Botched Releases: Do we Need to Roll Back?

Empirical Study on a Commercial Web App

Noureddine Kerzazi
ENSIAS, University of Mohammed V-Souissi
Rabat, Morocco
n.kerzazi@um5s.net.ma

Bram Adams
Polytechnique Montréal
Montréal, Canada
bram.adams@polymtl.ca

***Abstract*—Few minutes after a web-based software release, the release team might encounter log traces showing the new system crashing, hanging, or having poor performance. This is the start of the most nerve-wrecking moments of a product's release cycle, i.e., should one run the risk of not doing anything and users losing precious data, or of prematurely engaging the tedious (and costly) roll-back procedure towards the previous release? Thus far, only little attention has been paid by researchers to these so-called "botched releases", partly because of lack of release log data. This paper studies 345 releases of a large e-commerce web app over a period of 1.5 years, in which we identified 17 recurrent root causes of botched releases, classified into four major categories. We then build explanatory models to understand which root causes are the most important, and to explore the factors leading to botched releases.**

I. Introduction

In June 1999, four outages occurred at ebay.com, a well-known online auctioneer [14]. In particular, the web site shut down because of repetitive failures of its database server, the longest of which lasted twenty-one hours and lost the company around $5 million of revenue. In September 30, 2011, Apples MobileMe service[1] has been down recurrently for multiple hours, with email service not working for as many as 75% of its users. Online reactions such as *"Getting MobileMe was probably one of the worst decisions I've made."* were common across fora and social networks.

What both incidents (and many similar ones) have in common, is that despite recent advances in continuous delivery of software systems [5], [1], the deployment and release phases of a large-scale web app still represent critical risks such as the ones above [22], [13]. In particular, few minutes after a web-based software release, the release team suddenly might encounter evidence of the system crashing, hanging, or having poor performance. Despite the limited support for debugging in the production environment and stringent time constraints, the release team must be prepared to react timely and in a systematic way, since they have the responsibility to safeguard their companys profits and reputation.

In practice, such a team has two options. Either it sticks with the version that was just deployed, despite the occurring problems, trying to hold out until the next hot fix release to address the existing issues, or they return ("roll back"[2]) the production environment to the previous release, which did not yet have the encountered issues. Although rolling back seems the simplest and safest approach, one should realize that any changes to the database schema and/or contents made by the new version should be stored somehow until later on the rolled back features are re-enabled. Furthermore, problems like performance issues are likely to persist for a while with the old version, since the roll-back process will have re-initialized memory caches. Hence, right after a release, the release team consisting of release engineers, operations personnel and development team leads needs to make a major decision, basically predicting the trend of crashes, hangs and performance slowdown in the near-term.

Given that the topic of "botched releases" has only rarely been studied in the literature, this paper performs an empirical study of 345 releases of a large e-commerce system in the timespan of 1.5 years. Using version control, work item, release calendar and crash report data, we address the following research questions:

1) How often do botched releases cause crashes, hangs or performance slowdown?

2) Can we build a good explanatory model of botched releases?

3) What are the most important indicators of botched releases?

We hope that future work will build on our findings to create prediction models of botched releases that can help the release team in their difficult post-release decisions. The remainder of the paper is structured as follows. Section 2 provides a motivational example, followed by a discussion of prior work in Section 3. Then, Section 4 presents the setup of our empirical case study, motivation for our statical approaches, followed by a discussion of our findings in Section 5. Sections

[1]MobileMe was discontinued at the end of June 2012. For this outage and others, please visit https://pingdom.com.

[2]The first option, where one waits for a quick follow-up release to resolve the problems, is sometimes called "roll forward".

6 and 7 conclude with a discussion of threats to validity and the conclusion of the paper.

II. Background and Motivational Example

This section provides background related to the general process and tools used by the release team, as well as discusses a motivational example setting the stage for the paper.

A. Release Process

In recent years, the software engineering community has recognized the importance of solid release engineering practices [1], [5]. However, in many companies the release processes still are poorly documented [9], [7]. The lack of a common process description leads to two drawbacks: (1) the responsibilities of the release engineers (REs) are often misunderstood, and (2) team boundaries and the scope of the release activities are not clearly defined. Consequently, coordination breakdowns occur often between the release team and other roles, such as developers, database administrators (DBAs), and testers [12]. This lack of explicit coordination has a direct effect on the software release quality. Figure 1 shows the three general environments REs have to deal with in order to release a new version of their system. We describe in-depth the specific release process related to the context of our study, for more details we refer elsewhere [1].

New features or bug fixes are developed and tested in isolation by virtue of branches within the source code management system (SCM) [20]. These branches are mostly controlled by the development teams and allow them to make changes in source code, configuration files, databases or frameworks without affecting other teams (who are working in other branches). REs merge and integrate the code to the Trunk branch, where the work of different teams is merged together. In time, the features deemed ready are promoted to a PreRelease branch for stabilization, i.e., ironing out defects introduced during earlier merges and integration.

Once the source code has crossed all gates of testing (including automated unit and regression testing), the release team builds packages and pre-compiles the web app before it is pushed to the staging environment. In some companies, these processes are fully automated, while others require manual intervention (e.g., for cross-compilation and dependencies). The staging environment provides a test environment closely similar to production. For instance, it uses a partial replication of real databases, web cache, and relatively the same configuration in terms of frameworks and APIs. Nevertheless, this environment is usually in a separate part of the network, which means there are still differences with the actual production environment, such as different references to third party services and differences in configuration files and databases.

Once the source code has satisfied all tests in the staging environment, the new code is ready for the production environment. The first action taken by the release team is a health assessment of the production server farms and the provisioning process between servers, i.e., the load balancing between servers and caching of data. Then, the actual deployment and

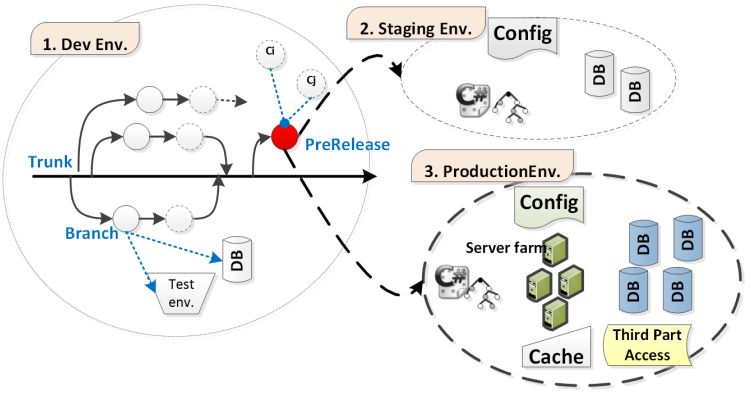


Figure 1: Different environments and interactions in the release engineering process.

Log#	Type	Message	Page	Timestamp	User#	IP	Machine	Select
33228752	Error	Object reference not set to an instance of an object.	CouchbasePageStatePersister-/login.aspx	1/24/2013 5:42:01 PM	0	[redacted]	[redacted]	Select
33228639	Error	StartIndex cannot be less than zero. Parameter name: startIndex	[redacted]payprocess_message.aspx	1/24/2013 5:36:14 PM	0	[redacted]	[redacted]	Select
33228352	Error	Property accessor 'Status' on object '[redacted]Foundation.Entities.Banking.BankAccount' threw the following exception:'Object reference not set to an instance of an object.'	/CSUserPages/CSUserBanking.aspx	1/24/2013 5:23:01 PM	9456132	[redacted]	[redacted]	Select

Figure 2: Reporting the post-release failures.

release activities take place, i.e., coordination with the DBA, pushing of the new build to the production servers, smoke testing (sanity checks) of the newly deployed system, official release to the users and monitoring of post-release failures. For the latter, the REs, supported by the QA team, evaluate the quality of the newly released packages. They mainly look for any large spikes of failures reported in the error logs within minutes after a release. This requires dedicated tool support for collecting and filtering failure reports.

For example, one of the most useful tools developed in-house by the organization under study was a web component for failure reporting (WFR) inspired by the work of Bettenburg et al. [2]. Whenever a failure occurs on the end-user side, the WFR generates a failure report, such as the one shown in Figure 2. For instance, the information about the failure encountered contains the failure time, failure point including the function and line of code that causes the failure, the IP address of the users device, hardware information, production server on which the failure occurred, and stack traces of thrown exceptions (see Figure 3). The latter are required to help the release team diagnose and fix failures rapidly in the production environment.

B. Motivational Example

By means of a fictional scenario (based on real events), we now illustrate the typical challenges experienced by the release team few minutes after a web-based software release. The scenario involves a Release Manager (RM), Quality Assurance lead (QA), and a Dev Team lead (DT), collaborating to release their company's product. RM is responsible for merging/integrating the code from the development teams' source code branches, building the packages, preparing the

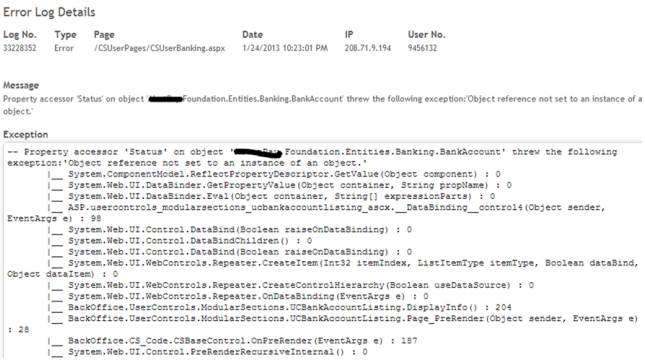


Figure 3: Example stack trace.

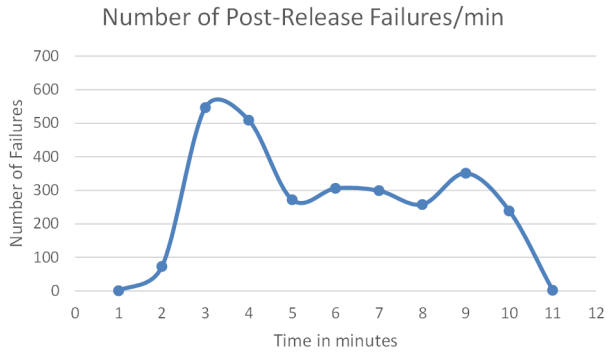


Figure 4: Example crash related to a missed configuration key.

production servers for provisioning, and notifying the DBA to run database-related scripts for the project that is to be released. QA will have to keep monitoring the upcoming post-release failures and to verify that the new functionalities work normally in the production environment (he already tested these functionalities on the development branch, CI system and staging environment). DT is available to help understand the context of potential problems related to the source code.

RM: *Packages are ready, all production servers are healthy, no specific configuration needs for this project, and the DBA is notified to run db scripts. Deploying new packages.*

QA: *Errors start coming related to the checkout module. This module is very sensitive for the business of the company as you may be aware. The situation is deemed critical. Could we rollback?*

RM: *Unfortunately, we cannot rollback because of the db scripts that already have been applied. Lets send the stack trace to DT. Meanwhile, verify whether these failures occur across all servers.*

QA: *Affirmative, the problem is generalized within all production servers...*

DT: *I think we are missing some keys within a configuration file. We have decided to add those keys later in order to allow switching on/off certain functionality.*

RM: *OK, the keys have been added and the provisioning of all servers started. We can indeed observe the post-release failures decreasing!*

Figure 4 shows the impact of this situation in terms of post-release failures. The issue was corrected in three (3) minutes by adding the missing keys within the configuration file. However, one can observe that it took more time to synchronize all production servers, re-initialize their caches and have all customer sessions that were using the flawed code expire.

The example is relatively simple, but it is representative of a concrete crash situation faced by release engineers, in this case caused by developer oversight (a developer forgot to check in changes to a configuration file). At other times, a problem may have been caused by a temporary glitch, a system shutdown or hardware outage affecting a large percentage of users. We introduce the term "botched release" for such problems, for which we provide the following working definition:

> *A botched release is a release experiencing abnormal system behaviour (e.g., crashes, hangs or poor performance) few minutes after being deployed and released into the production environment.*

In all cases of botched releases, the release team ends up having to debug and making changes directly to the production system, or rolling back to the previously installed version. Clearly, neither solution is without risk, but the release team at least would like to know which one is the right decision to take. The ability to detect botched releases early on, therefore, can lead to significant improvements in web-based release practices.

III. Related Work

This section presents related work in the domains of software reliability measurement and defect prediction techniques.

A. Reliability Measurement

Li et al. [15] analyzed software defects after the first year of deployment for two commercial medium-size products aiming to predict field defects and thus to enable more accurate maintenance planning. The study involved 28 releases for two software products. The authors attempted to predict future defects based on the history of metrics regarding product, process, deployment, usage, and configuration. They explored 32 metrics ranging from cyclomatic complexity (CC) to the CMMI level of maturity. However, most of the metrics are too fine-grained to be of use for release engineers, who need to reason at the level of components and releases instead of individual commits, files or functions.

Studies within Microsoft [16], [4], [24] suggested a characterization of differences between pre and post-release versions of a large software product. Based on statistical analysis of defects reported for Beta-Releases, the researchers attempted to adjust field quality predictions for Windows 7 according to usage context and scenarios for the beta version. The authors measured the failure rates during the first 7 active hours of beta version usage within controlled environments. They reported a 59% improvement of the final product quality. Similarly, Song Xue et al. [24] tried to predict system reliability based on beta

usage by measuring failure frequency per day per machine during the first initial 15 calendar days.

Mockus et al. [17] suggested an approach for monitoring post-release failures of a large telecommunication system three months after release in order to assess deployment of both the software and hardware. They found that one month is long enough to get an early indication of software quality and short enough so that a development team can act quickly to take corrective actions if the perceived quality is problematic.

A common characteristic of prior work is that reliability measurement is user-oriented, taking into account the data sent by users and their behaviour. Also, the delay for reliability adjustment takes in between 7 hours to a year. In this work, we explore problems encountered by a web-based release team few minutes after deployment into production servers. In this work, we aim to provide an accurate description of botched releases in an industrial context.

B. Post-Release Failure Prediction

Substantial work has been done on the prediction of software failures. Kim et al. [10] investigated strategies to predict top crashes using machine learning and the history of crashes as reported by Firefox users. The authors attempted to predict whether a crash is a top crash the first time it occurs. Later, they also [11] introduced an approach to predict in advance whether methods run a high risk to crash. This approach then allows focused tests on specific parts of the system.

More recently, Kamei et al. [8] presented a Just-in-Time (JIT) change risk model that identifies defect-prone software changes based on a variety of factors extracted from commits and bug reports. The authors showed on a large-scale system that a JIT model can not only predict whether or not a change will lead to defects, but also allows to reduce the overhead of inspecting all changes (using 20% of the effort to identify 35% of all defect-inducing changes).

Past research has reported several additional factors leading to software defects (e.g., [10], [8], [22], [19]). We investigated a number of control variables according to socio-technical and organizational dimensions in order to build models to explain and predict botched releases.

IV. Setup of Case Study

In order to understand the different kinds of botched releases as well as what factors are good indicators of botched releases, we performed an empirical study on actual field data of a large e-commerce system. This section discusses the setup of the case study.

A. Study Context

The study took place in a large commercial organization in charge of developing a complex financial system. Its web-based and mobile products are used in 192 countries, and its development team is distributed across two sites in Canada and India, with a centralized release team. The system contains over 1,507,291 lines of code organized in 49 major assemblies spread across 8,524 source code files.

We had the opportunity to be on-site with the release team for extended periods of time (more than 20 months), with the first author acting as the *Integrator and Release Manager* for over three years. His tasks included management of releases for six projects (*Main*, *Back-office*, *APIs*, *Mobile*, *Sandbox*, and *Content*), with release cycle times from two months down to multiple releases per day. During this period, he was able to instrument the release engineering tool set to log important data for analysis.

B. Data Set

Our study comprises both qualitative and quantitative analysis. The qualitative analysis was conducted with respect to the guidelines provided by Wohlin [23] regarding empirical research with industry. The first author attended daily meetings including kickoffs, ongoing progress, and release meetings. These meetings, which involved REs, Devs, Testers, and BAs, provided an in-depth understanding of socio-technical contexts surrounding the qualitative data. The observations allowed us to get insights on assumptions behind botched releases, which helped to build the list of candidate factors to be explored. For instance, we were able to analyze checklists used to verify release procedures.

Quantitative data were collected from the release calendar (i.e., a release logbook published on the company's intranet), source code repository (SCM), the collaborative system for recording work item descriptions (feature, bug, etc.), and post-release failures reported automatically to a built-in Crash Report System (CRS). Figure 5 shows our data extraction framework, which is centered around the release log book. This is a database recording all operations performed onto production environments (e.g., placing new packages into servers, changing configuration files, or running scheduled tasks). It is shared on an internal Sharepoint server for the purpose of automatic release notifications and traceability of all changes carried out by release team members, IT, security team, or DBA on the production environment. Log book entries typically include the timestamp of the intervention, brief description of the release content, parts of the system impacted, tags of the source code changes involved, identifiers of features and bug fixes contained in the release.

By complementing the log book with data from more conventional repositories like version control, continuous integration and work item repositories, we came up with a list of metrics that could be linked to different perspectives of botched releases. Based on our qualitative data (in particular our discussions with release engineers) and availability of data for each release (some metrics were only added in later releases), we narrowed down our metric list to those summarized in Table I. We now discuss each of them in more detail, grouped by metric dimension.

Integration - Integration is the process responsible for making new features developed by development teams compatible with new features developed by other teams in the meantime. Teams typically work on a separate branch in the version control system, which corresponds to an isolated space on which one

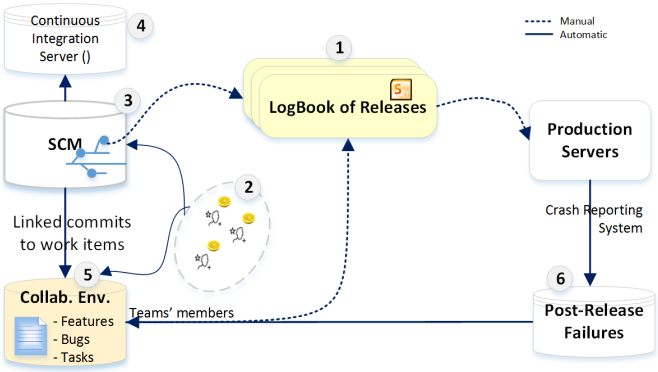


Figure 5: Data extraction framework, centered around the release log book.

can make changes without impacting the work of other teams (in their branches). The way in which code propagates from a team's own branch to the project's master branch and the upcoming release can be indicative of whether that release will be botched.

For example, the larger the number of conflicts (incompatibilities) while merging a branch into master (Magnitude), the less compatible the feature is with existing features, hence the higher the expected chance of botched releases. Furthermore, the more often the compilation and automated tests failed when integrating a branch into master (NBB), the more indications that a team's new feature does not integrate well with the existing system or new features developed in the meantime.

Working too long on a branch without synchronizing with the latest development is the main cause of conflicts. Hence, the more teams synchronize during development (Nsync), the less conflicts would be expected and hence the less botched releases.

History - Apart from compatibility of features, changes during the development process also can be expected to impact botched releases. If a release is developed as planned, according to the initial requirements, one expects more stability in the developed features. However, often plans change during development, in which case unforeseen requirements need to be addressed, which might be less thought out and hence more risky for the upcoming release. The RevWI metric counts the number of such revisions to the work items assigned to a release.

Another indication of change is the number of developers that have worked on a release (NDev), with major features typically requiring more people. Such features, despite being tested and reviewed more thoroughly, require more people to collaborate and synchronize, which is more risky.

Size - Traditional defect prediction models predict the likelihood of commits or files to contain defects that will need to be fixed post-release, unless more tests can be performed on them before release. Since botched releases in some sense correspond to post-release defects, traditional churn (TChurn) and spread (TNF and TS) metrics could be expected to have a strong link with botched releases.

Purpose - Each developer team has its expertise, focusing on certain parts of the system and certain technologies. Similarly, each release has a scope and purpose. The latter refers to the difference between feature releases and for example minor bug fixing or urgent hot fixes. The NatB metric captures this purpose (distinguishing between Feature, Bug, Hotfix, redesign and more trivial Content releases), while the TS metric captures the name of the changed projects (i.e., the 6 projects discussed earlier), since some subsystems are more fragile than others and hence more likely to break the production environment upon release.

Trust - Finally, the release engineer is the final responsible for deciding what is being deployed and released. This is a large responsibility, hence even the release engineer will be cautious about what to release and what not. This is why release engineers need to build a trust relationship with the developers responsible for the code going into the next release. However, release engineers often get burned by trusting unreliable developers, getting the blame of a botched release themselves [21].

To avoid this, each release engineer has a personal system to measure or assess the reliability of a developer's contributions. For example, Chuck Rossi, the release engineering manager of Facebook, has a trust system in which each developer starts with a fresh slate of Karma, only to see it being reduced with each major error [21]. Developers with minimum Karma can only get their code into a release after additional review, since they lack enough trust by the release engineer. In the company analyzed by our study, a similar trust measure exists, counting the number of times a team has broken production before (lemons). We transformed this into a scale from 1 (do not trust) to 5 (highly trusted).

V. Results

This section presents the results of our research questions that aim to explore botched releases and find indications of them based on the 5 dimensions of metrics of Table I. For each question, we first discuss its motivation, the approach followed and our findings.

A. RQ1: How often do botched releases cause crashes, hangs or performance slowdown?

Motivation. To the best of our knowledge, no company has published vexing data about botched releases. In particular, how often do botched releases occur for a modern web application? How many of those releases face crashes or hangs, or suffer performance issues? This question uses descriptive statistics on the data of the analyzed system to provide an accurate picture of botched releases.

Approach. We analyzed 345 releases of different types. We discarded trivial releases such as those related to website content, resource files such as mailing lists, and some core libraries developed in parallel by support teams. This helps us avoid skewing our analysis with releases that did not follow

Table I: Summary of Metrics

Dim.	Name	Definition	Motivation
Integration	#NBB	#broken builds	The more CI builds failed, the more risky a release.
	Magnitude	Integration Effort	The more conflicts between branches of different teams, the more risky a release.
	Nsync	#branch syncs	The more teams synchronized their branches with the master branch before merging, the less conflicts.
History	#RevWI	#revisions to work items	The more revisions have been made to work items, the more changes were introduced during development, the more instability in the release's requirements.
	NDev	#developers changing the code	The larger the team, the higher the chance of failures.
Size	TNF	total #modified files	The more files changed, the higher the chance of problems.
	TChurn	source code churn	The more churn, the more likely failures in production are.
Purpose	NatB	purpose of the release	Releases involving redesign are more likely to end up with a botched release.
	TS	name of modified subsystem	The more subsystems a change is spread across, the more chance to break production.
Trust	#lemons	previous experience with teams	A team that has often broken production is more likely to trigger a new incident.

the normal process of integrating, building, packaging, then deploying and releasing into production servers. Eventually, 320 releases were left.

Findings. 77.5% (248 out of 320) of the releases were completed without problems, while 22.5% (72 out of 320) were botched. Figure 6 shows the causes of botched releases, i.e., what types of artifacts have been changed and caused a botched release. In general, one can observe that the source code is the root cause of 53.3% of the crashes, 35.5% of the system hangs, and 84% of cases of poor performance (e.g., high rate of time-out reported). Hence, surprisingly, except for performance problems, source code is not the sole cause of botched releases. Instead, differences between the characteristics of production and development environment are responsible for most of the crashes and hangs. We explain this in more detail below.

Around 50% of post-release failures that cause system crashes pertain to the source code. In most cases, it is challenging to figure out why the system is crashing due to the absence of debugging tools in the production environment. In this case, the release team attempts to rollback the release when possible. A further analysis of the code fixes produced for system crashing reveals that the amount of lines of code changed is small and 82% of them are related to data processing, for instance, a program trying to parse a null value extracted from the database. Since real data is often obscured to developers for business reasons, specific corner cases causing null values could easily be hidden during development and only pop up after release. To mitigate these issues, we have seen developers adopt simple defensive programming techniques such as testing of null reference before calling methods.

Configuration failures account for almost 20% of system crashes and hangs. Such failures typically occur when a configuration element has been omitted. For instance, keys within a configuration file or bindings to an external service are commonly forgotten during integration. Especially if the production and test environment are not identical in configuration, such inconsistencies easily slip into a release, causing crashes and hangs. Humble et al. [5] recommend to store configuration files into version control, and reusing the same repository in all environments, including development, Test and Production. A tool such as ZooKeeper can help maintain configuration information.

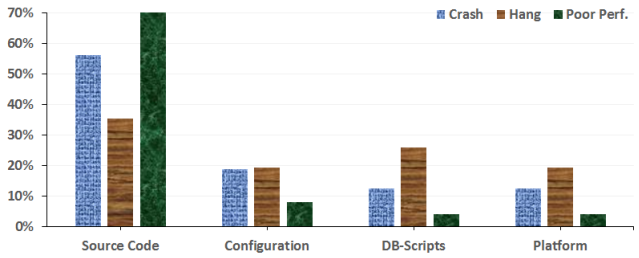
25.8% of hangs and 12.5% of crashes are due to problems with database scripts. Especially for hangs, database scripts are responsible for a large number of problems. If we consider configuration and database scripts as both being related to data processing, 31.3% of crashes and 45.2% of hangs are explained by them. Although source code is responsible for more crashes and performance problems, data processing provides a major alternative cause of botched releases for hangs. This implies that maintaining an explicit coordination between REs and DBAs is as important as technical factors.

The reason for this is that, similar to configuration changes, database changes are made to files that are not always versioned and often are installed in a machine's system directories. Hence, these files are easily forgotten when committing changes. Furthermore, these changes are often necessary to solve small dependency issues related to updated libraries or packages on a machine. Hence, they are more akin to small, but necessary hacks, with developers not noting down (or committing) what they have changed. Finally, developers typically use environments that are different from the test or production system. As such, even if configuration or database changes would have been committed, they might break on the test or production system because of incompatibility.

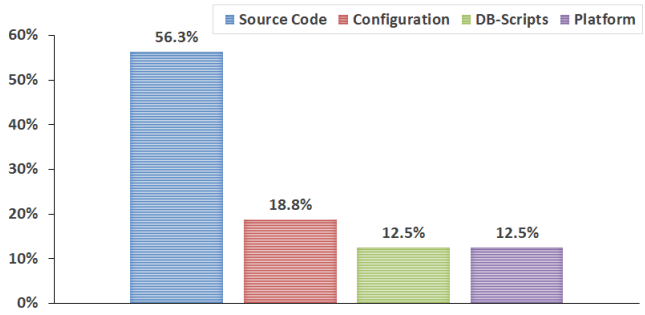
19.4% of hangs and 12.5% of crashes are due to platform changes. Similar observations hold as for configuration and database files. The advent of infrastructure-as-code [5], which allows to specify a platform like a server or virtual machine textually using a programming language, then automatically instantiate it, promises to improve the situation for platform changes. Since these specifications are textual, they can (and should [5]) be checked into version control, and reused on all environments.

B. RQ2: Can we build a good explanatory model of botched releases?

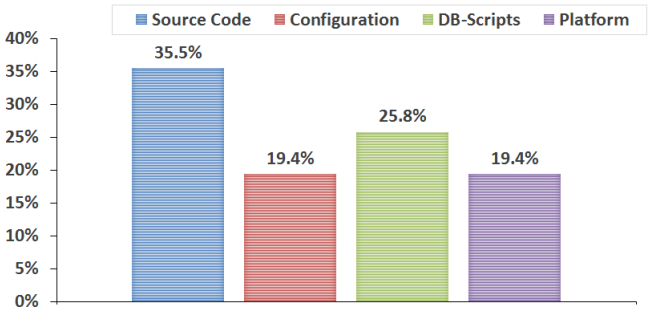
Motivation. In order to understand which factors play the largest role in botched releases, we build explanatory models of botched releases using the metrics of Table I as independent variables. Such a model would help release engineers (and DevOps engineers) in the trenches to be more aware about factors leading to botched software releases, and also help



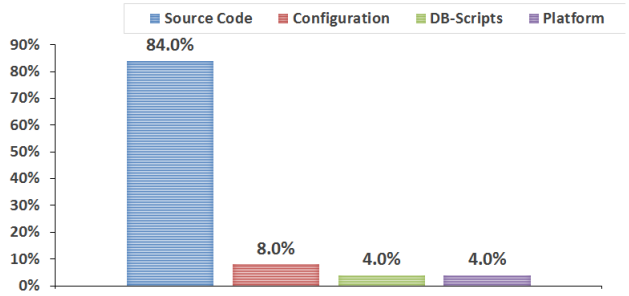
(a) Distribution of Causes of Botched Releases



(b) Distribution of Causes of System **Crashes**



(c) Distribution of Causes of System **Hang**



(d) Distribution of Causes of Poor **Performance**

Figure 6: Distribution of the causes of botched releases (overall and per type of botched release).

researchers to get more insights into modern release issues. Finally, as a practical implication, it would allow enhancing release dashboards with tools monitoring the identified indicators, trying to prevent chaotic releases.

Approach. We build random forest classification models to classify a release as either botched or not. Random forest is a powerful classifier used by many researchers, lending its power from the ability to build multiple decision tree models, then combine those models into one macro-level model. Given that we require concrete values for all metrics, we had to filter our data set of releases from 345 to 231 of which 98 were marked as botched and 133 claimed being successful. This is a relatively small data set, which requires a powerful model like random forest. However, the data is relatively balanced, i.e., roughly the same proportion of botched and successful releases.

We used stratified 10-fold cross-validation, which means that we divided the training set of 231 releases into 10 parts. Then, nine parts are being used to train a model, after which the tenth part is used as test set. This is repeated, such that each part ends up as test set once. The resulting confusion matrix can then be used to calculate precision and recall. Precision is the percentage of releases tagged by the model as botched that really are botched, which gives an idea about false positives. Recall is the percentage of all botched releases that were found by the model, giving an idea about how complete the model is.

To measure how good the model performs in comparison to a random model, we calculated the Area Under the Curve (AUC). This value lies between 0 and 1. The higher the AUC compared to 0.5, the better a model performs compared to a random model. The ROC curve used to compute AUC basically explores all possible values of thresholds used by random forest.

Finally, to determine which metrics have the strongest link with botched releases, we measure the importance of variables in the random forest models using the increased mean square error (%IncMSE). This is the error obtained by comparing the random forest models' predictions with predictions generated using randomly permutated predictor values. If a variable is important in the model, then randomly assigning other values to that variable should have a negative influence on the accuracy of the prediction. In other words, large changes in %IncMSE indicate more important variables, which have a stronger link with botched releases.

Findings. The random forest model obtains a precision of 88.3% (83/94) and recall of 84.7% (83/98). The AUC is equally high, with a value of 0.94. Since testing the collinearity assumptions between our dependent variables did not identify strong correlations (VIF less than 2.64), the high precision and recall values likely are due to the relatively small data set on which the models have been trained and tested. Hence, while we indeed are able to build explanatory models with high performance, we should take the obtained performance with a grain of salt. At least, the performance gives some indication that the metrics of Table I contain some knowledge about botched releases. In order to make stronger claims about this, RQ3 adopts a complementary statistical technique (effect

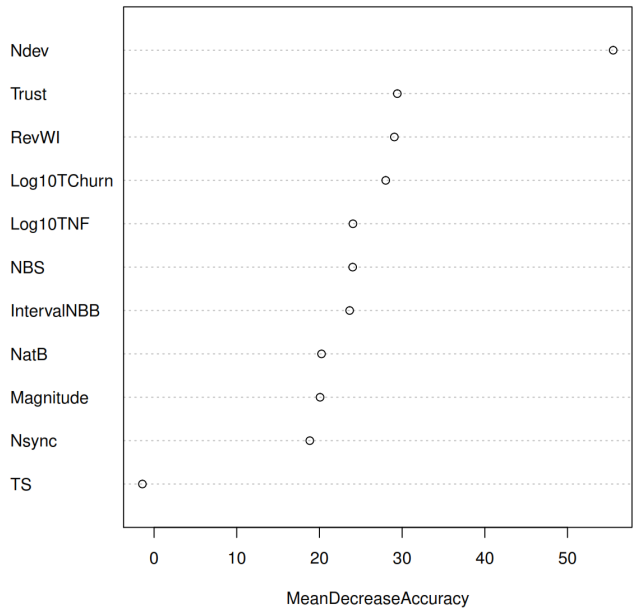


Figure 7: Variable importance for the random forest model.

size) that is robust to sample size.

The number of developers working on a release has the strongest link with botched releases. While RQ3 focuses in more detail on important metrics, here we use the variable importance scores of Figure 7 to obtain an initial idea of essential characteristics of botched releases. NDev clearly is the most important metric in the model, with an importance twice as high as the other metrics. Hence, large teams seem an important indicator of botched releases. This result is in line with existing work showing that a growing team size has a negative affect on productivity, which can be traced back to early studies of the *Ringelmann effect* [6] and *Brooks' law*. Indeed, complex collaborative tasks within a source code branch entail coordination and communication overhead issues that become more likely challenging as the number of developers (NDev) increases. Furthermore, we notice that the trustworthiness of developers in terms of successful releases (Trust), the degree of requirement instability (RevWI) and amount of change to the system by a release (TChurn and TNF along with their log-transformed values) are the next most important metrics.

All of these intuitively make sense, but only Trust and RevWI seem actionable, i.e., one cannot prevent teams from making large changes for a new feature or requiring a lot of personnel. However, requirements could be stabilized better (less changes of work items such as features and bug reports) or developers could be trained to pay more attention when handing over their code for merge or release.

The least important metric by far is the name of the changed component (TS), indicating that botched releases happen throughout the system or without specific preference for a component. The integration metrics in general tend to trail towards the back of the metrics, even though the gap with the next more important metric (NatB) is not that wide.

Metric	df	Mean Square	F	Sig.	Effect Size
TChurn	41	.065	2.604	.001	.672
TNF	12	.127	5.109	.000	.541
Trust	2	.602	24.195	.000	.482
Ndev	6	.122	4.888	.000	.361
NatB	3	.187	7.507	.000	.302
RevWI	6	.049	1.978	.086	.186
NBB	2	.037	1.486	.236	.054
Magnitude	4	.012	.487	.746	.036
Tscode	5	.009	.371	.866	.034
Nsync	3	.014	.560	.643	.031

Table II: Ranking of metrics by effect size.

C. RQ3: What are the most important indicators of botched releases?

Motivation. The initial analysis of important metrics in RQ2 has given some indicators of botched releases. Yet, the relatively small size of our data sample might have introduced noise into our findings. To get more confidence about the factors that lead to botched releases, here we aim to get more accurate recommendations for release engineers, such as a checklist on what to look for before releasing, to aid predicting problematic releases.

Approach. In order to examine how large is the difference between the different characteristics in terms of effect on botched releases, we use the Partial Eta-Squared (PES) [3] statistical method. PES is an established effect size measure in the medical and social sciences literature (where sample sizes typically are small, see e.g. [18]) to compute the proportion of the total variability attributable to a given factor. It expresses the difference between the means of groups in terms of standard deviation units. This statistical method alleviates some of the issues associated with the small size of samples from which our findings are drawn and may thereby protect us from inaccurate interpretations of the statistical significances alone. Moreover, effect size measurement is a standardized number that we can compare to effect sizes of other studies regardless of detailed knowledge of their measurement scales. The meaning of effect size varies by context, but the standard interpretation offered by Cohen (1988) is: 0.012 = small, 0.034 = moderate, and 0.138 = large. Table II summarizes the effect size for our metrics.

Findings.

Size-based metrics have the largest effect sizes. TChurn has an effect size of 0.672, while its fellow Size metric TNF has an equally large effect size of 0.541. TChurn was also recognized in the random forest model as one of the most important metrics, which provides more confidence in our findings. In other words, features with large, spread-out changes are more risky for botched releases. As mentioned before, though, not much can be done to prevent such botched releases, except for *(1)* reducing the scope of releases (making them smaller), *(2)* or requesting more effective tests within source code branches (e.g., nightly regression tests). This is indeed what continuous delivery [5] aims to do.

Trust has the third highest, large effect size. This again confirms our findings for the random forest model, where developers' trustworthiness in terms of delivering solid code

that will not backfire in production is found to be strongly linked to botched releases.

The History metrics also have large effect size. The number of developers (0.361) and requirement instability (0.186) again play a major role, although NDev was the number one metric for the random forest model. Since a large Partial Eta-Squared effect size starts from a value of 0.138, in practice an effect size of 0.361 is not that different from 0.672, since both are considered huge in terms of Cohen's interpretation. We prefer absolute values rather than labels.

The final metric with large effect size is the nature of a release. NatB's effect size of 0.302 hints that some kinds of releases tend to break more easily. In particular, we found that redesign releases were the most failure-prone. Such releases correspond to re-architectures (large refactorings) of the system made by software architects to improve performance or future maintenance. Hence, they tend to break the system in unforeseen ways, especially for branches dedicated to source code redesign.

The remaining metrics only have a moderate effect size. Except for NatB, the five least important metrics of the random forest model coincide with those having moderate effect size. In other words, we can be confident that the number of broken builds (NBB), amount of integration effort (Magnitude), amount of synchronization before merging (Nsync), and the changed location (TS) do not matter that much for botched releases.

Especially the low effect size for the three Integration metrics is surprising, given the amount of effort required to integrate features into the master branch or resolve merge conflicts. Indeed, it looks like:

> *What matters more is not how code flows into a release, but rather the amount of development work that went into it, the kind of release and who was responsible for it. Of these factors, only the last one can really be controlled or at least improved. Requirement stability can also be controlled (if care is taken), but has a lower impact than the other factors.*

VI. Threats to Validity

In this study, we examine the factors leading to botched releases. We simply report observations from an industrial context without claiming causation, hence there are no threats to internal validity.

A. Construct Validity

Construct validity threats are related to: (1) preprocessing of data for *RQ1*; and (2) the metrics we have chosen to build our model for *RQ2* and calculate effect size for *RQ3*. First, data is gathered from different sources. Mapping data from the logbook of releases to source code, to the collaborative environment, and to the crash system report (CSR) is not a trivial task. Because of the imperfection of the traceability approach across different data sources, our study might suffer from linkage bias.

However, the rigorous process for traceability and responsibility in place makes it easier to institutionalize the internal information. For instance, filling the logbook after each release is an internal requirement stated by all departments that need to know what is going on (e.g., customer support, training, top management, and so on). Recording source code tags and identifiers for features and bug description is a requirement for both the release team and Business Analysts to be aware about the content of each release. Finally, the SCM manager strictly enforces the linkages between the change sets and work items (e.g., Features and Bugs).The hard part is related to the way of persisting, in a compact way, the flow of post-release defects reported by the CRS few minutes after each release.

Second, metrics chosen to build our explanatory model of botched releases are based on the experience of the release engineers with whom we discussed the list of metrics. There is no reason to believe that the results are systematically biased.

Finally, the relatively small sample size yielded abnormally high fit of the random forest model. However, crosschecking the variable importance metrics with those based on effect size showed that both approaches roughly identify the same metrics as most important. This gives us confidence into the model and effect sizes, despite the small sample size.

B. Conclusion Validity

Conclusion validity concerns possible violations of the assumptions of the statistical tests. Factors identified as linked to botched releases depend on the specific context and culture of the company. For instance, development teams use extensive parallel development based on a strategy of source code branching, which might add an overhead for integrating the source code.

C. External Validity

This study used data gathered from one industrial context, which might not be enough to claim generalization. However, we provide more insights by long term observation and following an action research method. Hence, the goal of our study was not to generalize results observed in one specific context, but instead we are generating a body of knowledge about the factors that limit the software release practices from a set of observations obtained through a large industrial project.

Further studies are needed for more evidence, and we plan to replicate the study in other industrial and Open Source Software contexts to resolve those threats.

VII. Conclusion

In this study, we explored the factors leading to so-called botched releases, i.e., releases that trigger a high rate of crashes, hangs or performance problems immediately after introducing changes in the production environment. We explored the prevalence of such releases in an industrial system, then tried to better understand what are the causes of these botched releases. Then, we built explanatory models and calculated effect sizes to determine ***what are the main factors leading to botched releases***.

Surprisingly, we have seen that integration factors do not play a major role. Instead, the size of the changes that went into the release, the trustworthiness or seriousness of developers regarding handing over their features for release, and the instability of requirements turn out to be the main factors. Unfortunately, only trustworthiness and requirement instability are actionable, unless practices like continuous delivery are being used to reduce the scope, and hence the amount of work, going into a release.

A second surprise is that source code is not the major artifact causing botched releases. Instead, configuration and database scripts are more commonly linked to hangs and crashes. This means that research should spend more time on those kinds of files, in particular on ways to synchronize these files with source code as well as to keep track of changes to them. Recent advances in infrastructure-as-code are a good first step, but especially for changes in database schema only few research exists to date. Better yet, we have observed that even when crossing rigorous testing gates, each software release is on probation and not fully trusted.

Finally, organizations constantly introduce new indicators, such as analysis of regression tests results, code review interactions, instant messaging analysis, and even sentiment analysis. Hence, we make no claim that the factors explored here cover the space of all possible situations (they do cover the basic data sources every release engineering infrastructure should have). Although our study covers a total of 10 factors related to release engineering and botched releases, we also make no claim that one study can fit all contexts. Hence, more studies in industrial contexts are required. We hope that our exploration will stimulate further research on that topic and support practitioners to better understand release practices.

ACKNOWLEDGMENTS

The authors would like to thank Sandeep Sandhu (QA Team Lead), Wilton Godinho (Development Team Lead), and Xavier Francis (BAs Team Lead) for their useful input.

REFERENCES

[1] B. Adams and S. McIntosh. Modern release engineering in a nutshell – why researchers should care. In *Leaders of Tomorrow: Future of Software Engineering, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[3] P. D. Ellis. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results, 1st Edition*. Cambridge University Press, 2010.

[4] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[5] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[6] A. G. Ingham, G. Levinger, J. Graves, and V. Peckham. The ringelmann effect: Studies of group size and group performance. *Journal of Experimental Social Psychology*, 10(4):371 – 384, 1974.

[7] M. Kajko-Mattsson and F. Yulong. Outlining a model of a release management process. *J. Integr. Des. Process Sci.*, 9(4):13–25, Oct. 2005.

[8] Y. Kamei, E. Shihab, B. Adams, A. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, 39(6):757–773, June 2013.

[9] N. Kerzazi and P. Robillard. Kanbanize the release engineering process. In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 9–12, May 2013.

[10] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Trans. Softw. Eng.*, 37(3):430–447, May 2011.

[11] S. Kim, T. Zimmermann, R. Premraj, N. Bettenburg, and S. Shivaji. Predicting method crashes with bytecode operations. In *Proceedings of the 6th India Software Engineering Conference*, ISEC '13, pages 3–12, New York, NY, USA, 2013. ACM.

[12] I. Kwan, A. Schroter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *Software Engineering, IEEE Transactions on*, 37(3):307–324, May 2011.

[13] A. Lahtela and M. Jantti. Challenges and problems in release management process: A case study. In *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, pages 10–13, July 2011.

[14] N. Levy. Ebay auction site restored after fourth outage in a month. http://downrightnow.com/, June 1999.

[15] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 413–422, New York, NY, USA, 2006.

[16] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko. Characterizing the differences between pre- and post-release versions of software. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 716–725, 2011.

[17] A. Mockus, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 225–233, 2005.

[18] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. D. Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Transactions on Software Engineering*, 37(2):161–187, 2011.

[19] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press.

[20] D. Perry, H. Siy, and L. Votta. Parallel changes in large scale software development: an observational case study. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 251–260, Apr 1998.

[21] C. Rossi. Moving to mobile: The challenges of moving from web to mobile releases. Keynote at RELENG 2014, April 2014. https://www.youtube.com/watch?v=Nffzkkdq7GM.

[22] H. Seo and S. Kim. Predicting recurring crash stacks. In *Proceedings of the 27th International Conference on Automated Software Engineering*, ASE 2012, pages 180–189. ACM, 2012.

[23] C. Wohlin. Empirical software engineering research with industry: Top 10 challenges. In *Conducting Empirical Studies in Industry (CESI), 2013 1st International Workshop on*, pages 43–46, May 2013.

[24] S. Xue, P. L. Li, J. P. Mullally, M. Ni, G. Nichols, S. Heddaya, and B. Murphy. Predicting the reliability of mass-market software in the marketplace based on beta usage: a study of windows vista and windows 7. Technical Report MSR-TR-2011-2, Microsoft Research, January 2011.