit-learn` and `Keras`. V2 correctly generates candidate environment configuration for both Python 2 and Python 3 containing the latest versions of both of these packages.

The first validation inside of a candidate environment results in a failure, returning the message "`No module named 'sklearn.cross_validation'`" even though the `sci-kit` package is included in the environment configuration. V2 recognizes this as a potential instance of configuration drift and applies the mutation operator `DecrementSemverMinor(scikit-learn==0.20.3) -> 0.19.2`.

The next validation also results in a failure, but this time with the message "`No module named 'tensorflow.'`" V2 determines that the previous failure has been fixed, because execution progressed past the line which caused the previous failure, and recognizes this as a potential instance of configuration drift involving `Keras`. In searching for a patch, it applies the mutation operators `DecrementSemverMajor(Keras==2.2.4) -> 1.2.2` and `DecrementSemverMajor(Keras==1.2.2) -> 0.3.3`.

Finally, V2 encounters the error "`name 'labeled_sample' is not defined`." It again recognizes that the previous failure has been fixed. Further, it recognizes that this failure indicates

[6]https://gist.github.com/5866756

[7]https://gist.github.com/eba2bf1a0ecd3e541146e98f35d49739

an error with the gist that cannot be fixed by modifying the environment configuration and terminates search.

VI. Limitations

V2 presents a promising approach to finding and repairing instances of configuration drift within a code snippet and its dependencies. We show that it is capable of finding and repairing drift in a large corpus of open source Python code snippets. There are, however, some limiting factors which make it challenging to uncover all instances of drift.

a) Silent Failures: Feedback-directed search requires that configuration drift result in a crash during snippet execution, because this failure is used to guide the search process and determine when the configuration drift has been patched. This requirement is sound for a large number of breaking API changes, such as removal of modules or a change in the code signature, because such changes result in runtime or compile time exceptions. However, some breaking changes in dependency behavior may cause a failure which does not manifest as a crash. V2 cannot address such cases, although we note that the problem of determining if program behavior is expected is an open and challenging area of research [16].

b) Mutation Operators and Ordering: V2 concentrates on major and minor dependency versions because they are the most likely to introduce or change behavior of an API, and its search strategy explores versions in the configuration space with equal precedence, only giving preference to configurations that are more similar to the original candidate configuration. However, developers differ in how they reason about the stability of dependencies, and communities differ on how they handle versioning [17]. Other search strategies may ultimately prove to be more effective at uncovering configuration drift.

c) Overzealous Pruning: Parts of the V2 search algorithm are informed by heuristics. In cases where a heuristic erroneously produces a false positive or false negative determination, search may be terminated prematurely. For example, if a validation failure is heuristically determined to not be fixable by modifying dependency versions, but a fix actually does exist, V2 will stop exploration and be unable to report the configuration drift.

For some dependencies, search is informed by a version upgrade matrix obtained from TravisCI build results. However, an upgrade matrix may be incomplete or not sufficiently represent some breaking upgrades, causing incorrect pruning.

d) Snippet Properties: Some code snippets may be unsuitable for uncovering configuration drift. Gists can be overly general, relying on other services and requiring resources like authentication tokens [3]. Failures related to these resources may shadow configuration drift that could otherwise be discovered. Notebooks are heavily used within the data science community [4], and data science dependencies may be more stable than Python dependencies in general. Additionally, gists were restricted to having at most 10 direct dependencies. While this does not necessarily restrict the total number of dependencies, it may impact the distribution of results.

e) Discovery: Because exploration of the configuration space is guided by feedback from validation, V2 discovers at most one instance of configuration drift at a time. Further, validation can be an expensive process, and other instances of drift may be hidden by the most recent validation failure.

VII. Discussion and Future Work

Recent work has shown that managing dependencies is a large and time consuming problem developers face when engaging in configuration management [18], [8], [19], [3], [9]. In particular, some developers have highlighted dependency management as one of the largest problems facing Python, requiring developers to spend non-trivial amounts of time on configuration for basic tasks.[8] This problem impacts data scientists, who must overcome the friction imposed by configuration just to perform their job.[9]

In addition, upgrading a version of a library that contains incompatibilities in data structures, signature changes to API calls, or behavior breaking changes [20] can result in additional effort to address these changes. If a developer fails to understand the nature of a dependency change or performs insufficient testing, they can introduce undetected faults in code. Such factors may cause a developer to become reluctant to update dependencies. But, if they delay updating code too long, they may be locked out of being able to use important new features only available in new versions, as it becomes more and more difficult to adapt their code. For this reason, detecting instances of configuration drift has direct implications for configuration management. Being able to highlight where code contains outdated API usages, even between dependencies of that code, can help developers locate and fix usage of APIs that have been changed, benefiting work in API migration. Further, by pairing each instance of drift with a corresponding fix, we motivate code repair [21], and reuse. For example, repairing Jupyter notebooks can enable replicating important calculations even in older notebooks.

Although we focus on Python, we believe detection of configuration drift is an important problem for other languages. For example, updating the MongoDB driver for Node.js from `2.0.x` to `2.1.x` affected the way order by parameters are used in the *sort* function. Previously, parameters were allowed to be specified as list of objects: `[{publish_date: -1}]` but in the newer version, list of lists must be provided: `[["publish_date", -1]]`. V2 is capable of detecting such a breaking change in any language so long as the code snippet may be parsed and validated, and previous versions of dependencies are made available.

While the technique outlined in this paper shows promise, we believe there are several areas for improvement.

a) Fuzzing: Data in version upgrade matrices is necessarily limited, as it originates from upgrades to existing open source projects which both use upgrade bots and build through TravisCI. When data does exist for a package, the matrix

[8]https://twitter.com/jeanqasaur/status/1104990612057518081
[9]https://twitter.com/LibSkrat/status/1122857944675229698

is usually sparse. We believe that the amount and quality of available upgrade data can be augmented. One method would be to engage in fuzzing of dependency versions for libraries. For example, the upgrade matrix for numpy could be filled in by choosing a set of snippets or generated tests that cover the API and observing their execution with different versions of the library. This additional execution data could help determine clusters of incompatible versions that indicate a breaking change for particular API uses.

Traditional fuzzing techniques can also be implemented to improve drift detection. Some code snippets, particularly those meant to be used as examples, define functions but do not execute them. Executing these functions with generated input would ensure greater coverage of the code snippet and its API uses, allowing further detection of instances where configuration has drifted.

b) Progression in the Face of Adversity: V2 currently only recognizes configuration drift caused by versions of installed dependencies. However, code snippets often depend on resources external to the configuration environment, such as communication with databases or REST APIs. When validation of a code snippet fails due to an external service, future work can focus on determining if the failure represents an instance of configuration drift and attempt to generate a fix. If a fix cannot be found, a "mock" patch sufficient to remove the validation failure could be inserted, allowing analysis of the snippet to continue. Missing resources, such as files on the local file system, could also be synthesized to allow code snippet analysis to continue.

c) Improvements to Search: Feedback-directed search uses heuristics and checkpointing to reduce the search space. This results in improved performance, but at the potential cost of completeness. This could be mitigated by still searching all versions, but using version matrices for prioritization. Additionally, the checkpointing process can be modified to allow for backtracking to a previous checkpoint if no solution is found (as might be the case in library version conflicts). However, V2 does not currently do so.

Finally, work can be done to improve the quality of search through the environment space. V2 assumes that the original candidate environment is complete and contains all required dependencies, if not the correct versions. We leave it to future work how best to continue searching if it is discovered that this is not the case.

d) Optimization and Improvement: The validation strategy employed by V2 currently starts by creating and configuring a new environment. This is sufficient to guarantee correctness of validation, but requires an expensive configuration phase after every mutation performed by feedback-directed search. We note that, because feedback-directed search generates a new environment configuration by applying a single mutation to the previous configuration, V2 could optimize the validation process by creating and continually mutating a single environment specification. This would greatly reduce the cost of validation.

VIII. Related Work

Previous work has concentrated on evaluating the executability of code in the context of its configuration. Sulír and Porubän find 38% of Java builds fail due to dependency related errors [8]. McIntosh et al. agree that build maintenance imposes a large overhead on developers [22]. Yang et al. [7] evaluate Python code snippets, as do Horton and Parnin [3]. They find that Python code snippets are not executable by default due to configuration errors, and Horton and Parnin additionally highlight the difficulty of correct configuration, of which inferring correct dependency versions ranks highly.

Additional studies have focused specifically on how developers manage dependencies and versions within environment configuration. Xavier, Hora, and Valente note that developers deliberately make breaking changes to APIs for several reasons related to code maintenance [23]. Further, developers struggle with different options for specifying dependency versions that are supported by many package management systems [24], [25], and analysis of such systems motivates the need for better tooling to deal with problems related to versions [26].

Although these works demonstrate that configuration of dependencies is a frequent cause of build and execution failure, and that developers have need of better tooling support of versions [27], [28], [29], [30], no study attempts to uncover instances of configuration drift automatically. Horton and Parnin do classify 15 code snippets which have drifted and only execute successfully with an older version of a dependency (Table II from Gistable [3]), but this classification was performed by manually generating configuration scripts.

Existing work has been performed on the automatic repair of environment configurations. Weiss et al. presented Tortoise, a system which synthesizes patches for configuration scripts, but can only do so by analyzing recorded developer actions [31]. Horton and Parnin work on inferring working environment configurations without the need of developer interaction, but they crucially ignore dependency versions [9].

Work by Macho et al. [12] and by Hassan and Wang [13] has presented automated repair of configuration scripts that includes version modifications. These are the closest related works of which we are aware. Crucially, both techniques require an existing environment configuration, and neither approach uses feedback to guide search. In addition, they focus on the Java build ecosystem, and cannot repair Python code snippets. Macho et al. prioritize versions using a distance metric that prefers small version changes. Hassan and Wang generate a ranked list of all possible patches and exhaustively apply them until some patch succeeds or there are no more patches to apply. Neither approach attempts to prune the configuration space, and prioritization is limited. There are approaches that use feedback [32] and historical knowledge [33], but they focus on code, not computing environments.

This work presents an improvement over previous approaches by incorporating feedback and prior knowledge for more efficient identification of repairs. We also focus on identifying instances of configuration drift in code, determining when dependencies have been updated, and synthesizing a patch as a proof of existence. This is in contrast to other approaches, which focus on an existing environment configuration and assume failures result from the code being updated.

IX. Conclusion

Motivated by the problem of detecting out-of-date code snippets, we implement V2. V2 discovers instances of configuration drift in Python code snippets, where each instance is identified by looking for crashes that can be fixed with modifications to versions of installed dependencies.

Two techniques lie at the heart of V2's ability to uncover configuration drift. The first is feedback-directed search, a search strategy that uses information gleaned from executing a code snippet to determine which portion of the configuration space to explore next. The second is a corpus of version upgrade matrices that describe the before and after build statuses for over seven thousand package upgrade events in open source Python projects. These version upgrade matrices allow V2 to significantly prune the configuration space, and effectively prioritize the remaining states, leading to faster discovery of working configurations.

V2 found 248 instances of configuration drift in the Gist and Jupyter datasets. It was able to do so quickly, requiring many fewer validations than a baseline prioritization strategy.

Acknowledgements

This work is funded in part by the NSF SHF grant #1814798.

References

[1] C. Parnin, C. Treude, and M. A. Storey, "Blogging developer knowledge: Motivations, challenges, and future directions," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 211–214.

[2] W. Wang, G. Poo-Caamaño, E. Wilde, and D. M. German, "What is the gist?: Understanding the use of public gists on github," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 314–323. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820556

[3] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[4] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 32:1–32:12. [Online]. Available: http://doi.acm.org/10.1145/3173574.3173606

[5] C. Treude and M. Aniche, "Where does google find api documentation?" in *IEEE/ACM 2nd International Workshop on API Usage and Evolution*, ser. WAPI'18. New York, NY, USA: ACM, 2018.

[6] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: Any snippets there?" in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 280–290. [Online]. Available: https://doi.org/10.1109/MSR.2017.13

[7] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: An analysis of stack overflow code snippets," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 391–402. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901767

[8] M. Sulír and J. Porubän, "A quantitative study of java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2016. New York, NY, USA: ACM, 2016, pp. 17–25. [Online]. Available: http://doi.acm.org/10.1145/3001878.3001882

[9] E. Horton and C. Parnin, "Dockerizeme: Automatic inference of environment dependencies for python code snippets," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019.

[10] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19, 2019.

[11] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, pp. 1–37, 2018.

[12] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 106–117.

[13] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 2018 ACM/IEEE 40th International Conference on Software Engineering*, ser. ICSE 2018, 2018.

[14] B. Peterson, "Pep 373 – python 2.7 release schedule," https://www.python.org/dev/peps/pep-0373/, online; Accessed May 7, 2019.

[15] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 356–366.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[17] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, Nov 2015, pp. 86–89.

[18] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)," in *International Conference on Software Engineering (ICSE)*, 2014.

[19] L. S. M. M. Simon Urli, Zhongxing Yu, "How to design a program repair bot? insights from the repairnator project," in *40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP)*, ser. ICSE 2018, 2018, pp. 1–10. [Online]. Available: https://hal.inria.fr/hal-01691496/document

[20] E. Ruiz, S. Mostafa, and X. Wang, "Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries," Department of Computer Science, University of Texas at San Antonio, Tech. Rep., 2015. [Online]. Available: http://xywang.100871.net/TechReport_EmpIncomp.pdf

[21] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1173706.1173734

[22] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985813

[23] L. Xavier, A. Hora, and M. T. Valente, "Why do we break apis? first answers from developers," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 392–396.

[24] J. Dietrich, D. J Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," 03 2019.

[25] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 323–333.

[26] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 2–12.

[27] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway,

NJ, USA: IEEE Press, 2017, pp. 84–94. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155577

[28] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, Dec 2007.

[29] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062512

[30] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000805

[31] A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive system configuration repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 625–636. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155641

[32] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, vol. PP, 01 2018.

[33] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 213–224.