The Co-evolution of the WordPress Platform and its Plugins

JIAHUEI LIN, Queen's University, Canada
MOHAMMED SAYAGH, ETS - Quebec University, Canada
AHMED E. HASSAN, Queen's University, Canada

One can extend the features of a software system by installing a set of additional components called plugins. WordPress, as a typical example of such plugin-based software ecosystems, is used by millions of websites and has a large number (i.e., 54,777) of available plugins. These plugin-based software ecosystems are different from traditional ecosystems (e.g., NPM dependencies) in a sense that there is a high coupling between a platform and its plugins, compared to traditional ecosystems for which components might not necessarily depend on each other (e.g., NPM libraries do not depend on a specific version of NPM or a specific version of a client software system). The high coupling between a plugin and its platform and other plugins causes incompatibility issues that occur during the co-evolution of a plugin and its platform as well as other plugins. In fact, incompatibility issues represent a major challenge when upgrading WordPress or its plugins. According to our study of the top 500 most-released WordPress plugins, we observe that incompatibility issues represent the 3^{rd} major cause for bad releases, which are rapidly (within the next 24 hours) fixed via urgent releases. 32% of these incompatibilities are between a plugin and WordPress while 19% are between peer plugins. In this paper, we study how plugins co-evolve with the underlying platform as well as other plugins, in an effort to understand the practices that are related support such co-evolution and reduce incompatibility issues. In particular, we investigate how plugins support the latest available versions of WordPress, as well as how plugins are related to each other, and how they co-evolve. We observe that a plugin's support of new versions of WordPress with a large amount of code change is risky, as the releases which declare such support have a higher chance to be followed by an urgent release compared to ordinary releases. Although plugins support the latest WordPress version, plugin developers omit important changes such as deleting the use of removed WordPress APIs, which are removed a median of 873 days after the APIs have been removed from the source code of WordPress. Plugins introduce new releases that are made according to a median of 5 other plugins, which we refer to as peer-triggered releases. A median of 20% of the peer-triggered releases are urgent releases that fix problems in their previous releases. The most common goal of peer-triggered releases is the fixing of incompatibility issues that a plugin detects as late as after a median of 36 days since the last release of another plugin. Our work sheds light into the co-evolution of WordPress plugins with their platform as well as peer plugins in an effort to uncover the practices of plugin evolution, so WordPress can accordingly design approaches to avoid incompatibility issues.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Maintaining software**;

Additional Key Words and Phrases: Plugin-based Ecosystems - Incompatibility issues - plugins co-evolution

ACM Reference Format:
Jiahuei Lin, Mohammed Sayagh, and Ahmed E. Hassan. 2021. The Co-evolution of the WordPress Platform and its Plugins. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 25 pages. https://doi.org/10.1145/3447876

Authors' addresses: Jiahuei Lin, jhlin@cs.queensu.ca, Queen's University, Software Analysis and Intelligence Lab (SAIL), Kingston, ON, Canada; Mohammed Sayagh, mohammed.sayagh@etsmtl.ca, ETS - Quebec University, Department of Software Engineering and IT, Montreal, QC, Canada; Ahmed E. Hassan, ahmed@cs.queensu.ca, Queen's University, Software Analysis and Intelligence Lab (SAIL), Kingston, ON, Canada.

Manuscript submitted to ACM

1 INTRODUCTION

A plugin-based software ecosystem is an ecosystem that is constituted of a platform that can be extended with a set of "plugins". WordPress, as a typical example of a plugin-based ecosystem, has more than 54,777 plugins that are available to end-users. For example, one can install the *FV Flowplayer Video Player* plugin to display videos on his WordPress website. The richness of these plugins makes WordPress one of the most popular content management platforms. In fact, WordPress is used by 61.8% of all the websites, whose content management system is known [38]. WordPress is also used to build well known websites such as the New York Post, USA Today, and TIME.com websites [1].

Plugin-based software ecosystems are different from traditional ecosystems. For instance, the relation between the plugins and the WordPress platform is symbiotic. The features that are provided by the platform are often limited and need to be extended through the plugins. The plugins, on the other side, cannot be used without the platform. In addition, plugins interfere with each other through their shared platform. That is contrary to traditional ecosystems, for which a component can often evolve independently from the other components. In addition, components in traditional ecosystems might never interfere with each other if they do not share any resources or do not invoke each other. Thus, there exists a high coupling between plugins and the platform as well as the different plugins for plugin-based software ecosystems.

Such a high coupling leads to incompatibility issues that occur when a plugin co-evolves with the platform or with other plugins. For example, a plugin might introduce a release that is incompatible with the platform or with other plugins. Given the richness of plugins, the freedom to intermix plugins, and their varying release cycles, the users of WordPress often must cope with incompatibility issues, which discourage users from upgrading their WordPress and plugins [5]. For example, a user's website crashed after upgrading to the latest WordPress version due to an incompatibility between a plugin and that latest WordPress version [9]. In other words, the plugin did not co-evolve correctly with its platform.

A large amount of research has examined incompatibility issues in traditional ecosystems, such as project dependencies [10, 17, 26, 41] (i.e., project dependency refers to a relation between two working applications that contain a set of source files, configurations, assets, etc.), out-of-date dependencies [28, 33], library dependencies [23, 31], and package dependencies [15, 22] (i.e., package dependency refers to a package consisting of a group of related classes and source files that require another package in order to work). However, a few research efforts [16, 35] investigated the incompatibility issues in plugin-based software ecosystems, which have higher chances for incompatibility issues compared to traditional ecosystems, as we noted earlier.

Through a preliminary study of the top 500 WordPress plugins with the largest number of releases, we observe that incompatibility issues represent the 3rd major cause of bad releases, which are rapidly addressed via urgent releases. Note that this type of releases is also studied by prior work [19, 25], which also defined the urgent releases as the releases whose goal is to fix errors that were introduced in a previous release that was recently published (within 24 hours before the urgent release). 32% of these incompatibility issues occur between plugins and the WordPress platform while 19% are between peer plugins.

The goal of this paper is to shed light into the co-evolution of plugins with their platform and other plugins in an effort to understand the practices of such co-evolution, so WordPress can better design approaches to avoid incompatibility issues. We focus our study on the WordPress ecosystem since it is one of the most successful and widely used plugin-based ecosystems. In particular, we address the following research questions:

RQ1: When plugins support a new version of WordPress?

Plugins announce their supports for a newly released version of WordPress after a median of 38 days, with a lag of one release behind the latest available version of WordPress. However, each plugin does not have a consistent lag for every WordPress version. Plugin developers use a new WordPress API as fast as 1 day after its introduction and delete the use of removed APIs after a median of 873 days.

RQ2: How plugins support a new version of WordPress?

89% of the studied plugins have at least one release that does not support the latest version of WordPress. These releases have a higher chance of being followed by an urgent release (153%) compared to ordinary releases. Furthermore, plugin releases that announce support to a new version of WordPress with a large amount of code changes are more likely to be followed by an urgent release.

RQ3: How WordPress plugins co-evolve?

Plugins ship new releases that are triggered due to a median of 5 other plugins (i.e., peer-triggered releases). 60% of the pair of plugins that have a peer-triggered release do not explicitly depend on each other. Peer-triggered releases have a lower chance of being followed by an urgent release (66%) and a median of 20% of the peer-triggered releases are urgent releases. Fixing incompatibility issues is the most common goal (46%) for peer-triggered releases.

Our results provide an understanding on how a plugin co-evolves with its platform and other peer plugins. Such an understanding provides WordPress developers with better understanding of the co-evolution in plugin-based software ecosystems in order to avoid incompatibility issues.

Paper organization. Section 2 provides background information about the release process of WordPress plugins and describes the related work to our study. Section 3 provides our preliminary analysis on the incompatibilities that are fixed through urgent releases. Section 4 answers our research questions. Section 5 discusses threats to the validity of our observations. Finally, Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, we provide background information about how plugins ship new releases and support new versions of WordPress (Section 2.1). We discuss related work on empirical studies of the co-evolution of different components of an ecosystem (Section 2.2) and incompatibility issues (Section 2.3) .

2.1 How plugins support new WordPress versions

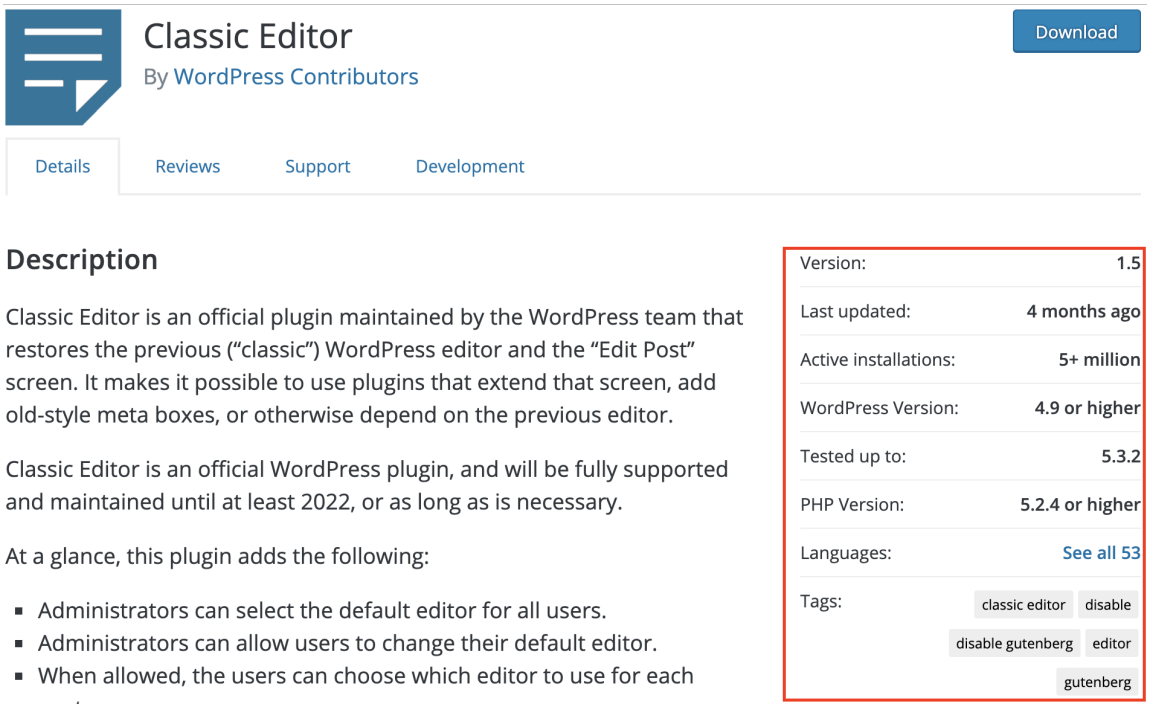
Each plugin has a "readme" file that contains the meta-data of the plugin, such as the latest available version and the versions of WordPress that the plugin officially supports. The *readme* file of a plugin is used by the WordPress platform to auto-generate a web page for that plugin. For example, the latest available version of the *"Classic Editor"* plugin is *1.5*, as indicated by the "*Stable-tag*" in the *readme* file, and as displayed on the web page of the plugin, as shown in Figure 1. The "Tested up to" tag indicates that the plugin is supporting WordPress versions up to *5.3.2*. The *readme* file changes are recorded in the source code repository of each plugin.

2.2 Co-evolution of different components in ecosystems

Prior work studied the co-evolution of a project and its dependencies [11, 21, 23, 27]. Hora et al. [21] observed that developers discover and react to API changes after a median of 34 days and 95% of the software systems never adopt

```
Line
1  === Classic Editor ===
2  Contributors: wordpressdotorg, azaozz, melchoyce, chanthaboune, alexislloyd, pento, youknowriad, desrosj, luciano-croce
3  Tags: gutenberg, disable, disable gutenberg, editor, classic editor, block editor
4  Requires at least: 4.9
5  Tested up to: 5.3
6  Stable tag: 1.5
7  Requires PHP: 5.2.4
8  License: GPLv2 or later
9  License URI: http://www.gnu.org/licenses/gpl-2.0.html
```

(a) An example of a readme file



(b) A plugin web page example. The metadata information in the red box are generated from the readme file. The information indicates the versions of the plugin, the supported WordPress version, and programming languages. Note that the *Tested-up-to* tag only defines the minor version (e.g.,5.3) tested by plugins and WordPress will automatically add the latest micro version (e.g., 5.3.2) to it.

Fig. 1. An example of a readme file and the web page displaying the associated information in the readme file.

new API changes. Kula et al. [23] identified that 81.5% of the software systems stick with outdated dependencies, due to the overhead of updating such dependencies. Bavota et al. [11] observed that developers update to the new version of a dependency when that new version includes major changes such as new features or a large amount of bug fixes. The co-evolution of different components of an ecosystem leads to incompatibility issues, which impacts the stability and fault-proneness of an ecosystem [12], and results in poor user experiences [40].

Another line of research is related to breaking changes [13, 14, 32]. Raemaekers et al. [32] studied subsequent releases of more than 20,000 libraries in the Maven repository and observed that one third of all releases introduced at least one breaking change (e.g., method removals) that affects their client systems. Raemaekers et al. [32] also observed that developers tend to keep up with the latest version of dependent libraries with a median of 0 release lag of the

versions between the library version included in a project and the latest available version of the library. Bogart et al. [13] observed the npm ecosystem focuses strongly on signaling change through semantic versioning for developers to manage their dependencies, while developers bear to update their dependencies when breaking changes are made in the dependencies in the R ecosystem. Even breaking changes are costly in terms of interruptions and rework, each software ecosystem makes such braking changes for various purposes. For example Bogart et al. [13] reported that the most important reason for breaking changes is technical debt, rather than bugs or new features in the Eclipse, R and npm ecosystems. Brito et al. [14] reported that another important reason for breaking changes is to implement new features to simplify source code and improve maintainability in Java libraries of popular GitHub projects.

2.3 WordPress incompatibilities

A few prior studies focused on the incompatibility issues in the WordPress ecosystem [16, 30, 34, 35]. Sayagh et al. [34, 35] observed that plugins share a large percentage of the platform configuration options. For example, 79% of all WordPress configurable constants and 85.16% of all WordPress database options are used by at least two different plugins, which suggests potential conflicts between plugins. Nguyen et al. [30] leveraged a source code analysis technique to identify incompatibility errors between different WordPress plugins. Eshkevari et al. [16] levareged static and dynamic source code analysis techniques to identify incompatibility issues between two plugins. While these studies focused on how to detect and fix incompatibility issues, our study focuses on investigating the co-evolution of the plugin-platform and peer-plugins, which can guide WordPress developers to develop mechanisms that support such co-evolution to prevent incompatibility issues.

3 (PRELIMINARY QUESTION) WHAT IS THE PROPORTION OF URGENT RELEASES THAT ARE DUE TO INCOMPATIBILITY ISSUES?

Motivation: The goal of this preliminary research question is to quantify the prevalence of incompatibility issues. These issues occur due to missed interactions between a plugin and the platform or other peer plugins. In this preliminary research question, we focus on the interactions that were missed and that are critical enough to lead to urgent releases. These releases are shipped within 24 hours from their respective prior release. Urgent releases are perceived to be harmful as noted by several prior studies (e.g., [18, 25]).

Approach: To study how often incompatibility issues lead to urgent releases, we first need to investigate the prevalence of urgent releases and whether the urgent releases are related to their prior releases. Then, we investigate the main causes of the urgent releases in order to quantify how often they occur due to incompatibility issues.

To do so, we investigate the releases of the top 500 most-released plugins (the **Plugin Dataset**). To obtain these plugins, we collect all the 60,592 plugins that have a *readme* file and the *"Stable-tag"* and sort these plugins based on the number of their releases, which we obtain from the historical changes to the *"Stable-tag"* in a plugin's *readme* file.

To address our preliminary research question, we first quantify the number of urgent releases for each of the top 500 most-released WordPress plugins. We then conduct a qualitative analysis on a representative random sample (confidence level = 95%, confidence interval = 5%) of 375 urgent releases. The goal of this qualitative analysis is to first identify whether urgent releases are related to their prior release. We consider that an urgent release is related to its prior release (1) when both of the release notes of the urgent and prior release mention the same information (e.g., features), or (2) when both releases change the same parts of the code (e.g., functions or classes). However, we cannot conclude if

an urgent release is related to its prior release when both releases change different parts of the code. The second goal of the qualitative analysis is to quantify how often the urgent releases are caused by incompatibility issues. To do so, we use the extended Swanson's maintenance classification [20, 39] to label the causes of urgent releases. We perform an iterative process that is similar to prior work [36, 37] to figure out what causes an urgent release. In particular, we manually analyze the release notes and commit messages of the 375 urgent releases against their respective prior release. For example, one stated *"fixing default sort"* in the release note of version 2.12.6 of the *12-step-meeting-list*[1] plugin, which was an urgent release. The sort feature was introduced in version 2.12.5 as stated *"now able to set sort order on meetings page"* in its release note. Meanwhile, there were overlaps between the code changes made in version 2.12.6 and 2.12.6. Therefore, we determine that the urgent release 2.12.6 was to fix bugs in version 2.12.5 and label the root cause of the urgent release as a malfunction. Note that our dataset contains 16,444 urgent releases for the top 500 most-released plugins.

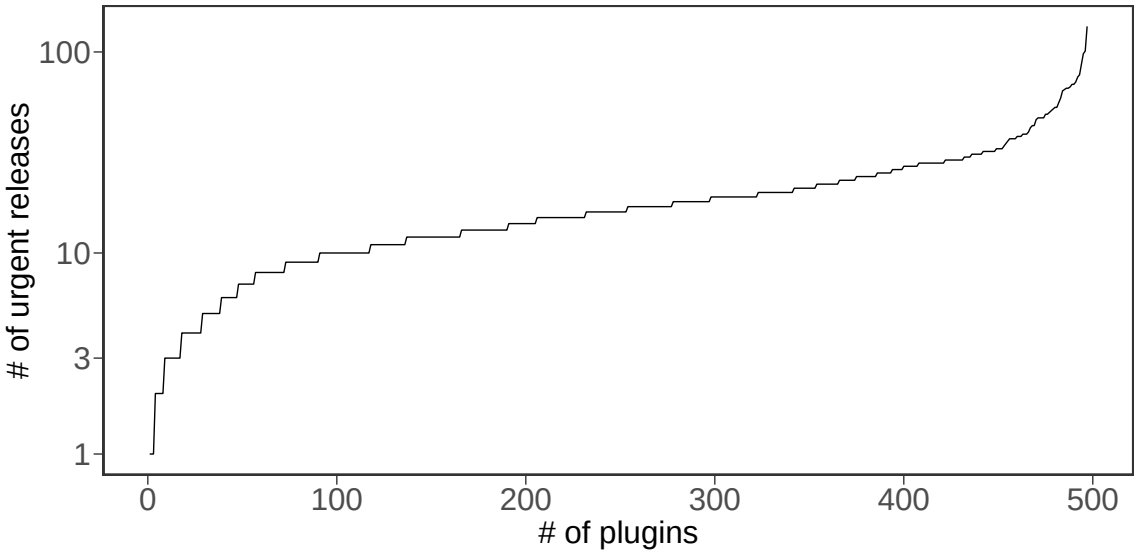


Fig. 2. The distribution of plugins with their respective number of urgent releases. 497 out of the studied plugins have at least one urgent release, while three do not have any urgent release.

Results: 497 (99%) of the studied plugins have at least one urgent release (i.e., a release within 24 hours of its prior release). The studied plugins have a median of 24 urgent releases, as shown in Figure 2. 73 (14%) of the studied plugins have less than 10 urgent releases while 62 (12%) of the studied plugins have more than 30 urgent releases. We also observe that some releases are followed not just by one urgent release, but by a median of 4.5 and up to 18 urgent releases. For example, the *Wp-Easycart* plugin[2] continuously released 14 urgent releases (i.e., from 1.0.8 to 1.0.19) from July 22 to July 24, 2013. The causes of these urgent releases include bug fixes in the plugin, bugs related to WordPress (e.g., notices appearing everywhere when WordPress was in debug mode), and bugs related to other software systems (e.g., deprecated functions in PHP 5.3, did not store data in MySQL). The *Hitsteps-Visitor-Manager* plugin[3] released 5

[1] https://wordpress.org/plugins/12-step-meeting-list/
[2] https://wordpress.org/plugins/wp-easycart/
[3] https://wordpress.org/plugins/hitsteps-visitor-manager/

Table 1. Causes for urgent releases based on extended Swanson categories of changes. Note that one release can address issues across several categories. Incompatibility is the 3^{rd} most common cause for urgent releases.

Category	Issues Addressed	Description	Count
Corrective	Visual	Modifying changes related to user interface. For example, ensuring that RichText links follow the global colors.	130
	Data Processing Logic	Changing the data processing logic in the operational flows. For example, fix the logic of email piping in the multi-site and single-site detection.	88
	Malfunction	Functional errors or processing failures	34
Adaptive	Incompatibility	Fixing conflicts with the platform (e.g., WordPress, browsers), other plugins, libraries, or services (e.g. API).	**69**
Perfective	Performance	Enhancing functionality by improving throughput. For example, reduce the amount of data that is loaded at the initial stage when a page is opened.	9
	Security	Fixing security vulnerabilities or avoiding the leakage of personal sensitive data. For example, a security fix to prevent malicious uploads.	8
Implementation	Improvement	Adding new features or premium service.	30
	Localization	Changes related to languages, regions, or translations. For example, updating the language files.	20
Non-functional	Documentation	Adding or revising meta data, guidelines, figures, or relevant information.	36
	Code Cleanup	Cleaning or refactoring the source code.	11
	Other	-	9

urgent releases to resolve incompatibility issues with *Woocommerce* analytics and the *Contact-Form-7* plugin[4] between June 10 and 15 in 2015.

84% of the urgent releases in our qualitatively examined urgent releases are fixing bugs in their prior release, where the 3^{rd} most common cause of these bugs (32% of the qualitatively examined urgent releases) are related to incompatibility issues. The majority of urgent releases are fixing issues that affect the operation of a plugin and result in failures. For example, an urgent release provides a set of new settings that were missing in prior release, which displayed multiple errors on users' websites. In addition, 9% of the urgent releases are complementing features in their prior releases rather than fixing their prior releases. For example, the *Easyindex* plugin[5] stated that "*Update: Made the instructions for thumbnail preselection clearer*" to guide users on how to use the new features. Note that we are not able to classify 7% of the urgent releases, which do not have any associated release notes or precise commit messages or due to the urgent releases exploiting different classes or functions than the prior release. The 3^{rd} most common issue that plugin developers fix via their urgent releases is related to incompatibility issues, as shown

[4]https://wordpress.org/plugins/contact-form-7/
[5]https://wordpress.org/plugins/easyindex

in Table 1. For example, the *Bns-login* plugin[6] released the urgent release 1.5.1 to fix incompatibility issues with the WordPress version 3.0.1. 32% of these incompatibility issues are between the plugins and WordPress, 19% are between different plugins, 32% are between the plugins and other software systems (e.g., database, PHP), and 26% are between the plugins and third party (e.g., JQuery libraries, Google APIs).

Summary of PQ

99% of the studied plugins release at least one urgent release. The 3^{rd} most common cause for these urgent releases are done to address incompatibility issues. Among these issues, 32% of the incompatibility issues are between plugins and WordPress and 19% are between peer-plugins.

4 UNDERSTANDING THE RELATION BETWEEN WORDPRESS AND PLUGIN RELEASES

Motivated by the observation of our preliminary study, we study when (RQ1) and how (RQ2) the top 500 most-released plugins co-evolve with WordPress and how the plugins co-evolve related to each other (RQ3).

RQ1: When plugins support a new version of WordPress?

Motivation: The goal of this research question is to investigate when plugins support new versions of WordPress, and the impact of the timing of such a support on the quality of the plugin support releases (i.e., releases on which plugins support new WordPress versions). We measure such a quality by the occurrence of urgent releases, as such releases indicate that a bug is introduced in their prior releases, as discussed by prior work and our preliminary study. Our research question sheds light on the dynamics of the relation between WordPress and its plugins.

Approach: To investigate when plugins support a new version of WordPress, we measure the duration of the *"off-sync period"*, which consists of the lag between the release of a new WordPress version and when a plugin releases a version (*"support release"*) that supports that new WordPress version, as shown in Figure 3. As illustrated in the same Figure, we refer to the plugin releases that are shipped within the off-sync period as "*off-sync releases*" and the other plugin releases as "*in-sync releases*". We also investigate whether plugins are consistent with their support of new WordPress versions. For example, a plugin always releases a support release within 7 days after a new version of WordPress. Finally, we investigate when plugins make changes according to the changes of WordPress APIs.

In this research question, we study the plugins' releases as well as WordPress releases. In particular, we use the **Plugin Dataset** (the set of the top 500 most-released plugins) and their associated releases, which are obtained following the approach discussed in our preliminary research question. In addition, we collect all the versions of WordPress including their release dates and notes from the WordPress release page[7] (the **WordPress Dataset**). WordPress had five major versions with each of them having 10 minor versions (e.g., 4.0 to 4.9 for the WordPress version 4) except for version 5, which is the latest major version of WordPress and which has only three minor versions at the time of our data collection. We exclude the WordPress version 1 since it has only 4 active plugins. Note that we do not consider the testing and beta versions of WordPress, such as *"beta 1"* or *"release candidate 1"*. These versions are not official and are just for testing purposes[8].

[6] https://wordpress.org/plugins/bns-login
[7] https://wordpress.org/news/category/releases/
[8] https://wordpress.org/news/2019/10/wordpress-5-3-release-candidate/

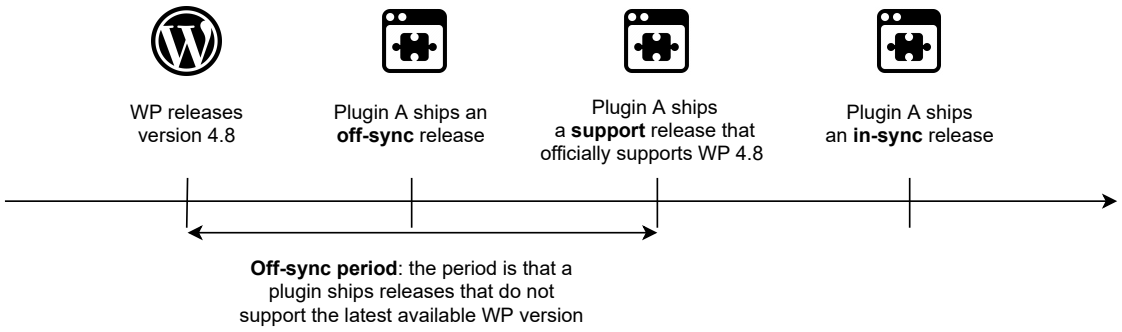


Fig. 3. An illustration of the **"off-sync period"**, which consists of the time between the release of a new version of WordPress and when a plugin supports that new version. We refer to the release that supports a new WordPress version by **"support release"**. During the off-sync period, a plugin can release an **"off-sync release"**, which is shipped by a plugin before it declares support to the latest available version of WordPress. We refer to any release that is shipped after the support of the latest WordPress version by **"in-sync"** release.

Since the duration of the off-sync periods varies from one plugin and WordPress releases to another, we classify the support releases for a new version of WordPress into 4 groups:

(1) **Early adopters** are plugin releases that support WordPress Beta versions.
(2) **Fast adopters** are plugin releases that support new WordPress versions within one week after the official release date of a new WordPress version.
(3) **Normal adopters** are plugin releases that support a new WordPress version after one week and within 60 days from the release date of the new WordPress version.
(4) **Slow adopters** are plugin releases that support a new WordPress version after at least 60 days.

We then compare which of these groups has a higher chance to be followed by an urgent release.

Finally, we investigate when plugins adopt a new or delete a removed WordPress API. To do so, we investigate which type of plugin releases (i.e., in-sync, off-sync, or support releases) change (add or delete) a WordPress API. Then, we investigate the lag between the time when a new WordPress API is added and when it is used by the plugins. Similarly, we investigate when a WordPress API is removed and when it is removed from the source code of our studied plugins.

Results: Plugins support a new version of WordPress after a median of 38 days with a lag of just one release behind the latest available WordPress version. 28% of the support releases are made within one week after the official release date of a new version of WordPress, while 38% are made at least 60 days after a new version of WordPress. Our observation holds for version 4 and its minor versions, as shown in Figure 4. For instance, only 27% of the studied plugins support WordPress 4.0 within the first week and a median of 22% of the studied plugins support minor versions of WordPress 4.0 (i.e., 4.1 to 4.9) within the first week. The lag (in days) for plugins supporting a new version of WordPress is similar to the lag of updating API changes (a median of 34 days) in the Pharo ecosystem, as reported by Hora et al. [21], while it is faster than the lag of API adoption in the Android ecosystem (average 14 months), as reported by McDonnell et al. [27].

97% of the plugins have at least one support release that is made as late as after a week from the release of the latest WordPress release. For instance, only 3% of the studied plugins always support new versions of WordPress within their first week. The studied plugins support some versions of WordPress as early as within the first week,

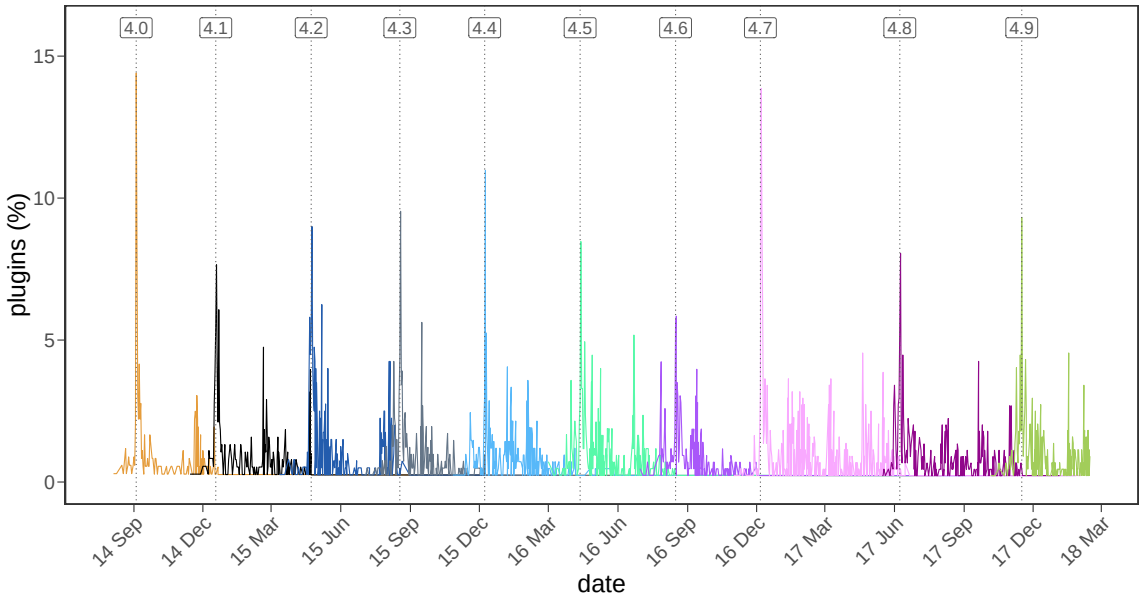


Fig. 4. The distribution of plugins shipping a support release for a new WordPress version. Early spikes (i.e., early and fast adopters) appear across all minor versions of WordPress version 4. The vertical dashed lines indicate when WordPress released a version. Note that this plot considers just version 4 and we observe similar results for the other WordPress versions (i.e., 2, 3, and 5).

while the same plugins support other versions as late as after 60 days. For example, *Woocommerce-Checkout-Manager*[9] shipped a release to support WordPress 4.6 on August 19, 2016, which is three days after the release of version 4.6, while the release supporting WordPress 4.9 was introduced 84 days after its release date.

We do not observe any large differences among the four groups of adopters in terms of the chance to be followed by an urgent release. Early, fast, normal and slow adopters have a median of 0%, 0%, 11%, and 13%, respectively, of their releases that are followed by an urgent release. The percentage of normal and slow adopters that are followed by an urgent release is statistically significantly different (Wilcoxon test: p-value $< 2.2e^{-16}$, $\alpha = 0.01$) compared to early and fast adopters, as shown in Figure 5. However, this difference has a small effect size (0.26). In particular, 478 (i.e., 96%) out of the studied plugins have at least one early or fast adopter, while a median of 0% of their such adopters are followed by an urgent release.

Although WordPress plugins support new WordPress versions, the plugins make changes according to the new WordPress version after a median of 431 days. For instance, even when plugins support a new WordPress version that removes a WordPress API, the studied plugins keep using these APIs that are completely removed from the source code of WordPress for a median of 873 additional days. WordPress removes an API from its source code in a median of 10 minor versions (i.e., 1,320 days as WordPress releases a minor version in a median of 132 days) after the introduction of the API. Furthermore, most of the API related changes are not made by support releases, but by in-sync releases, as shown in Figure 6. For example, *post_permalink()* has been removed since WordPress 4.4[10]. The

[9] https://wordpress.org/plugins/woocommerce-checkout-manager/
[10] https://developer.wordpress.org/reference/functions/post_permalink/

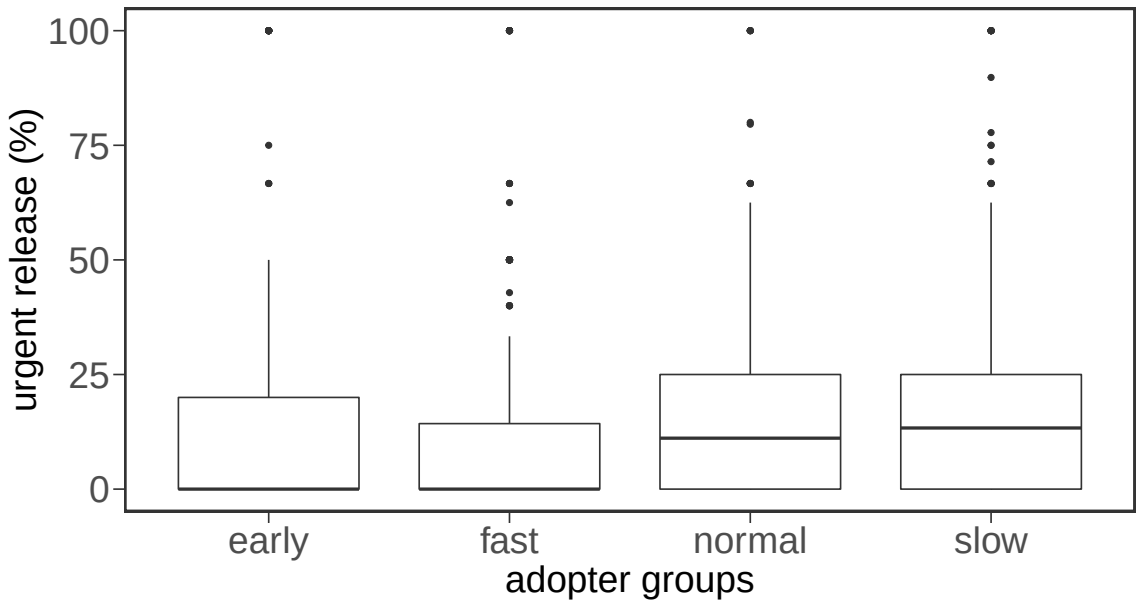


Fig. 5. The comparison of the distribution of urgent release ratio for support releases based on their adoption speed. Slow and normal adopters are more likely to be followed by an urgent release.

photo-gallery plugin[11] supported WordPress 4.4 one day after its release, while the plugin removed the call to the *post_permalink()* API (in an in-sync release) after 671 days. We also observe that newly introduced APIs are used as fast as 1 day from their addition by WordPress. Finally, 25% of the WordPress APIs were never used by any WordPress plugin. We randomly selected 50 APIs from these APIs to further investigate whether they need special settings (e.g., configuration settings) before using them. 76% of these 50 APIs are to manipulate particular information of a tiny element (e.g., get the last name of the author of the current post, determine if a string is well-formatted) on a website. The others (24%) are for administrators to control and maintain their websites, such as getting the size of a directory recursively, handling warnings and notices in each sub-domain of multi-sites. In general, plugins only use these APIs when they support certain functionality (e.g., multi-site management).

Summary of RQ1

Plugins support a new version of WordPress after a medium of 38 days with a lag of just one release behind the latest available WordPress version. Such a lag period is inconsistent even for the same plugin. Moreover, plugin developers use a new WordPress API as fast as 1 day after its introduction and delete removed APIs after a median of 873 days.

[11] https://wordpress.org/plugins/photo-gallery/

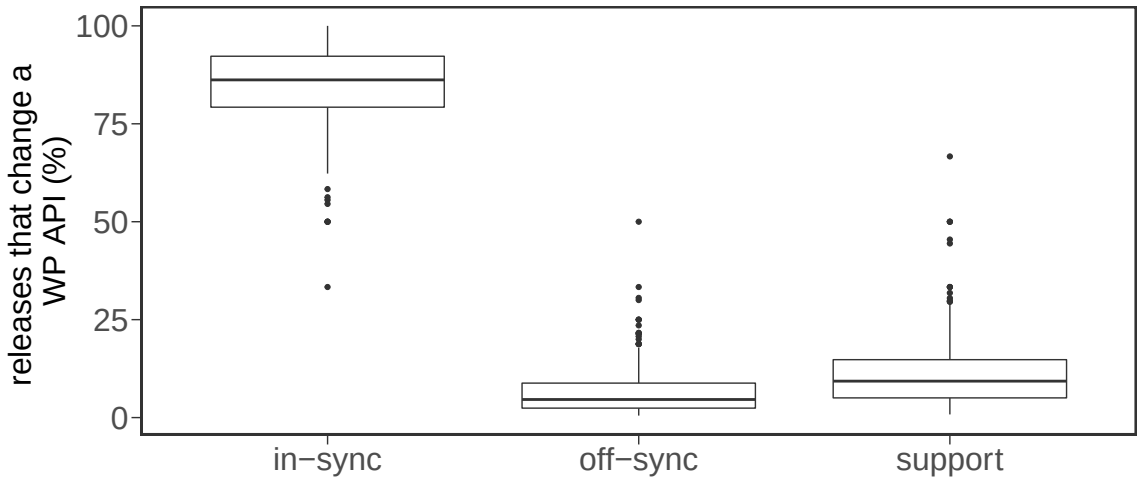


Fig. 6. The comparison of the percentage of releases that change a WordPress API among in-sync, off-sync and support releases. Plugin developers are more likely to make changes related to the associated WordPress APIs in in-sync releases compared to support releases.

RQ2: How plugins support a new version of WordPress?

Motivation: The goal of this research question is to investigate how plugins support new WordPress releases, so WordPress developers can accordingly design tools to help plugins better support new WordPress versions and reduce incompatibility issues. In particular, this research question investigates how plugins support new WordPress versions from three perspectives: we investigate whether plugins release changes (aka., off-sync releases as shown in Figure 3) **before** supporting new WordPress versions, and how likely these off-sync releases will contain issues that will be fixed by an urgent release. We secondly investigate the amount of code changes **during** the support of new WordPress releases. We also investigate **post-support** by studying how likely support releases as well as releases that make a change according to the modification of a WordPress API will be followed by an urgent release.

Approach: To study the practices that are **before the support** of new WordPress versions, we quantify the amount of off-sync releases that plugins have. In particular, we count for each plugin the number of releases that exist between the release of each WordPress version and the plugin's support release for the same new WordPress version. We also compare how often these off-sync releases are followed by an urgent release compared to in-sync releases (i.e., releases that are made after supporting new WordPress versions).

Second, we investigate **the support itself** by studying their code changes. In particular, we compare the amount of code changes of support releases, in-sync, and off-sync releases. We also compare the impact of the amount of code changes on the quality of support releases. The quality is quantified by the chance of a support release to be followed by an urgent release. In particular, we compare the support releases with a large amount of code change (i.e., the top 25% of our studied support releases in terms of code changes) and support releases with a few code changes (i.e., the bottom 25% of our studied support releases). Since plugins do not always use new WordPress APIs or delete removed WordPress APIs in support releases, we also quantify the quality of releases that change (add or delete) a WordPress API. Similarly to the other practices, we measure the quality with the number of urgent releases.

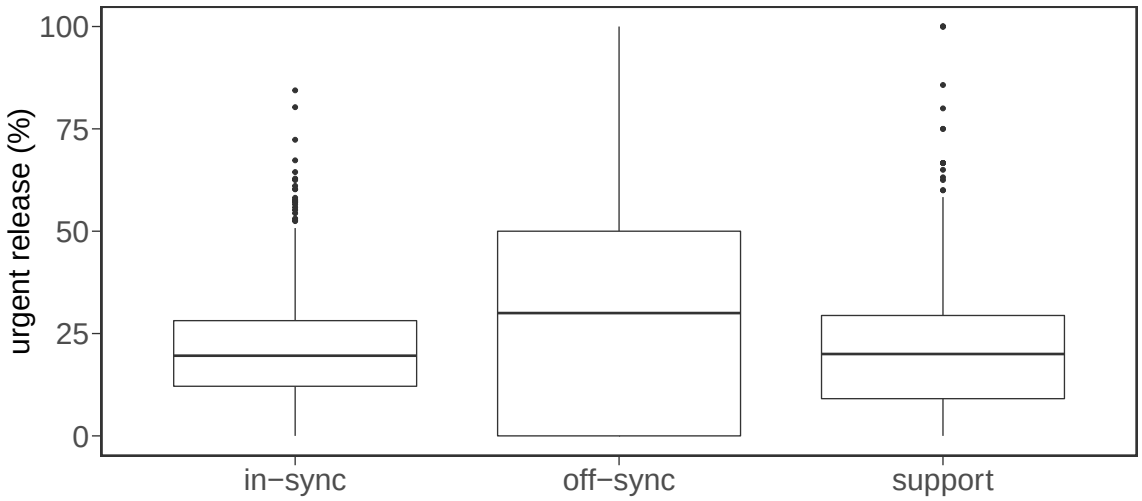


Fig. 7. The comparison of the distribution of urgent release ratio among in-sync, off-sync and support releases. Off-sync releases are more likely to be followed by an urgent release compared to support and in-sync releases.

Finally, we study the quality of support releases in comparison with ordinary releases by measuring how likely such releases will be followed by urgent releases. Similarly, we study the quality of releases that make a change according to a WordPress API (e.g., releases that add a new WordPress API or delete a removed WordPress API).

Result: The majority of the studied plugins (i.e., 89%) have at least one off-sync release. Such releases are more likely to be followed by an urgent release (153%) compared to in-sync releases. The studied plugins have a median of 6 off-sync releases. 31% of these off-sync releases are made within one week after a new version of WordPress, while 17% are made even after 60 days. Figure 7 shows the boxplots that compare off-sync and in-sync releases in terms of the urgent release ratio (the percentage of the releases that are followed by an urgent release for a given plugin). The difference of the urgent release ratio between off-sync and in-sync releases is statistically significant (Wilcoxon test: p-value = $2.07e^{-7}$, $\alpha = 0.01$), with a medium effect size (0.49). An example from the *Caldera-Forms* plugin[12] shows how off-sync releases can introduce incompatibility issues with WordPress. For instance, the plugin had an off-sync release that was made 13 days after the official release of WordPress 5.0. The off-sync release was followed by 4 urgent releases that fix incompatibility issues related to WordPress 5.0, as stated in the release notes of the urgent releases (e.g., *"FIXED: Unable to continue with Freemius opt-in after update to WordPress 5.0."*).

While a median of 20% of support releases are followed by an urgent release, a median of 5% of support releases are urgent releases that fix their previous releases. 271 (54%) out of the studied plugins have at least one of its support releases that is also an urgent release. Figure 8 indicates that 351 plugins have less than 10% of their support releases that are urgent releases, while 2 plugins have more than 50%. For example, the *planso-forms* plugin[13] shipped an urgent release (i.e., 1.3.2) to support WordPress 4.2 due to an incompatibility issue, as noted in the release note of version 1.3.2: "*Checked for compatibility with WordPress 4.2. Changed the dynamic date and time variables to obey the WordPress settings.*"

[12] https://wordpress.org/plugins/caldera-forms
[13] https://wordpress.org/plugins/planso-forms

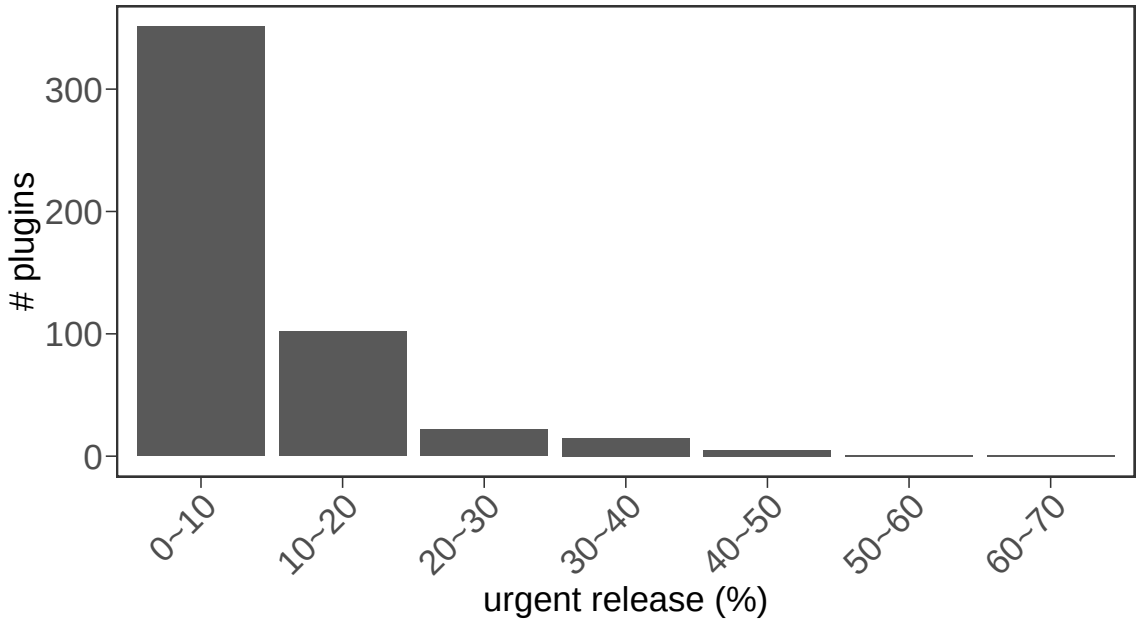


Fig. 8. The number of plugins that have a certain percentage of its support releases that are urgent releases. For example, 102 plugins have 10-20% of their support releases that are urgent releases.

Support releases with a large amount (i.e., top 25%) of code changes are more likely to be followed by an urgent release compared to support releases with a few code changes (bottom 25%). In particular, plugin developers modify a median of 33 lines of code for support releases, while they modify a median of 22 and 13 lines of code for in-sync and off-sync releases, respectively. Figure 9 indicates that the difference of the urgent release ratio between support releases with large changes and few changes is statistically significant (Wilcoxon test: p-value < $2.2e^{-16}$, $\alpha = 0.01$), with a medium effect size (0.6).

Releases with code changes related to WordPress APIs are more likely to be followed by an urgent release compared to ordinary releases (that do not have changes related to WordPress APIs). Figure 10 presents the difference between these two kinds of releases is statistically significant (Wilcoxon test: p-value = 0.005, $\alpha = 0.01$), with a small effect size (0.15). Although plugin developers are more likely to make changes related to WordPress APIs in the in-sync releases (Figure 6), the ratio of such in-sync releases that are followed by an urgent release (i.e., median of 20%) is lower than support releases that have changes related to WordPress APIs (i.e., median of 31%). Moreover, a median of 22% of in-sync releases that have changes to WordPress APIs are also urgent releases that fix their previous releases.

Summary of RQ2

The majority (89%) of the studied plugins have at least one off-sync release. The possibility of such off-sync releases being followed by an urgent release is 153% times higher than in-sync releases. Support releases with a large amount of code changes are risky since they are more likely to be followed by an urgent release.

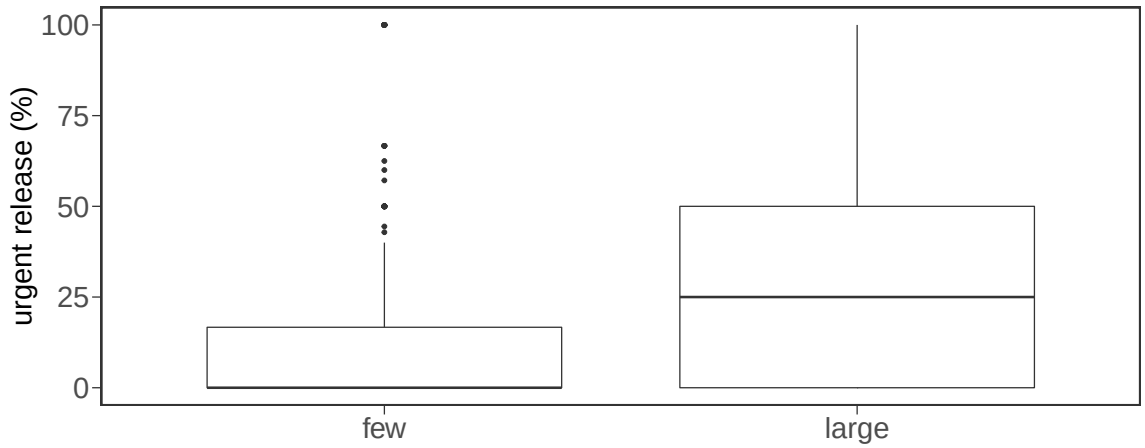


Fig. 9. The comparison of urgent release ratio for support releases with large v.s. few changes. Support releases with a large amount of code changes are more likely to be followed by an urgent release.

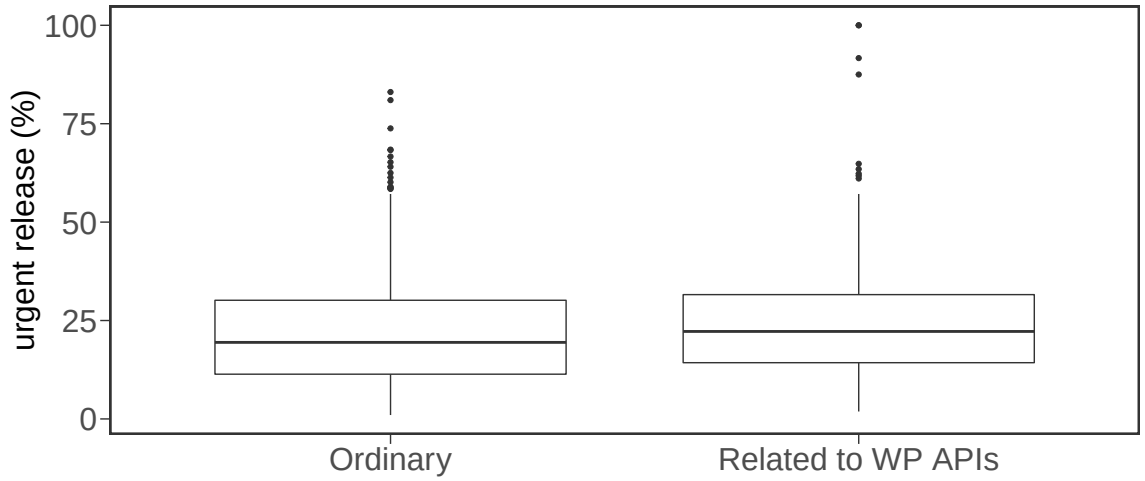


Fig. 10. The comparison of urgent release ratio for releases with a change related to WordPress APIs v.s. ordinary releases (no changes related to WordPress APIs). Releases with changes related to WordPress APIs are more likely to be followed by an urgent release compared to ordinary releases (that not have changes related to WordPress APIs).

RQ3: How WordPress plugins co-evolve?

Motivation: The goal of this research question is to investigate the co-evolution of plugins. Incompatibility issues do not occur solely between a plugin and its platform, but also between different plugins as discussed in our preliminary study. For example, a user noted an incompatibility issue between the *Jetpack* plugin and the *Yoast-SEO* plugin 2 days after the release of version 7.0 of the *Jetpack* plugin[14]. The problem was resolved once the user reverted the upgrade of *Jetpack* by downgrading to the version 6.9. Therefore, this research question investigates the relation between

[14] https://wordpress.org/support/topic/jetpack-7-0-incompatible-with-yoast-seo/

plugins by studying when a plugin releases a change according to another plugin (aka., *peer-triggered releases*), the quality (in terms of following urgent releases) of such a change, and what resources can be shared between plugins so incompatibilities occur. Our findings can provide a clearer understanding of the co-evolution of plugins, so WordPress can accordingly develop solutions that minimize the incompatibilities between WordPress plugins.

Approach: To investigate how plugins release new updates according to other plugins, we identify related plugins from the **Plugin DataSet**, which is obtained from our preliminary study. Since there is no metadata that indicates which plugin is related to which other plugin, we leverage the description of plugins and their release notes to identify related plugins. In particular, we identify that two plugins are related from one plugin's description or from the release notes of a plugin. A plugin is explicitly related (i.e., **explicit relation**) to another plugin when they extend each other's features according to one of the two plugins' descriptions. For example, the *ecwid-shopping-cart* plugin[15] generates shopping carts for users to place their orders on an e-commerce store. The plugin supports multilingual storefronts by leveraging multilingual plugins (e.g., the *polylang* plugin) to translate the information (e.g., name, description) of products. Therefore, the *ecwid-shopping-cart* plugin has an explicit relation with each of the multilingual plugins. In other cases, two plugins might interfere via an **implicit relation** when they are installed together leading to incompatibility issues. For example, Figure 11 shows that there is a relation between the *All-In-One-Wp-Security-and-Firewall* plugin and the *Woocommerce* plugin. We refer to the releases that mention other plugins as *"peer-triggered releases"*. Similar to RQ1 and RQ2, we first measure the quality of peer-triggered releases by calculating the amount of their following urgent releases.

We also investigate a lower-bound time required to fix an incompatibility between plugins. In particular, we investigate the time when plugins ship new releases in response to incompatibilities that were introduced by other plugins. Since we cannot identify for a plugin A that introduces an incompatibility issue that was fixed by a plugin B, we compute the lag between the release of the plugin B that fixes compatibility issues to the closest previous release of the plugin A, as a lower-bound estimation of the time that incompatibility issues between plugins persist.

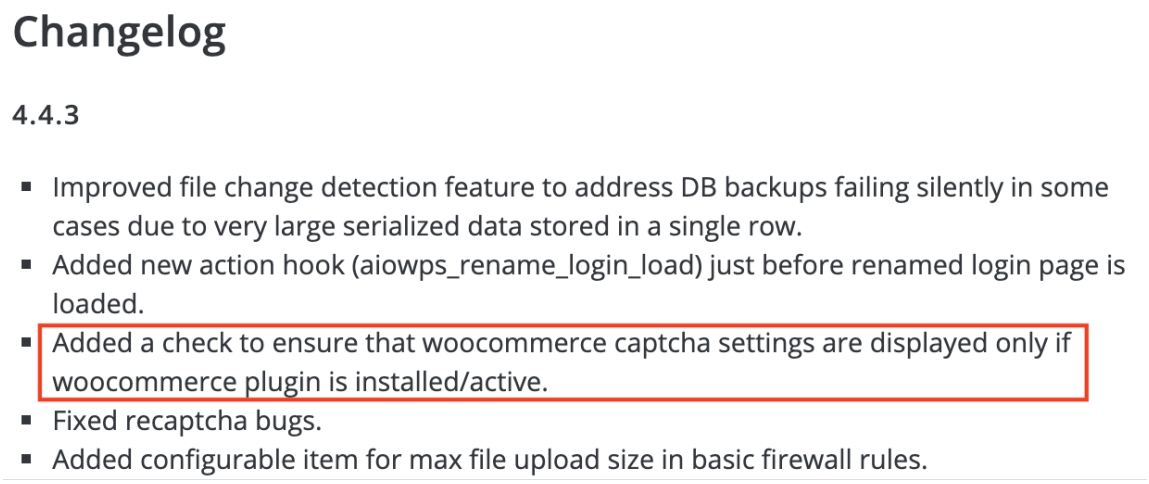


Fig. 11. An example of a pair of plugins, the *All-In-One-Wp-Security-and-Firewall* plugin and the *Woocommerce* plugin, based on the release note of the the *All-In-One-Wp-Security-and-Firewall* plugin.

[15] https://wordpress.org/plugins/ecwid-shopping-cart/

After defining a set of related pairs of plugins, we qualitatively investigate what resources plugins could share leading to incompatibility issues. To do so, we conduct a qualitative analysis for a statistically representative random sample (confidence level = 95%, confidence interval = 5%) of 240 peer-triggered releases to provide a catalog of the rationale for two plugins to be related. Among the 240 peer-triggered releases, we manually label each peer-triggered release that addresses incompatibility issues by investigating the release note of that release as well as its source code changes. We label the main causes for incompatibility issues between WordPress plugins in terms of resources that are shared between plugins. For example, one stated that *"fixed issue setting Featured flag and Catalog Visibility in woocommerce 3.0 and later"* in the release note of version 2.41 of the *pw-bulk-edit* plugin[16], a plugin for editing product metadata (e.g., name or price) on an e-commerce website. We determine the root cause of the incompatibility issue between the *pw-bulk-edit* and *woocommerce* plugins as layout, since the two plugins shared the same elements (e.g., labels, buttons) on a web page. The developers of the *pw-bulk-edit* plugin tweaked the invoked function for retrieving and displaying information in the elements of the web page in version 2.41.

Result: Plugins have a median of 6 peer-triggered releases. The plugins release changes according to a median of 5 other plugins (not all of these related plugins are in the top 500 most most-released plugins). Figure 12 illustrates that the studied plugins are related to one other plugin and as much as 78 other plugins (not all of these related plugins are in the top 500 most most-released plugins). For example, the *Squirrly-SEO* plugin[17] is related to 78 other plugins since it offers smart tools of search engine optimization that collect information from web pages where those plugins are installed to increase the ranking of their websites. In addition, the top three plugins having the largest number of relations with other plugins are *Woocommerce*[18], *Gutenberg*[19], and *Buddypress*[20]. *Woocommerce* is a popular eCommerce solution built on WordPress, *Gutenberg* is a flexible editor for users to create fancy content (e.g. media, photo gallery) easily on a website, and *Buddypress* provides a suite of components to integrate a modern, robust, and sophisticated social network into a website. We also observe that 94% of the pairs of related plugins do not share any developers in common.

Peer-triggered relations do not exist just between the plugins that directly depend on each other. In particular, 60% (144 out of 240) of the qualitatively studied plugins relations are between plugins that do not extend the features of each other or do not directly depend on each other (implicit relations). For example, *Super-Socializer*[21] integrates login, share, and comments from social media into a website. This plugin breaks the features of *Buddypress*, which is a competitor. Hence, the *Super-Socializer* developers have resolved the incompatibility issue and stated that *"[Bugfix] BuddyPress XProfile fields were being wiped out on Social Login, if social profile fields were not mapped to XProfile fields from plugin options"*. On the other side, *Woocommerce-Predictive-Search*[22] is a plugin that extends *Woocommerce* with an auto-complete feature so there is an explicit relation between the *Woocommerce-Predictive-Search* and *Woocommerce* plugin.

Peer-triggered releases are not risky, since peer-triggered releases are less likely to be followed by an urgent release, compared to ordinary releases (that are not associated with other plugins). Figure 14 shows that the ratio of urgent releases for peer-triggered releases is statistically significantly (Wilcoxon test: p-value = $1.66e^{-8}$; $\alpha = 0.01$; effect size is

[16] https://en-ca.wordpress.org/plugins/pw-bulk-edit/
[17] https://wordpress.org/plugins/squirrly-seo/
[18] https://wordpress.org/plugins/woocommerce/
[19] https://wordpress.org/plugins/gutenberg/
[20] https://wordpress.org/plugins/buddypress/
[21] https://wordpress.org/plugins/super-socializer/
[22] https://wordpress.org/plugins/woocommerce-predictive-search

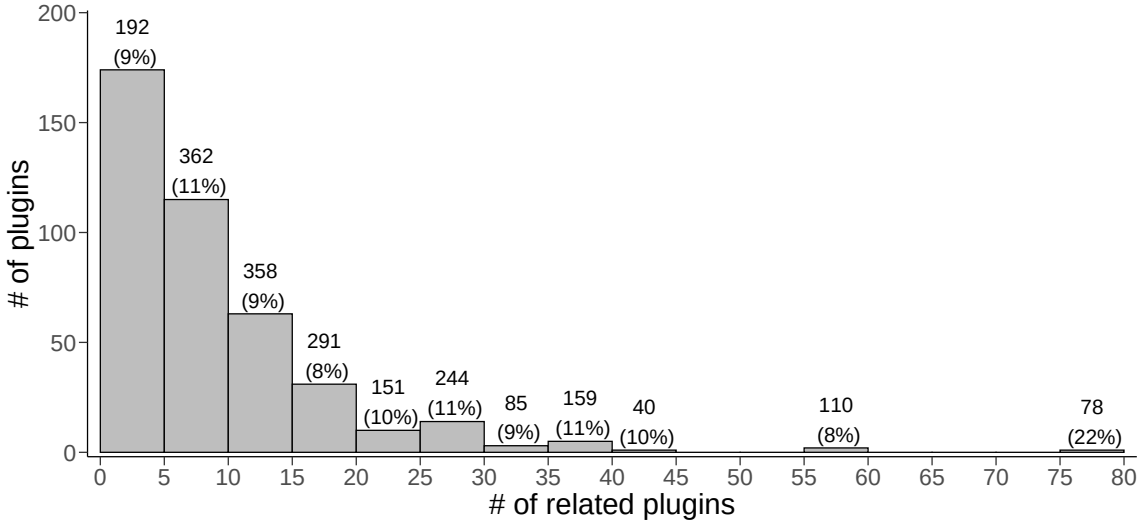


Fig. 12. The distribution of the number of related plugins for a given plugin. The numbers at the top of each bin indicate the unique number of related plugins and the percentage of these related plugins that are in the top 500 most-released plugins.

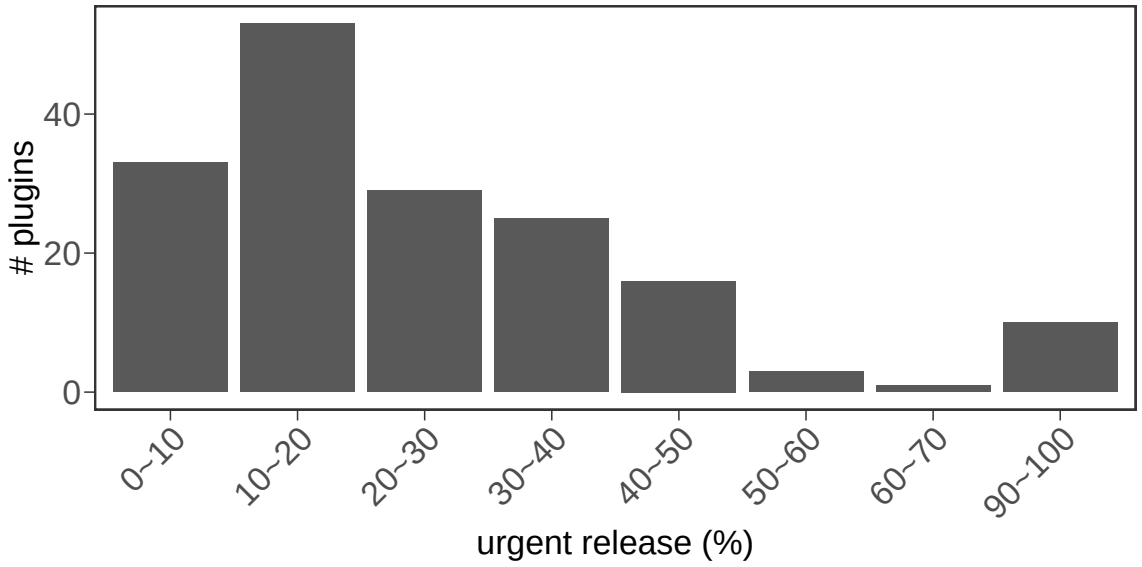


Fig. 13. The number of plugins that have a certain percentage of its peer-triggered releases that are urgent releases. For example, 53 plugins have 10-20% of its peer-triggered releases that are urgent releases.

small (0.21)) lower than the ratio of urgent releases that are associated with ordinary releases (that are not associated with other plugins). In addition, a median of 20% of peer-triggered releases are urgent releases that fix their previous releases. Figure 13 indicates the distribution of plugins that have a certain percentage of its peer-triggered releases that are urgent releases. 170 (34%) out of the studied plugins have at least one peer-triggered release that is an urgent

Table 2. The types of relations that exist between a pair of related plugins. Note that a relation can have multiple reasons.

Reason	Count(%)
Conflicts between plugins	110 (46%)
Leverage features from other plugins to extend its features	80 (33%)
Adjust/Add functions to offer additional settings for other plugins (e.g. options to implement *Yoast-SEO* breadcrumbs)	72 (30%)
Multiple plugins to form a feature (e.g., master-slave)	25 (10%)
Integrated newly created plugins to benefit to users (e.g., merge settings from different plugins together)	23 (10%)
Protective measurements to avoid conflicts or errors with plugins having a similar feature	8 (7%)

release. For example, one mentioned that "*FIXED: Block was not showing in the editor if Gutenberg was not active and WordPress 5.0.1+.*" in the release note of an urgent release for the *caldera-forms* plugin[23].

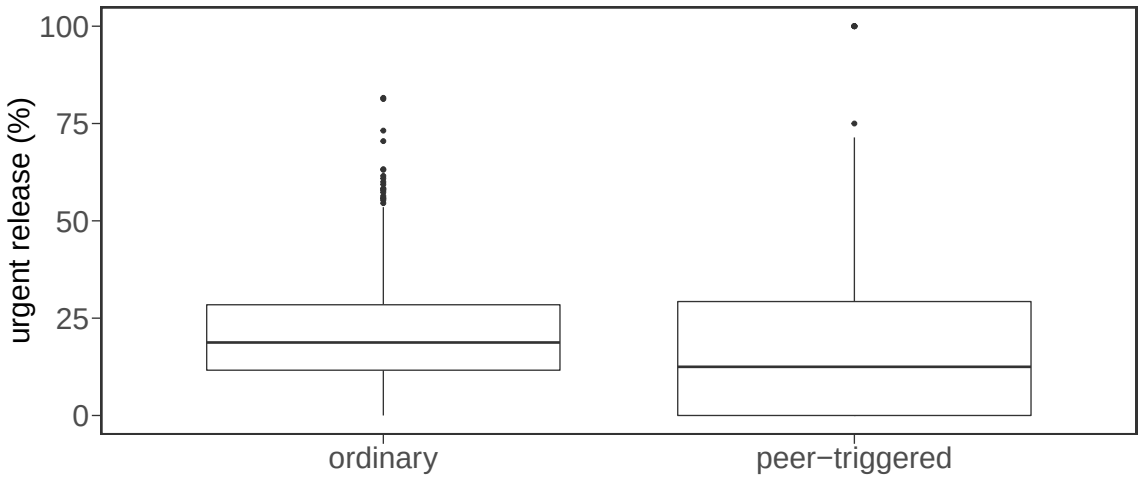


Fig. 14. The comparison of urgent release ratio for ordinary v.s. peer-triggered releases. Peer-triggered releases are less likely to be followed by an urgent release.

The most common goal of peer-triggered releases is fixing incompatibility issues between peer plugins, as shown in Table 2. In fact, 46% of these releases are related to incompatibility issues between plugins. In addition, plugins extend their features by using other plugins, as discussed earlier in the above for the explicit relations between plugins. Plugins also face incompatibility issues in these explicit relations, for which developers know in advance the plugins on which they depend. For example, as stated in the release notes of the *Affiliates-Manager* plugin[24], "*Fixed an issue that was preventing affiliates manager from creating new affiliates during WooCommerce checkout*".

[23] https://wordpress.org/plugins/caldera-forms
[24] https://wordpress.org/plugins/affiliates-manager/

Table 3. The types of resources that are shared between each pair of plugins that show a conflict.

Shared Resource	Description	Example	Reason	Count(%)
Data	Data related objects that are used in the operations of a web page (e.g., rendering, forms) are changed or removed by another plugins	"Fix – *Yoast SEO* was overriding Sunshine's own title tags with wrong data for individual galleries and images, not anymore" [8]	Two plugins (i.e., *yoast-seo* and *sunshine*) change data in the same global variable.	58 (53%)
Workflow	The order of invoking functions is impacted when a plugin is installed with other plugins, such as one function is invoked twice.	"Fixed: If you click on the small icon from the *WooCommerce* product editor to insert shortcode, it shows popup header twice" [4]	Two plugins added two textboxes that are sharing the same css-class "*css-textbox*". One of the two plugin added a javascript handler for the mouse-over on any textbox tagged with the *css-textbox* class style. Thus, the handler was executed for both textboxes, as illustrated in Figure 16a.	45 (41%)
Layout	HTML elements (e.g., buttons, images) are not located at their proper positions in a web page	"Fixed: Input width and box shadow options are not applied on file type input in *Contact Form 7* widget" [6]	One plugin changes the css class of an element that was created by another plugin after certain behaviors (e.g., mouse over), making users cannot interact with the element any more. See Figure 16b for more details.	35 (32%)
Library	A collection of files, programs, and code that assists developers to reduce the implementation time and can be used by different plugins	"Fixed Stripe library conflict if other *Stripe* plugin is installed" [7]	Several plugins include the *stripe-gateway* library in the same web page. When the web page loads the same library at the 2nd time, the variables and functions in the library conflict with the ones that have been loaded by the first plugin.	11 (10%)
Name	The same source code names (e.g., classes, variables, configurations) are repeated across plugins	"Renamed configuration page tab classes to avoid conflicts with other plugins" [2]	Css styles apply to several elements (e.g., labels) on a web page that are sharing the same name but are added by two different plugins.	6 (5%)
Cache	A asset is stored in cache for a web page to enhance the performance of the web page	"Avoid using relative asset URLs which may break caching plugins" [3]	Caching plugins cannot cache the assets without their full urls, leading to errors.	3 (2%)

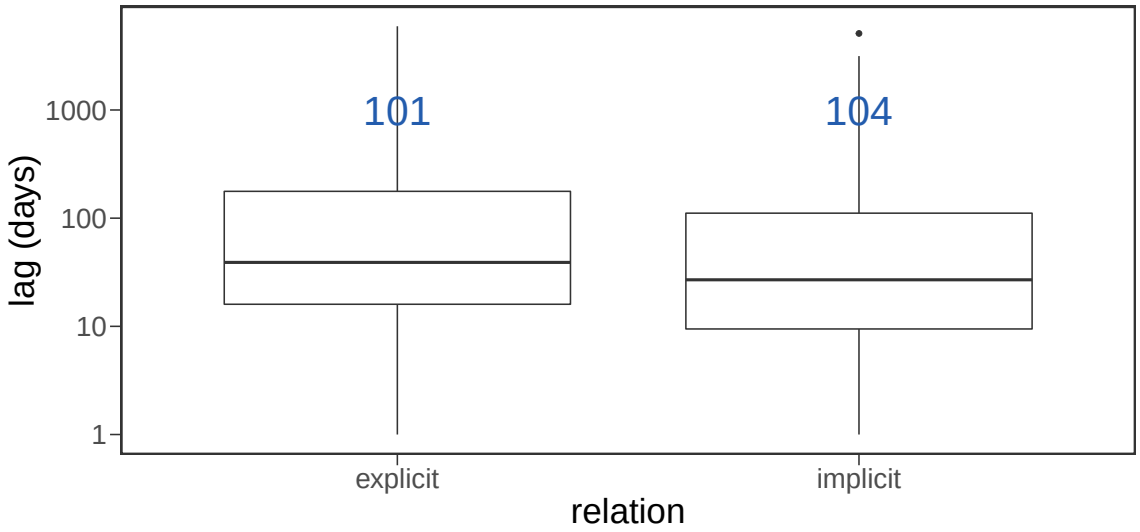
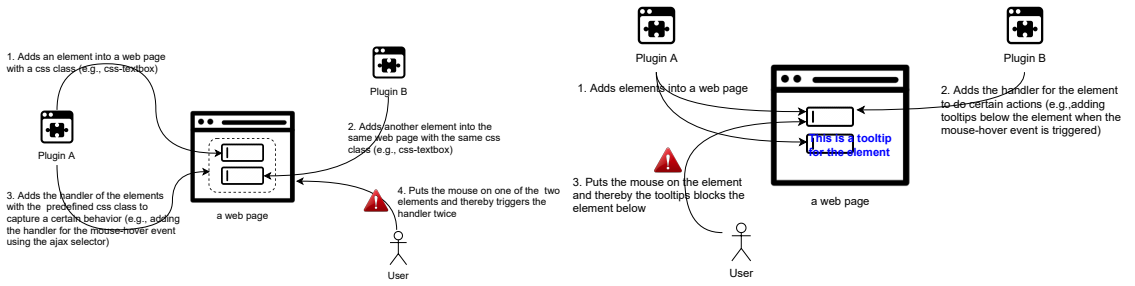


Fig. 15. The incompatibility last (lower-bound estimation) between pairs of plugins with explicit or implicit relations. The blue number above each boxplot represents the number of pairs of plugins are considered in each relation.



(a) The explanation of the conflict between two plugins while sharing **workflow** together (the 2nd example).

(b) The explanation of the conflict between two plugins while sharing **layout** together (the 3rd example).

Fig. 16. The explanation for the examples of sharing resources between plugins in Table 3.

Developers take a long time, i.e., a median of 36 days (lower-bound estimation), before fixing incompatibility issues. Based on our qualitatively studied incompatibilities, we observe that plugin developers take a median of 39 days or 27 days (lower-bound estimation) to fix an incompatibility between a pair of plugins having an explicit or implicit relation, respectively, as shown in Figure 15.

Sharing six types of resources is responsible for incompatibility issues between peer plugins, according to our investigation on the main causes for 110 incompatibilities. Table 3 presents the six types of resources, which are workflow, layout, data, library, name, and cache that plugins share with each other along with examples. While sharing the workflow and layout is expected because plugins work together to form a web page, sharing data, libraries, names, and caches between plugins is surprising because each plugin is designed for different purposes. Even when WordPress

provides a best practices guideline about the naming conventions[25] of these resources to avoid the incompatibilities, plugins still share these resources.

Summary of RQ3

Our studied plugins have a median of 6 peer-triggered releases. The first important goal of these peer-triggered releases is fixing incompatibility issues, which remain unfixed for 39 days and 27 days for explicitly and implicitly related plugins, respectively. The studied plugins have a median of 20% of their peer-triggered releases are urgent releases. These incompatibilities occur when two plugins share at least one of six types of resources.

5 THREATS TO VALIDITY

5.1 Internal Validity

An internal threat to validity is related to the identification of related plugins. For instance, we might learn more possible reasons for incompatibility issues in case we are able to identify all the possible related pairs of plugins. However, our approach that leverages the plugins' descriptions and release notes to identify related plugins allows us to observe that plugins can accidentally interact even if they do not extend each other. Finding more possible relations between plugins will enforce that conclusion. In addition, our approach to identify related plugins allows us to collect a large number of incompatibilities between plugins, which we qualitatively study to identify the possible causes of incompatibility issues. Finding more relations between the WordPress plugins can help identify more possible causes for incompatibility issues. Thus, we encourage future studies to investigate approaches that help identify all possible relations between WordPress plugins.

We also do not generalize our results to the WordPress changes that can be triggered by the plugins. As the plugins are designed to extend the functionality of WordPress instead of the opposite, our study focuses on how the plugins react to the changes of WordPress and whether there is any relation between the plugins themselves. While there is no systematic way to identify whether a WordPress change is triggered by a plugin, we observed certain examples where a plugin developer suggested WordPress certain improvements[26], which was integrated into WordPress after several versions (from 2.9 to 4.4). Thus, we encourage future work to investigate how WordPress reacts to the suggestions made by the plugins.

5.2 External Validity

Similarly to prior studies [16, 24, 29, 30, 34, 35] that focused just on WordPress, we do not generalize our findings to other content management systems (e.g., Drupal and Joomla). However, WordPress is the most popular CMS platform and has been used by more than 42% of the websites whose content management system (i.e., more than 65% of market share) is known [38], compared to other popular CMSs, such as 1.5% for Drupal and 2.1% for Joomla. In addition, WordPress has a structured historical repository that allows us to collect data for analyzing the co-evolution between the plugins and their platform. That is not the case for Joomla, as it does not have any hosted code repositories for its plugins so we cannot analyze the plugins changes according to the changes to the Joomla platform. Drupal has a

[25] https://developer.wordpress.org/plugins/plugin-basics/best-practices/
[26] https://core.trac.wordpress.org/ticket/11334

completely different release strategy for plugins, compared to WordPress. The developers of Drupal plugins specify a version of Drupal when they create a plugin. In other words, each Drupal plugin only works for a particular version of Drupal, which is completely different from WordPress and its plugins. In fact, users can install any version of WordPress plugins on any version of WordPress. Hence, the results we learn from our study might not be applicable to Drupal. That said, Drupal and Joomla have many incompatibility issues between the platform and plugins (e.g., the *config-terms* plugin was incompabilitible with Drupal 9.2[27], the *admin-tools* plugin[28] was incompatible with Joomla 4.0.3[29]) and we encourage future studies to investigate the incompatibilities in these other CMSs. Nevertheless, a large number of posts on incompatibilities between WordPress and its plugins or between plugins with each other are reported on the WordPress support forum, indicating how common the incompatibility issues are[30]. For example, one stated an incompatibility between a plugin and WordPress 5.3. "*When I click on Add New Post, Blank White screen appears.*[31]"

6 CONCLUSION

A plugin-based software ecosystem, such as WordPress, is an ecosystem that consists of a platform that can be extend by additional components, that are called plugins. Differently from traditional ecosystems, the platform and the plugins of a plugin-based software ecosystem have a symbiotic relation. The platform needs the plugins, which in turn require the platform. Such a high coupling between the platform and the plugins increases the chances for incompatibility issues. In fact, users hesitate to upgrade their WordPress and plugins due to incompatibility issues. According to our preliminary study, plugins often miss relations with the platform or other plugins, which cause important incompatibility issues that are as important to be rapidly addressed via urgent updates. 32% and 19% of the incompatibilities that are fixed through urgent updates are between a plugin and its platform and between a pair of plugins, respectively.

In this paper, we conduct an empirical study that investigates how plugins ship releases in a plugin-based software ecosystem, so we can get a clear understanding of how plugins and the platform as well as peer plugins co-evolve. Such understanding can guide future studies and WordPress developers better support such co-evolution by designing mechanisms and evaluating their impact on reducing incompatibility issues. For example, WordPress can design mechanisms for notifying that a new version will be released and which APIs will be removed in the new version. These mechanisms can also warn plugin developers when they accidentally share resources (e.g., unique css classes, variable names) with other plugins. Our findings suggest that (RQ1) even if plugins support the latest WordPress versions, they do so after a long period. Even worse, they make changes according to the WordPress APIs after a long period (i.e., a median of 431 days). During such an-out-of sync period, plugins can manifest incompatibility issues with their platform. (RQ2) Plugins release risky changes that are out-of-sync with WordPress releases, whereas the support releases and especially those with a large source code modification are risky. Finally, (RQ3) incompatibility issues occur not just between plugins that extend each other's features, and these incompatibilities occur when plugins accidentally share six types of resources with each other.

REFERENCES

[1] Online. CASE Studies. https://wpvip.com/case-studies/. last accessed: 2020-05-15.

[27] https://www.drupal.org/project/config_terms/issues/3200627
[28] https://extensions.joomla.org/extension/admin-tools/
[29] https://www.akeeba.com/support/admin-tools/Ticket/35795:extensions-not-compatible-with-joomla-4-0-3.html
[30] https://wordpress.org/search/incompatibility
[31] https://wordpress.org/support/topic/incompatible-with-wp-5-3-gutenberg/

[2] Online. Link Library. https://wordpress.org/plugins/link-library/#developers. last accessed: 2020-05-15.

[3] Online. Page Builder by SiteOrigin. https://wordpress.org/plugins/siteorigin-panels/#developers. last accessed: 2020-05-15.

[4] Online. Photo Gallery by 10Web – Mobile-Friendly Image Gallery. https://wordpress.org/plugins/photo-gallery/#developers. last accessed: 2020-05-15.

[5] Online. Plugin Compatibility Beta. https://wordpress.org/news/2009/10/plugin-compatibility-beta/. last accessed: 2020-05-15.

[6] Online. Premium Addons for Elementor. https://wordpress.org/plugins/premium-addons-for-elementor/#developers. last accessed: 2020-05-15.

[7] Online. Simple Membership. https://wordpress.org/plugins/simple-membership/#developers. last accessed: 2020-05-15.

[8] Online. Sunshine Photo Cart. https://wordpress.org/plugins/sunshine-photo-cart/#developers. last accessed: 2020-05-15.

[9] Online. Urgent: plugin update for 4.7.1? https://wordpress.org/support/topic/urgent-plugin-update-for-4-7-1/. last accessed: 2020-05-15.

[10] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE international conference on software maintenance (ICSM'13)*. IEEE, 280–289.

[11] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering (EMSE'15)* 20, 5 (2015), 1275–1317.

[12] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE'14)* 41, 4 (2014), 384–407.

[13] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 109–120.

[14] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25, 2 (2020), 1458–1492.

[15] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*. IEEE, 2–12.

[16] Laleh Eshkevari, Giuliano Antoniol, James R Cordy, and Massimiliano Di Penta. 2014. Identifying and locating interference issues in PHP applications: the case of WordPress. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. 157–167.

[17] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2014. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*. IEEE, 84–93.

[18] Safwat Hassan, Cor-Paul Bezemer, and Ahmed E Hassan. 2018. Studying bad updates of top free-to-download apps in the google play store. *IEEE Transactions on Software Engineering (TSE'18)* (2018).

[19] Safwat Hassan, Weiyi Shang, and Ahmed E Hassan. 2017. An empirical study of emergency updates for top android mobile apps. *Empirical Software Engineering (EMSE'17)* 22, 1 (2017), 505–546.

[20] Abram Hindle, Daniel M German, and Ric Holt. 2008. What do large commits tell us? A taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories (MSR'08)*. 99–108.

[21] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. 2018. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal* 26, 1 (2018), 161–191.

[22] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR'17)*. IEEE, 102–112.

[23] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering (EMSE'18)* 23, 1 (2018), 384–417.

[24] Ar Kar Kyaw, Franco Sioquim, and Justin Joseph. 2015. Dictionary attack on Wordpress: Security and forensic analysis. In *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*. IEEE, 158–164.

[25] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. 2017. Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering (EMSE'17)* 22, 4 (2017), 2095–2126.

[26] Mircea Lungu, Romain Robbes, and Michele Lanza. 2010. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE'10)*. 309–312.

[27] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance (ICSM'13)*. IEEE, 70–79.

[28] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 84–94.

[29] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature location using crowd-based screencasts. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 192–202.

[30] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 907–918.

[31] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM'12)*. IEEE, 378–387.

[32] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*. IEEE, 215–224.

[33] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. 1–11.

[34] Mohammed Sayagh and Bram Adams. 2015. Multi-layer software configuration: Empirical study on wordpress. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*. IEEE, 31–40.

[35] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On cross-stack configuration errors. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 255–265.

[36] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering (TSE'99)* 25, 4 (1999), 557–572.

[37] Carolyn B Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM'08)*. 149–157.

[38] Web Technology Surveys. Online. Usage of content management systems. https://w3techs.com/technologies/overview/content_management. last accessed: 2020-05-15.

[39] E Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*. 492–497.

[40] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 226–237.

[41] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*. IEEE, 138–147.