How PHP Releases Are Adopted in the Wild?

Jukka Ruohonen
University of Turku, Finland
Email: juanruo@utu.fi

Ville Leppänen
University of Turku, Finland
Email: ville.leppanen@utu.fi

***Abstract*—This empirical paper examines the adoption of PHP releases in the the contemporary world wide web. Motivated by continuous software engineering practices and software traceability improvements for release engineering, the empirical analysis is based on big data collected by web crawling. According to the empirical results based on discrete time-homogeneous Markov chain (DTMC) analysis, (i) adoption of PHP releases has been relatively uniform across the domains observed, (ii) which tend to also adopt either old or new PHP releases relatively infrequently. Although there are outliers, (iii) downgrading of PHP releases is generally rare. To some extent, (iv) the results vary between the recent history from 2016 to early 2017 and the long-run evolution in the 2010s. In addition to these empirical results, the paper contributes to the software evolution and release engineering research traditions by elaborating the applied use of DTMCs for systematic empirical tracing of online software deployments.**

***Index Terms*—release engineering, software evolution, continuous delivery, patching, upgrading, downgrading, web crawling**

I. Introduction

Programming languages evolve like any other software [1]. Like most software, also programming languages require release engineering, and as with conventional software, users of a programming language are likely to abandon the language if it is not properly updated and maintained to meet the continuously changing requirements [2]. In recent years, different continuous software engineering practices have become increasingly popular for the development and maintenance of conventional software. Interestingly, also programming languages such as PHP have adopted a strategy of continuous releases scheduled to occur in a fast and fixed release cycle.

This paper investigates the continuous release engineering of the PHP programming language from a perspective of deployments using the language to serve some of the most popular web sites in the current Internet. While the release engineering practices used by the PHP project establish the practical motivation, the primary scholarly purpose of the investigation is to examine the previously unexplored use of classical DTMCs for studying release engineering. For putting the elaborated DTMCs into work, large web crawling datasets are used to analyze the current PHP release adoption patterns.

Markov chains belong to the classical methodology toolbox in reliability engineering [3], [4]. Discrete time-homogeneous Markov chains have recently been also adopted for studying different empirical software engineering problems, including those related to software evolution [5]. Thus far, however, empirical applications have been limited in the release engineering domain. In addition to patching this limitation in the literature, this paper contributes to the release-based approaches (as opposed to approaches based on version control and bug tracking systems) for studying software evolution.

There exists a large amount of empirical work using a release-oriented perspective to study software evolution in general and release engineering in particular. Backward-compatibility [6], library dependencies [7], and so-called rapid releases [8], [9], [10] are good examples of recent questions examined (for a review of current research challenges see [11]). Many of these studies deal with upgrading and downgrading question either explicitly or implicitly. There is thus plenty of prior work for framing the questions examined.

However, much of the existing empirical work has literally been release-based, whereas this paper leans towards a deployment-based approach. In other words, much of the prior work is on the producer-side, while the consumer-side has received less attention [12]. While bringing these two sides closer to each other remains a major research challenge, it is worthwhile to further note that the release-versus-deployment distinction applies also to studies examining the evolution of programming languages. In particular, there is some prior work examining PHP deployments seen in the wild [13], but the evolution of the programming language itself—with its features and flaws—has received more attention [14], [15]. The same applies to PHP source code analysis, which has mostly concentrated on evaluating “off-the-shelf” PHP applications (e.g., [16], [17]) without attempting to cover custom applications seen in the wild. Although source code analysis is not pursued in this paper, the deployments examined still cover many custom PHP applications. Due to such applications, better knowledge about adoption and patching on the consumer-side is valuable for those on the producer-side. In other words, it is important to consider continuous tracing of software deployments in order to improve the feedback loops required for sound continuous software engineering practices.

The remainder of the paper is structured into four sections. Section II motivates the research background in more detail and formulates the research questions for the empirical analysis. Section III outlines the DTMC modeling approach. Results are presented in Section IV based on large longitudinal datasets compiled from a few third-party web crawling snapshots. Conclusions and discussion follow in Section V.

II. Background

The scholarly background can be motivated by considering the feedback channels that are essential for the contemporary continuous software engineering practices. After connecting these practices to the concept of software traceability, research questions are formulated in relation to the current release engineering strategy of the PHP programming language.

A. Motivation

Continuous software engineering is an umbrella term covering multiple contemporary software engineering tools and methodologies, including but not limited to continuous planning, continuous budgeting, continuous integration, continuous delivery, continuous deployment, continuous testing, continuous evolution, continuous maintenance, continuous feedback, and, ultimately, continuous innovation [18], [19]. These overlapping continuous-prefixed concepts are also well-recognized in the release engineering research domain.

However, much of the existing research has concentrated on traditional software engineering aspects, such as integration, build systems, testing, and maintenance. This emphasis is reflected in the attempts to define the concept of release engineering. For instance, release engineering has been defined as "a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high-quality software reliably and predictably" [20]. Although the word improvement appears in the definition, a little emphasis is placed on feedback from customers, investors, and other stakeholders, which is a fundamental element in the contemporary continuous software engineering practices and processes [8], [21]. By and large, these feedback mechanisms have constituted an enduring challenge for empirical software evolution research in general [22], [23]. This gap in the literature is noteworthy because the availability of information about releases has increased substantially in recent years.

Different "telemetry" solutions—including crash reports and other "call-home" features—are increasingly popular in many software industry segments. The availability of feedback data is not limited to features explicitly integrated into software, however. Social media, review and rating sites, and related elements of the contemporary world wide web provide a wealth of information for systematic tracing of releases. A major challenge for modern release engineering relates to integration of such data into meaningful solutions that help developers and stakeholders to make informed decisions about the evolution and patching of software deployments [11]. For summarizing this key challenge, Fig. 1 depicts a relational map of a few interconnected continuous software engineering concepts. In this paper, the focus is at the lower-right corner, which is labeled as continuous tracing of the continuously engineered releases that are continuously deployed in the wild.

The concept of continuous tracing can be linked to software traceability, which "refers to the ability to describe and follow the life of a requirement", release, or other software artifact "in both a forwards and backwards direction" [24]. This definition can be used to clarify and frame the scope of this paper.

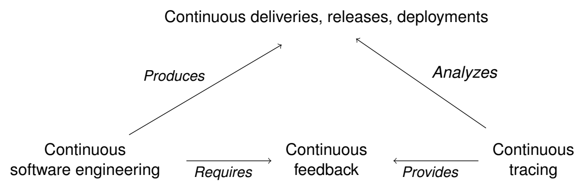
Continuous deliveries, releases, deployments
Produces
Analyzes
Continuous software engineering
Requires
Continuous feedback
Provides
Continuous tracing

Fig. 1. A Terminological Map

By appending the word continuous to the traceability term, it is emphasized that tracing of software artifacts should be systematic and continuous throughout the life cycles of the artifacts. In this paper, the software engineering artifacts are releases of the PHP programming language, but the units of analysis are deployments using the releases for serving web pages. The tracing in forward and backward directions is done by observing upgrading (i.e., roll-forward), downgrading (i.e., roll-back or reverting), and release adoption (i.e., either upgrading or downgrading) patterns of PHP deployments.

B. PHP Releases and Research Questions

The first version of the PHP programming language was announced in 1995. The second, third, fourth, and fifth major versions followed in 1997, 1998, 2000, 2004, respectively. The sixth major version was branched for development in 2010. Instead of evolving into a production-ready major release branch, the controversies regarding Unicode support resulted in backporting of features from PHP 6 to the fifth major branch [25]. Currently, most PHP deployments still run with PHP 5, while the head of development occurs in the PHP 7 branch, which is not compatible with the previous major branches due to numerous new language features. For the programming language developers involved in the project, it is relevant to know an answer to the following question (RQ_1).

RQ_1 *How widespread has the transition been to PHP 7?*

The PHP project follows the semantic versioning strategy conveyed via the "*major.minor.maintenance*" versioning scheme. According to this versioning strategy, in essence, a major release should be reserved for incompatible changes to the application programming interfaces (APIs); a minor release should aggregate functionality enhancements that are backward-compatible; and, finally, maintenance releases should be reserved for small backward-compatible bug fixes and reliability improvements [6], [7]. These versioning principles have also guided the PHP release process since 2010 when a fixed release cycle was agreed upon. According to the current strategy, minor releases are scheduled to occur annually, whereas maintenance versions are released at least once a month [26]. Backward-compatibility and API stability are guaranteed within major branches. At the time of writing, the 5.6, 7.0, and 7.1 minor branches are still supported [27], which is in accordance with the guarantee of three years of support (bug and security fixes) for each minor release.

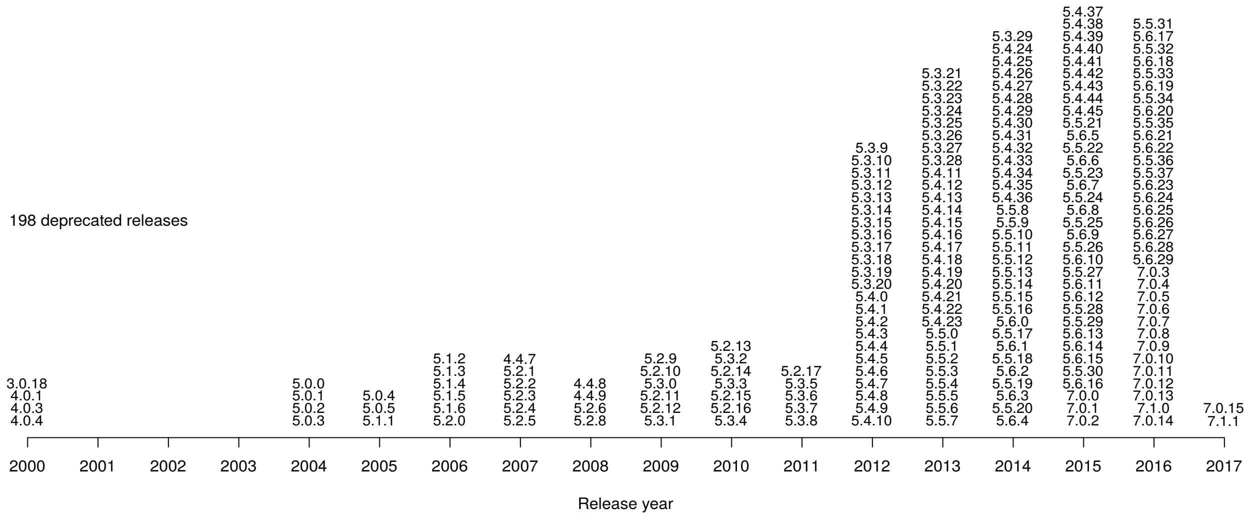

Fig. 2. Documented PHP Release History (unsupported versions as of March 2017, parsed from [28])

For a programming language project, the monthly cycle for maintenance releases is extremely rapid. On paper, this cycle is actually faster than those used for the development of many web browsers, such as Firefox [9]. As is visible from the illustration in Fig. 2, the strategy of monthly maintenance releases has also resulted a large amount of versions from circa 2012 onward. Given the rapid release cycle and the large amount of releases made in recent years, it is relevant to solicit an answer to the research question RQ_2.

RQ_2 *How prevalent has the adoption of PHP releases been?*

The question about release adoption includes both upgrading and downgrading of PHP versions. From a release engineering perspective, particularly interesting are cases whereby a deployment downgrades its PHP version. For instance: if a web site used a version `5.5.0` at some point in time but then later adopted a version `5.4.0`, perhaps there were difficulties in adopting the new release. If such downgrading is common, it might be worthwhile to revisit the supportive activities [29] associated with release engineering. Actual bug fixes notwithstanding, these activities include sufficient release notes, good and up-to-date documentation, user support, easy installation procedures, pre-install checks, sane defaults, migration instructions, and related release engineering aspects. Given this reasoning, the question (RQ_3) is worth asking.

RQ_3 *How common is downgrading of PHP deployments?*

At a more abstract level of thought, it is relevant to know how consistent or uniform release adoption has generally been in recent history. By uniformity, it is meant that most deployments follow the semantic versioning strategy in their upgrades, moving within a major or minor branch in a relatively logical manner. When planning for new releases or supportive activities thereto, it is less relevant to try to support deployments that adopt releases in a chaotic manner. For instance: if a site upgraded from `5.4.0` to `7.1.1` but then moved to `5.5.0` while using a version `5.3.1` in-between, there are likely problems in the maintenance of the site, which cannot be addressed by the means of release engineering. If such chaotic patterns are widespread, on the other hand, it may be relevant to reconsider the appropriateness of a release strategy. Thus, the following question (RQ_4) is justified.

RQ_4 *How uniform has the adoption PHP releases been?*

Finally, the following atheoretical assertion can be placed for controlling the answers to the research questions outlined.

RQ_5 *Do the answer to RQ_1, RQ_2, RQ_3, and RQ_4 vary between the recent short-run history and the long-run evolution?*

The concepts of prevalence, uniformity, short-run, and long-run are further elaborated in the subsequent sections that introduce the Markov chain framework and the empirical data.

III. Approach

A few remarks about the fundamental properties of DTMCs are required to outline the research approach. After these remarks, computation and operationalization are discussed.

A. DTMCs in Brief

A first-order discrete time Markov chain is a finite sequence of random variables $X_1, X_2, \ldots, X_t, \ldots$, satisfying the fundamental Markov property according to which the probability distribution of a forthcoming X_{t+1} depends on the immediately preceding X_t but not on $X_{t-1}, X_{t-2}, \ldots, X_1$. If $S = \{s_1, \ldots, s_n\}$ denotes a set of all possible values of the random variables, the Markov property implies that the probability of moving to a next state in the state space S is

$$\begin{aligned}\Pr(X_{t+1} = s_{t+1} \mid X_1 = s_1, X_2 = s_2, \ldots, X_t = s_t) \qquad (1)\\ = \Pr(X_{t+1} = s_{t+1} \mid X_t = s_t).\end{aligned}$$

This first-order Markov property implies a "memoryless" model, meaning that predicting a future state depends only on the current state. In addition to (a) assuming that (1) holds, (b) the chains observed are assumed to be time-homogeneous. The latter condition means that a transition probability

$$p_{ij} = \Pr(X_{t+1} = s_j \mid X_t = s_i) \tag{2}$$

from a state $s_i \in S$ to state $s_j \in S$ is independent from t,

$$\Pr(X_{t+1} = s_j \mid X_t = s_i) = \Pr(X_t = s_j \mid X_{t-1} = s_i). \tag{3}$$

In other words, the transition probabilities do not change as time passes. This assumption can be further accompanied by emphasizing that (c) only discrete chains are considered without explicit linkage to continuous calendar-time. This further restriction implies that the transition probability in (2) does not depend on the calendar-time lag between s_j and s_i, irrespective whether the lag is measured in months or years.

Finally, (d) the state changes associated with two distinct (exogenous) sequences, $X_1, X_2, \dots,$ and $Y_1, Y_2, \dots$, are assumed to be independent from each other, such that

$$\begin{aligned}&\Pr(X_{t+1} = s_j \mid X_t = s_i, Y_1 = s_1, \dots, Y_t = s_k), \\ &\quad = \Pr(X_{t+1} = s_j \mid X_t = s_i).\end{aligned} \tag{4}$$

In other words, the cross-sectional empirical analysis is conducted without considering any potential dependencies between individual sequences and their state changes.

B. Computation

The empirical setup is based on a sample of m domains:

$$\begin{array}{c} X_1^{(1)}, X_2^{(1)}, \dots, X_{r_1}^{(1)} \\ \vdots \\ X_1^{(m)}, X_2^{(m)}, \dots, X_{r_m}^{(m)} \end{array} \tag{5}$$

that are exogenous with respect to each other, such that (4) holds for any pair of sequences and their state changes.

Due to practical reasons stemming from data collection, the length of the sequences and state changes are both allowed to vary across domains. For instance the length of the sequence for the k:th domain, denoted by r_k, may differ from another sequence length r_{k+1}. These varying sequence lengths correspond with the times each domain is observed empirically.

As described later in Section IV-A1, the maximum sequence lengths are 14 and 6 for all domains in the short-run and long-run examinations, respectively. In addition, a constraint $r_k \geq 2$ is imposed for all m domains to ensure that state changes are possible to begin with. Even when the k:th domain is observed fourteen times, however, the length of the state space may equal one in case the domain in question never changed the PHP version of its deployment. In contrast, the maximum value r_k for $|S_k|$ is attained by a domain that has changed its PHP version each time the domain is observed.

The transition probabilities are estimated by

$$\hat{p}_{ij}^{(k)} = \begin{cases} 0 & \text{if } f_{i.}^{(k)} = 0, \\ f_{ij}^{(k)} \;/\; f_{i.}^{(k)} & \text{if } f_{i.}^{(k)} \neq 0, \end{cases} \tag{6}$$

where $f_{ij}^{(k)}$ denotes the frequency of $(X_t = s_i, X_{t+1} = s_j)$ PHP version sequences for the k:th domain and

$$f_{i.}^{(k)} = \sum_{j=1}^{|S_k|} f_{ij}^{(k)}. \tag{7}$$

The special case $f_{i.}^{(k)} = 0$ occurs when the last observed state denotes a previously unseen PHP version, meaning that there is not enough data to estimate the transition probability for this state. This additional, context-specific alteration notwithstanding, the equation (6) conveys a conventional maximum likelihood estimator (MLE) for a transition probability from the i:th to the j:th state [30], [31]. While a small custom implementation is used for the MLE computations, the results were further verified with an existing R implementation [32].

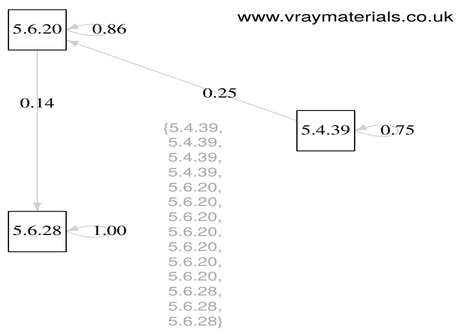

Fig. 3. An Example DTMC for PHP Release Adoption

To illustrate the computation in practice, consider the example in Fig. 3. A full sequence of fourteen observations is available for this domain, while the state space contains three unique PHP versions. The first state corresponds with the PHP version `5.4.39`. Because the domain used the same deployment also during the three subsequent observations, the probability of upgrading from this version to `5.6.20` was $1/4 = 0.25$. The probability of subsequently upgrading the deployment from `5.6.20` to `5.6.28` is even lower, given the seven times the domain `www.vraymaterials.co.uk` stayed with its `5.6.20` deployment. Thus, the prevalence of release adoption has been modest for this particular domain in the short-run. For evaluating the prevalence among hundreds of thousands of web sites, a few custom metrics can be derived.

C. Metrics

The research question about prevalence (RQ$_2$) can be answered with a metric based on the estimated transition probabilities. Thus, let $\mathbf{P}_k$ denote a $|S_k| \times |S_k|$ matrix of estimated transition probabilities for the k:th domain. For instance, the 3×3 transition probability matrix underneath the illustration in Fig. 3 is defined by

$$\underbrace{\begin{bmatrix} 0.75 & 0.25 & 0.00 \\ 0.00 & 0.86 & 0.14 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}}_{\texttt{5.4.39 5.6.20 5.6.28}} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \begin{matrix} \texttt{5.4.39,} \\ \texttt{5.6.20,} \\ \texttt{5.6.28,} \end{matrix} \tag{8}$$

which can be read as an adjacency matrix for a weighted and directed graph. The trace of this matrix (that is, the sum of the diagonal elements) provides a simple measure for the persistence of a DTMC phenomenon [31]. For answering to the first research question RQ_2, this simple but powerful idea allows to operationalize the concept of prevalence with

$$\delta_k = \frac{1}{|S_k|} \sum_{i=1}^{|S_k|} \left(1 - \hat{p}_{ii}^{(k)}\right), \tag{9}$$

where $\delta_k \in [0, 1]$ for all k. In other words, the closer a δ_k is to unity, the more prevalent has the adoption of releases been. If $\delta_k = 0$, the k:th web site never changed its PHP deployment. Collecting the scalars to a vector $\boldsymbol{\delta} = [\delta_1, \delta_2, \ldots, \delta_m]$ allows evaluating the prevalence among the m domains observed.

Answering to the research questions RQ_3 and RQ_4 is better done with the version sequences in (5) rather than with the transition probabilities within the state spaces. Thus, for evaluating how uniform PHP release adoption has generally been among the m domains observed (RQ_4), a simple metric is available by counting the unique version sequences, scaling the resulting amount by m. Although this metric approaches zero as $m \to \infty$, it still gives a good overall sense about the uniformity of typical PHP release adoption patterns.

Although calendar-time records can be used for comparing release orderings [7], a metric for downgrading (RQ_3) can be also computed directly from the PHP version sequences. For all domains with $|S_k| \geq 2$, downgrading can occur via three different scenarios: (a) when $\textit{major}_{i+1} < \textit{major}_i$, that is, when the major version number of a current deployment is larger than the major version number of a subsequent deployment; (b) when $\textit{major}_{i+1} = \textit{major}_i$ but $\textit{minor}_{i+1} < \textit{minor}_i$; or (c) when both the major and minor version numbers remain the same but the maintenance version number of the i:th state is larger than the number of the subsequent state. Given these three distinct cases, all m version sequences are processed by comparing $(r_k - 1)$ times the i:th version to the $(i + 1)$:th version, recording the number of downgrades at each step. If d_k denotes the number of downgrades recorded for the k:th domain, a vector $\boldsymbol{\phi} = [d_1 \;/\; r_1 - 1, \ldots, d_m \;/\; r_m - 1]$ defines a simple metric for evaluating how common PHP downgrading has generally been. Analogous to (9), values close to unity indicate frequent downgrading. In theory, also different weights could be used for the three different downgrading scenarios, but this simple counting scheme is sufficient because downgrading should be relatively rare in the context of popular web sites.

The transition matrices $\mathbf{P}_1, \ldots, \mathbf{P}_m$ offer another viewpoint to downgrading: whenever states s_i and s_j communicate (such that there is a transition from s_i to s_j and from s_j to s_i), there is also downgrading of PHP versions. Given this reasoning, a further metric can be computed by counting the number of communicating state pairs and scaling the result appropriately:

$$\gamma_k = \frac{1}{r_k - 1} \sum_{i=1}^{|S_k|} \sum_{j=i}^{|S_k|} \mathrm{I}\left(\hat{p}_{ij}^{(k)}\right) \mathrm{I}\left(\hat{p}_{ji}^{(k)}\right) \tag{10}$$

where $\mathrm{I}(\cdot)$ is an indicator function outputting

$$\mathrm{I}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases} \tag{11}$$

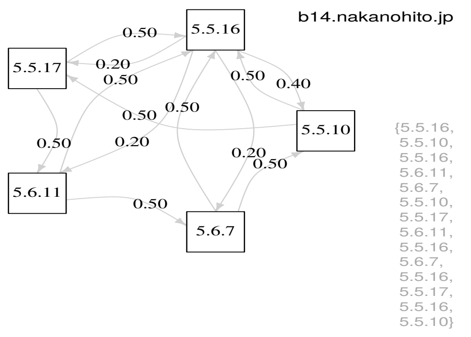

Fig. 4. Another Example DTMC for PHP Release Adoption

As an example, consider the quite messy real-world release adoption pattern visualized in Fig. 4. This particular domain downgraded its PHP deployment as many as seven times during 2016 and early 2017. Therefore, $\phi_k = 7 \;/\; (14 - 1) \simeq 0.54$ and $\gamma_k = 4 \;/\; 13 \simeq 0.31$, given the four communicating pairs. Both values are rather high, which indicates that the two downgrading metrics can be used also for probing outlying domains that may have problems with maintenance of their PHP deployments.

IV. Experimental Results

The empirical results are disseminated by first introducing the data used for the DTMC computation. The metrics elaborated in the previous Section III-C are subsequently used for summarizing the empirical findings.

A. Data

Two datasets are used for the empirical analysis: one for observing short-run release adoption and the other for proxying the long-run evolution of PHP deployments seen in the wild. While calendar-time is not explicitly observed with the DTMCs computed, the definitions for short-run and long-run are still based on calendar-time: with one exception, the short-run dataset covers a period from January 2016 to March 2017 under a monthly sampling frequency, while the long-run dataset is based on annual records in a period between January 2012 and February 2017. Given the PHP release lineage illustrated in Fig. 2, a truly long-run analysis should start already from the year 2000—or even earlier, but the historical periods used are imposed by the source of empirical data. This data source should be also elaborated in more detail.

1) Snapshots: Both datasets are compiled from a few large web crawling snapshots that contain data on hypertext transfer protocol (HTTP) headers used for identifying PHP versions. These open data snapshots are provided by the HTTP Archive web crawling project [33], which has also been used in previous research [13] alongside analogous archives [34], [35]. In total, fourteen crawling snapshots are used for compiling the short-run dataset (see Table I). Due to data availability issues, the long-run dataset is compiled only from six snapshots (see Table II), the earliest of which dates to January 2012.

TABLE I
CHARACTERISTICS OF THE SHORT-RUN DATASET [33][a]

	Start of crawling	User-agent	Size (GB)[b]	PHP domains[c]
1.	March 1, 2017	Chrome	$\simeq 47$	206,739
2.	February 1, 2017	Chrome	$\simeq 45$	202,772
3.	December 2, 2016	Chrome	$\simeq 51$	216,362
4.	November 1, 2016	Chrome	$\simeq 49$	220,871
5.	October 1, 2016	Chrome	$\simeq 51$	222,645
7.	September 1, 2016	Chrome	$\simeq 49$	222,051
7.	August 1, 2016	Chrome	$\simeq 48$	227,757
8.	July 1, 2016	Chrome	$\simeq 48$	225,623
9.	June 1, 2016	Chrome	$\simeq 56$	226,797
10.	May 1, 2016	Chrome	$\simeq 52$	217,592
11.	April 1, 2016	Chrome	$\simeq 50$	219,564
12.	March 1, 2016	Chrome	$\simeq 57$	230,028
13.	February 1, 2016	Chrome	$\simeq 52$	226,859
14.	January 1, 2016	Chrome	$\simeq 50$	225,362

[a] Note that the January 1 snapshot from 2017 was empty and had to be thus excluded. Due to this omission, there is a two month calendar-time delay between the second and the third snapshot. [b] The size refers to the unpacked snapshots. [c] See Section IV-A2 for a definition of a "PHP domain".

TABLE II
CHARACTERISTICS OF THE LONG-RUN DATASET [33][a]

	Start of crawling	User-agent	Size (GB)[b]	PHP domains[c]
1.	February 1, 2017	Chrome	$\simeq 45$	202,772
2.	January 1, 2016	Chrome	$\simeq 50$	225,362
3.	January 1, 2015	IE[d]	$\simeq 41$	247,568
4.	January 1, 2014	IE[d]	$\simeq 24$	158,775
5.	January 1, 2013	IE[d]	$\simeq 22$	165,741
6.	January 1, 2012	IE[d]	$\simeq 3.7$	31,445

[a,b,c] See the notes in Table I. [d] The abbreviation stands for Internet Explorer.

As shown in the two tables, the raw snapshots used are quite large. Because the crawls are seeded from Alexa's list of top-million busiest web sites [36], which is updated daily, the snapshot sizes also vary from a crawl to another. Moreover, it should be emphasized that the dates shown are only tentative regarding individual HTTP requests and responses: due to the large seeding list, crawling can take a relatively long amount of time [35]. Already because calendar-time is not explicitly observed, the issue is a minor concern for this paper, however.

The long-run dataset is affected by a change in the forged user-agent [37] used for making the requests. Although user-agents can have a substantial empirical effect for measuring web sites due to specific responses for specific browsers [38], the consequences should be small in this paper because it seems unlikely that PHP version strings in the HTTP response headers would vary according to a user-agent specified in the HTTP request headers. Therefore, it is more important to further remark that the long-run dataset is affected by changes made to the seeding of the web crawls, which is reflected in the smaller amount of PHP domains between 2012 and 2014. In contrast, each snapshot in the short-run dataset contains roughly the same amount of domains. Finally, it should be emphasized that the total amount of PHP-powered domains observed is substantially larger than reported in Tables I and II because the snapshots are "pooled" to include all domains that are present in at least two snapshots.

2) *Pre-processing:* The snapshots were pre-processed from the packaged archives delivered as CSV (comma-seperated value) files. Although the files are provided as open data, a couple of remarks should be made to ensure replicability of the datasets. First and foremost, the presence of PHP is identified via s simple (Python) regular expression of the following form: "`PHP/[0-9]{1}\.[0-9]{1}\.[0-9]{1,}`", where the quotation marks are not part of the expression. Notice that the expression excludes "invalid" versions such as `PHP/3.100`.

Second, unique domains are identified by extracting the network location from the uniform resource locators (URLs) crawled. Because multiple web pages may be crawled for each domain, duplicates are excluded by omitting the parsing of URLs for domains that have already been identified to run with PHP. It should be remarked that the concept of domain is inexplicit in the sense that no attempts are made to lookup the domains via the domain name system. Therefore, in theory, the domains may refer to actual domain names as well as Internet protocol addresses. For the purposes of this paper, the distinction is irrelevant, however.

B. Results

The dissemination of the results can be started by noting a few characteristics of the two datasets compiled from the web crawling snapshots. First and foremost, according to the numbers shown in Table III, about 451 and 220 hundred thousand domains were identified as running with PHP according to the simple pre-processing routines. By implication, well over half a million transition probabilities were estimated via (6). Second, on average, about a half of the maximum lengths of the version sequences are realized in the two datasets, although the standard deviations are large. In other words, a typical domain is observed a little over six times in the short-run and about three times in the long-run. The reason for not reaching the maximum lengths is simple: because the snapshots are not crawled from a fixed domain set, not all of the PHP domains observed are present across all snapshots. Third, the average size of the state space, $\frac{1}{m}\sum_{k=1}^{m}|S_k|$, is less than two in both datasets. Thus, for many domains, the transition probabilities are represented by the value one supplied via a 1×1 matrix. While there is still a sufficient amount of variance for analysis, already this observation allows to conclude that the prevalence of PHP release adoption has been at a modest level. Before continuing to the actual prevalence metric, a remark should be made about the most common PHP releases in the datasets.

According to the datasets, it is clear that the PHP 5 release branch has been the most popular deployment choice from the early 2000s onward. As shown in Fig. 5, the clear majority of the versions observed in the long-run are part of the PHP 5 branch. While there has also been few domains using the

TABLE III
SAMPLE CHARACTERISTICS

		Short-run	Long-run
Number of domains (m)		451,340	220,293
Number of versions (r)	Mean	6.5	2.9
	Std. dev.	4.0	1.1
Size of state space ($\|S\|$)	Mean	1.5	1.8
	Std. dev.	1.2	0.8

TABLE IV
UNIFORM PHP VERSION SEQUENCES

	Subset	Short-run	Long-run
Number of unique sequences	$\|S_k\| = 1$	2,470	568
	$\|S_k\| > 1$	62,376	43,585
Share of unique sequences (%)	$\|S_k\| = 1$	0.55	0.26
	$\|S_k\| > 1$	13.8	19.8

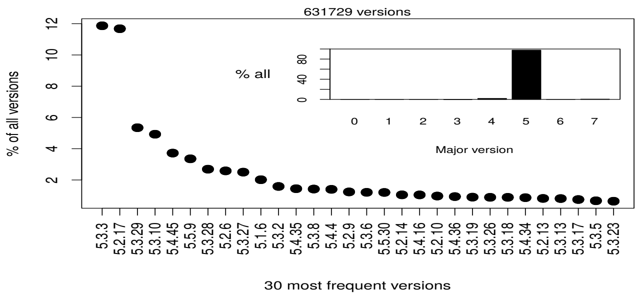

Fig. 5. Most Frequent PHP Releases in the Long-Run

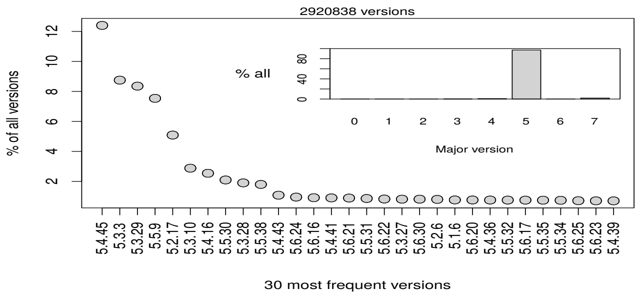

Fig. 6. Most Frequent PHP Releases in the Short-Run

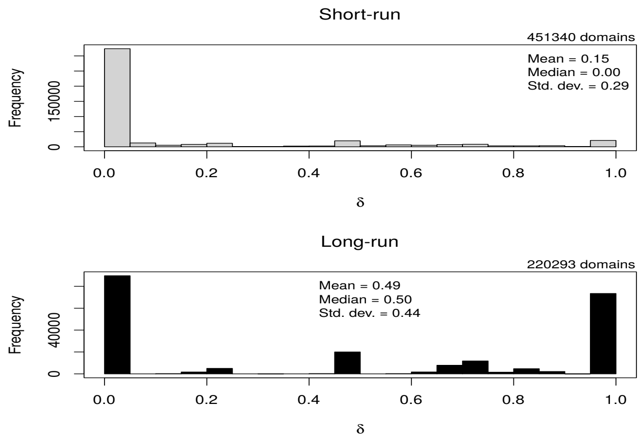

Fig. 7. Prevalence of PHP Release Adoption

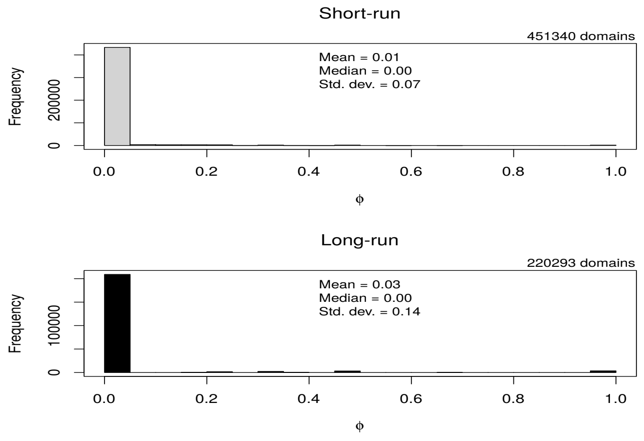

Fig. 8. Downgrading of PHP Releases #1

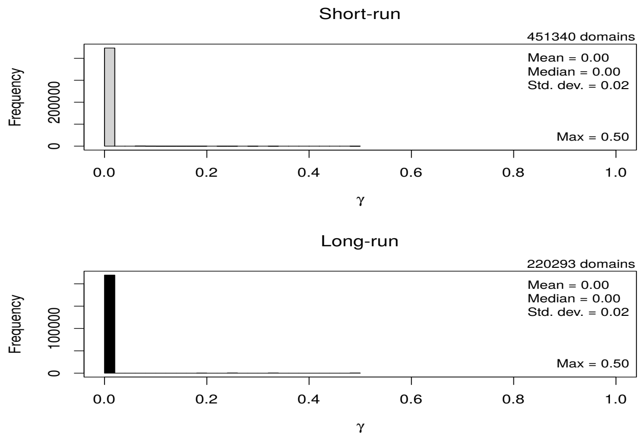

Fig. 9. Downgrading of PHP Releases #2

PHP 4 branch, the use of other major branches has been negligible in the long-run. The fifth major version has also retained its popularity in 2016 and 2017, as can be concluded from the subsequent Fig. 6. Adoption of PHP 7 has been modest (RQ$_1$). This conclusion does not change when only the last versions in the sequences are used to proxy popularity; in this case, only about 1.7 % of the domains have used a PHP 7 deployment in the short-run

The observation about relatively infrequent release adoption is reinforced by Fig. 7, which shows a histogram of the metric in (9) across the domains observed in the two datasets. There exists a difference between short-run release adoption and long-run evolution, however. The upper plot clearly indicates that monthly release adoption has generally been infrequent in 2016 and early 2017. In contrast, the lower plot displays a bimodal distribution: many domains observed have not adopted releases in the long-run, but almost an equal amount of domains have adopted releases annually. All in all, the prevalence of PHP release adoption has been modest in recent years (RQ_2), although less so in the long-run (RQ_5).

Moreover, PHP release adoption patterns have been relatively uniform across the domains observed (RQ_4), as can be concluded from the summary shown in Table IV. In total, only about $0.55 + 13.8 \simeq 14.4$ and 20.1 percent of the observed PHP version sequences are unique in the short-run and long-run datasets, respectively. While it should be kept in mind that these relative amounts are affected by the large amount of domains observed, these relative amounts still indicate a modest amount of unique release adoption patterns. Most of the patterns in both datasets describe common transition paths within the PHP 5 major release branch. From a release engineering perspective, this observation is a positive finding: most domains follow other domains in their upgrading patterns. Disorderly release adoption patterns are relatively rare.

The observations about infrequent release adoption and uniformity are reinforced by Fig. 8, which shows the frequency of the first downgrading metric in the two datasets. Most domains have not downgraded their PHP deployments even once. Although downgrading is slightly more common in the long-run, the standard deviations are generally small. By implication, the same observation applies also for the results regarding the second downgrading metric in (10). As can be concluded from Fig. 9, only a very few of the version sequences involve communicating PHP version pairs. The averages and standard deviations are both negligible. To summarize, downgrading has been rare (RQ_3), and there are no notable differences between short-run and long-run (RQ_5).

V. Discussion

The remainder of this paper first summarizes the main empirical findings, then enumerates a few threats to validity, and finally concludes with a couple of new research directions.

A. Summary of Results

This empirical paper observed PHP release adoption in two datasets covering over a half a million Internet domains and three million PHP versions deployed within these domains. The main findings can be summarized by briefly answering to the five research questions outlined in Section II-B.

- Adoption of PHP 7 has been modest (RQ_1). As of early 2017, only few popular web sites have adopted the new major release branch. Most sites continue to operate with releases made within the PHP 5 major branch.
- The prevalence of PHP release adoption has been at a modest level: popular web sites tend to upgrade their deployments relatively infrequently (RQ_2). The observation aligns with previous studies; relatively old PHP 5 versions are commonly used in cloud computing services [39].
- Downgrading has been uncommon; only a few outlying domains have downgraded their deployments (RQ_3).
- The adoption patterns have been highly uniform across popular domains; most domains tend to follow similar upgrading paths used also by other domains (RQ_4).
- Only the prevalence of adoption (RQ_2) varies between the short-run history (2016 – early 2017) and long-run evolution (2012 – early 2017). Namely, the longer the period observed, the more common has adoption been.

These findings provide also some material for contemplating about the current release engineering strategy of the PHP project. For the developers of the programming language, a pressing question relates to the means by which the currently slow adoption of PHP 7 could be boosted in the future. One option to consider might be a Firefox-style rapid release strategy, which has been suspected to increase user adoption compared to a traditional release strategy [10]. Because adoption has generally been infrequent among popular web sites, it might be possible to also debate whether the current release schedule is actually already too rapid for users and stakeholders. Although downgrading is rare and the adoption patterns are generally uniform, a further interesting question relates to the reasons why some domains downgrade, and whether there is anything that could be done to help outlying domains following chaotic release adoption paths.

B. Threats to Validity

Threats to validity can be enumerated by using the conventional threefold classification of construct validity, external validity, and internal validity. Although there exists no uniformly agreed definitions [40], for the purposes of this paper, these three validity concepts can be equated to questions related to generalizability (how results generalize to a different context or population), operationalization (how well a quantification matches a theoretical concept), and systematic computational errors (how well different biases are eliminated), respectively.

1) External Validity: Generalizability questions are always present when the theoretical population is the whole world wide web [35], including the so-called "deep web" not indexed by standard search engines. Even though generalizability toward such a population is practically impossible even for companies such as Google, it is possible to narrow the target of generalization toward a sub-population of the most popular PHP-powered domains. In this regard, HTTP Archive uses Alexa's popularity list, which is commonly perceived as a good choice for seeding of large-scale web crawling [41], [42], [43]. While external validity is presumably not threatened in this regard, (a) the results reported are likely specific to popular web sites. When considering further applications, such as those motivated by security questions [17], it is likely that more interesting cases are located in the fringes of the world wide web. Given that prevalence of PHP release adoption was observed to be at a modest level in a sample of popular sites, it is more than likely that even lower levels of adoption could be observed in a sample covering WordPress deployments, for instance. A common limitation [44] is also present: (b) the results apply only to domains using PHP for serving pages via plain HTTP, excluding sites using HTTPS.

2) Construct Validity: A notable threat to construct validity stems from the identification of PHP deployments via a regular expression from HTTP response headers (see Section IV-A2). This coarse identification technique is likely to include both false positives (popular domains incorrectly identified as running with PHP) and false negatives (the missing of popular PHP-powered domains). To evaluate the severity of this limitation, at minimum, parallel identification should be attempted from the actual web page content (cf. [39]). Because the primary identification requirement relates to the version of a PHP backend used for serving a particular web content, robust identification is likely challenging also from web page contents, however. Further research is therefore required to continue the work on identifying and fingerprinting PHP applications [45], including the PHP interpreter itself.

3) Internal Validity: The potential presence of systematic biases is best evaluated against the classical DTMC assumptions that were imposed for the statistical computation (see Section III-A). There are three notable concerns about these assumptions. First, the assumption in Eq. (1) implies that regardless whether a state change is due to security updates, reliability improvements, or new features, it is always the currently deployed version that defines the reference point for the change, regardless whether the decision to change versions is made by a human or a package manager. While the assumption seems sensible from a release engineering viewpoint, it is easily questioned from a software evolution perspective [5], [23]. If history matters also for PHP release adoption patterns, it would seem reasonable to recommend that further research should focus on higher-order Markov chains that have a memory [5], [46], [30]. The second concern relates to the assumption of independence between domains. Given that a substantial amount of contemporary web sites require connections to two or more servers [34], [47], PHP deployments may be uniformly managed and upgraded in a cloud computing service or other large deployment farm. Consequently, a PHP version sequence of a domain might be affected by a sequence of another domain. Conditional Markov chains [46], [48] may provide a useful tool for evaluating the potential severity of this cross-domain dependence assumption.

The third notable threat to internal validity relates to the PHP version sequences observed, which mandate making an addition assumption about the transition probabilities in (6). Consequently, by definition [49], the transition probability matrices computed are not stochastic matrices, that is, the row sums of these matrices do not necessarily equal one. Although this unavoidable limitation does not affect the results reported as such, it does affect additional computations involving eigenvalues [31], and particularly the stationary distributions toward which all irreducible, aperiodic, and positive recurrent Markov chains converge (for the mathematical background see [50]). This point should be kept in mind when considering further DTMC applications in the release engineering and software evolution contexts. Such applications are also a good way to point out a couple of new research directions.

C. Further Work

The primary purpose of this paper was to examine the usefulness of DTMC modeling for systematic tracing of web deployments in order to establish automated continuous feedback channel for server-side programming language developers and stakeholders. The paper fulfilled this goal: DTMCs are useful also in the release engineering context. For pursuing DTMC analysis further, a worthwhile goal would be to translate some of the concepts used in other disciplines to the language of release engineering and software evolution. For instance, simple DTMC metrics have been used to proxy such concepts as colonization, disturbance, and replacement [31]. With some theoretical and terminological alterations, such metrics and concepts could be adopted for pursuing DTMC modeling further in the release engineering context. While these concepts and metrics are directly applicable to traditional DTMCs, another prolific path forward involves altering the basic assumptions surrounding discrete time-homogeneous Markov chains. Conditional and higher-order chains are good examples in this regard. For continuous tracing of PHP deployments, continuous Markov chains (as opposed to discrete-time chains) seem prolific to consider in further research. For instance, different time-delay models [4] could be adopted for studying the time delays between successive state changes. The question about time delays is also fundamental in the release engineering context because the empirical transition probabilities depend on the sampling frequency used.

Acknowledgements

The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIMECC Oy, and the Cyber Trust research program for their support.

References

[1] J.-M. Favre, "Languages Evolve Too! Changing the Software Time Scale," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE 2005)*. Lisbon: IEEE, 2005, pp. 33–42.

[2] L. A. Meyerovich and A. S. Rabkin, "Socio-PLT: Principles for Programming Language Adoption," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Tucson: ACM, 2012, pp. 39–54.

[3] R. C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 118–125, 1980.

[4] W. Wang, "An Overview of the Recent Advances in Delay-Time-Based Maintenance Modeling," *Reliability Engineering and System Safety*, vol. 106, pp. 165–178, 2012.

[5] S. Wong and Y. Cai, "Generalizing Evolutionary Coupling with Stochastic Dependencies," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence: IEEE, 2011, pp. 293–302.

[6] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*. Victoria: IEEE, 2014, pp. 215–224.

[7] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a Library: A Study of the Latency to Adopt the Latest Maven Release," in *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, Montreal, 2015, pp. 520–524.

[8] T. Karvonen, W. Behutiye, M. Oivo, and P. Kuvaja, "Systematic Literature Review on the Impacts of Agile Release Engineering Practices," *Information and Software Technology*, vol. 86, pp. 87–100, 2017.

[9] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, "On Rapid Releases and Software Testing," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSME 2013)*. Madrid: IEEE, 2013, pp. 20–29.

[10] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues: An Empirical Study of the Mozilla Firefox Project," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*. Austin: ACM, 2016, pp. 374–385.

[11] B. Adams and S. McIntosh, "Modern Release Engineering in a Nutshell – Why Researchers Should Care," in *Proceedings IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. Osaka: IEEE, 2016, pp. 78–90.

[12] O. Baysal, R. Holmes, and M. W. Godfrey, "Mining Usage Data and Development Artifacts," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR 2012)*. Zurich: IEEE, 2012, pp. 98–107.

[13] J. Ruohonen, S. Hyrynsalmi, and V. Leppänen, "Exploring the Use of Deprecated PHP Releases in the Wild Internet: Still a LAMP Issue?" in *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016)*. Nîmes: ACM, 2016, pp. 26:1–26:12.

[14] T. Amanatidis and A. Chatzigeorgiou, "Studying the Evolution of PHP Web Applications," *Information and Software Technology*, vol. 72, pp. 48–67, 2016.

[15] M. Hills, "Evolution of Dynamic Feature Usage in PHP," in *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*. Montreal: IEEE, 2015, pp. 525–529.

[16] M. Hills, P. Klint, and J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2013)*. Lugano: ACM, 2013, pp. 325–335.

[17] I. Medeiros, N. Neves, and M. Correia, "Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[18] B. Fitzgerald and K. Stol, "Continuous Software Engineering: A Roadmap and Agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2015.

[19] C. Pang and A. Hindle, "Continuous Maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2016)*. Raleigh: IEEE, 2016, pp. 458–462.

[20] A. Dyck, R. Penners, and H. Lichter, "Towards Definitions for Release Engineering and DevOps," in *Proceedings of the Third International Workshop on Release Engineering (RELENG 2015)*. Florence: IEEE, 2015, pp. 3–3.

[21] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The Highways and Country Roads to Continuous Deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[22] R. P. de Oliveira, A. R. Santos, E. S. de Almeida, and G. S. da Silva Gomes, "Evaluating Lehman's Laws of Software Evolution Within Software Product Lines Industrial Projects," *Journal of Systems and Software*, vol. 131, pp. 347–365, 2016.

[23] J. Ruohonen, S. Hyrynsalmi, and V. Leppänen, "Time Series Trends in Software Evolution," *Journal of Software: Evolution and Process*, vol. 27, no. 2, pp. 990–1015, 2015.

[24] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proceedings of IEEE International Conference on Requirements Engineering (ICRE 1994)*. IEEE, 1994, pp. 94–101.

[25] P. Sturgeon, "The Neverending Muppet Debate of PHP 6 v PHP 7," 2014, Available online in March 2017: https://philsturgeon.uk/php/2014/07/23/neverending-muppet-debate-of-php-6-v-php-7/.

[26] The PHP Project, "Request for Comments: Release Process," 2010, Available online in March 2017: https://wiki.php.net/rfc/releaseprocess.

[27] ——, "Supported Versions," 2017, Available online in March 2017: http://php.net/supported-versions.php.

[28] ——, "Unsupported Historical Releases," 2017, Available online in March 2017: https://secure.php.net/releases/.

[29] M. V. Mäntylä and J. Vanhanen, "Software Deployment Activities and Challenges – A Case Study of Four Software Product Companies," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*. Oldenburg: IEEE, 2011, pp. 131–140.

[30] P. Singer, D. Helic, B. Taraghi, and M. Strohmaier, "Detecting Memory and Structure in Human Navigation Patterns Using Markov Chain Models of Varying Order," *PLOS ONE*, vol. 9, no. 7, p. e102070, 2014.

[31] M. F. Hill, J. D. Witman, and H. Caswell, "Markov Chain Analysis of Succession in a Rocky Subtidal Community," *The American Naturalist*, vol. 164, no. 2, pp. E46–E61, 2004.

[32] G. A. Spedicato, "markovchain: Discrete Time Markov Chains Made Easy," 2016, R package version 0.6, available online in March 2017: https://cran.r-project.org/web/packages/markovchain/index.html.

[33] HTTP Archive, "Downloads," 2017, Available online in March 2017: http://httparchive.org/downloads.php.

[34] T. Wambach and K. Bräunlich, "The Evolution of Third-Party Web Tracking," in *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP 2016)*, O. Camp, S. Furnell, and P. Mori, Eds. Rome: Springer, 2016.

[35] S. G. Ainsworth and M. L. Nelson, "Evaluating Sliding and Sticky Target Policies by Measuring Temporal Drift in Acyclic Walks Through a Web Archive," in *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL 2013)*. Indianapolis: ACM, 2013, pp. 39–48.

[36] HTTP Archive, "FAQ," 2017, Available online in March 2017: http://httparchive.org/about.php#faq.

[37] M. C. Calzarossa and L. Massari, "Analysis of Header Usage Patterns of HTTP Request Messages," in *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*. Paris: IEEE, 2014, pp. 847–853.

[38] K. Pham, A. Santos, and J. Freire, "Understanding Website Behavior Based on User Agent," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2016)*. Pisa: ACM, 2016, pp. 1053–1056.

[39] L. Wang, A. Nappa, J. Caballero, T. Ristenpart, and A. Akella, "WhoWas: A Platform for Measuring Web Deployments on IaaS Clouds," in *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC 2014)*. Vancouver: ACM, 2014, pp. 101–114.

[40] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*. Florence: IEEE, 2015, pp. 9–19.

[41] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan, "Adscape: Harvesting and Analyzing Online Display Ads," in *International Conference on World Wide Web (WWW 2014)*. Seoul: ACM, 2014, pp. 597–608.

[42] Y. J. Park, "A Broken System of Self-Regulation of Privacy Online? Surveillance, Control, and Limits of User Features in U.S. Websites," *Policy & Internet*, vol. 6, no. 4, pp. 360–376, 2014.

[43] A. F. Tappenden and J. Miller, "Cookies: A Deployment Study and the Testing Implications," *ACM Transactions on the Web*, vol. 3, no. 3, pp. 9:1 – 9:49, 2009.

[44] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, "Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure," in *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC 2013)*. Barcelona: ACM, 2013, pp. 177–190.

[45] M. Kozina, M. Golub, and S. Groš, "A Method for Identifying Web Applications," *International Journal of Information Security*, vol. 8, no. 6, pp. 455–467, 2009.

[46] W.-K. Ching, M. K. Ng, and E. S. Fung, "Higher-Order Multivariate Markov Chains and Their Applicatons," *Linear Algebra and Its Applications*, vol. 428, no. 2–3, pp. 492–507, 2008.

[47] B. Newton, K. Jeffay, and J. Aikat, "The Continued Evolution of Web Traffic," in *Proceedings of the IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)*. IEEE, 2013, pp. 80–89.

[48] S. Goutte, "Conditional Markov Regime Switching Model Applied to Economic Modelling," *Economic Modelling*, vol. 38, pp. 258–269, 2014.

[49] G. A. F. Seber, *A Matrix Handbook for Statisticians*. New Jersey: John Wiley & Sons, 2008.

[50] N. Privault, *Understanding Markov Chains: Examples and Applications*. Heidelberg: Springer, 2013.