The JDownloader Immune System for Continuous Deployment

Thomas Rechenmacher
AppWork GmbH
rechenmacher@appwork.org

Dirk Riehle
Computer Science Department
Friedrich-Alexander-University
Erlangen-Nürnberg, Germany
dirk@riehle.org

Michael Weber
Tradebyte Software GmbH
michael.weber9@gmail.com

Abstract

Continuous deployment can reduce the time from a source code change to a newly deployed application significantly. Increased innovation speed can make all the difference in a competitive market situation. However, deploying at high frequency requires high speeds of discovering bugs in the deployed software. Using the JDownloader file download manager as our example, we present a fitness model to evaluate a continuously deployed software during operation for expected behavior, present the design and implementation of a monitoring component, and evaluate the model and its implementation using data from JDownloader's multi-million member strong user base. Our evaluation finds that there had been thousands of undetected bugs, and that newly created bugs can be detected and reported 16 times faster than before.

1. Introduction

Continuous deployment (CD) is the process of deploying software continuously and automatically into production. JDownloader (JD) is a file download manager with more than 20 million users. Since 2012, the JD development team has been using CD to roll out new features, bug fixes, and other changes to application users at high frequency. In 2013, a custom build system and an incremental update service launched. These services are able to build, test, and deploy to production a change to the version control system (VCS) within 5-10 minutes. It then can take less than 20 minutes until an end-user benefits from a bug fix committed to the VCS.

The code of the JD application can be split into a core, developed and maintained by company Appwork GmbH, and more than thousand plugins, developed and maintained by an open source community. Only Appwork employees can build the core system.

The plugins give JD its main functionality. A plugin parses a website to find a download link. If something changes on the website, a plugin may stop working. CD is used to run a fully automated plugin build, test, and deploy cycle whenever a developer commits changes to the VCS. Until 2012, detecting a malfunctioning plugin, generating a bug report and notifying the developer was a slow and manual procedure.

For CD to work, it is paramount that bugs put into production are recognized as fast as possible. The old mechanism of determining and reporting bugs was unsuited to the fast development cycle that the new build system enabled. This article therefore presents the JD "immune system", a monitoring system and its underlying concepts that drastically cuts the time needed to detect anomalous behavior in production. The contributions of this article are the following:

1. Presentation of a fitness model to evaluate the current state of a system
2. Design and implementation of a software that implements this model
3. Evaluation of the model usin g data from JD's multi-million user base

After reviewing related work in Section 2, in Section 3 we present a mathematical model for a fitness time series that can be used to detect anomalous behavior. Section 4 then details the design and implementation of this model in the JD application. Section 5 evaluates this model using data gathered from operating it with real users for several months. Finally, Section 6 presents conclusions and outlines future work.

2. Related Work

2.1. Continuous Deployment

Continuous deployment (CD) emerged in the web application domain. Changes can be made without user interaction. Companies like Amazon measure time between deployments in seconds [29]. The impact of continuous delivery on web applications has been analyzed at Rally Software [17], Facebook and OANDA [22].

The increased frequency of software releases enabled by continuous deployment methods has sparked broad interest. Systematic literature reviews have been published on the topic [13] [19] [10]. Regarding software quality, these reviews report mixed results.

Mobile applications, notably Android applications, deal with a multitude of hardware devices, operating system versions, geographic differences, network

URI: https://hdl.handle.net/10125/64545
978-0-9981331-3-3

providers, and network failures. The application testing scenario is comparably complex. In contrast to web- or cloud-based solutions, updates cannot be deployed without the user's explicit permission leading to multiple versions installed on customer devices that must continue to work as intended. Also, updates can only be distributed via the operating system provider's tools (iOS App Store, Android Play store). This makes the update frequency not only dependent on user acceptance but also on the diligence of those providers.

In a case study of mobile software deployment at Facebook, Rossi et al. confirm that CD practices do not negatively impact software quality [20]. Crash rates, the number of identified critical issues and the number of fixes necessary after the creation of a release branch have remained constant or decreasing despite shortening release cycles from four weeks to two weeks (iOS) or one week (Android). Other studies [14] suggest that more frequently updated apps correlate with higher ratings in Google's Play store.

We found little research on continuous deployment in open source projects, perhaps because most projects have a clear separation between the source code and any (of many) deployed versions of the software. One survey by da Silva et al. [25] finds that while open source projects frequently release, in there large sample of projects there was only one with "continuous flow", which is similar to continuous deployment.

Most non-open-source examples emphasize the importance of testing through automated pipelines and manual tests. Facebook contracts a manual test team of roughly 100 people [20]. This approach is not feasible for many types of projects and companies. Some apply methods to test applications with real users either through opt-in alpha and beta programs or by using techniques like canary releases or dark launches [22] [3]. Jiang et al.'s work on the economics of public beta testing shows beta tests have a positive impact on software quality as well as market success through word-of-mouth effects [9]. Rodriguez et al. identify user involvement through canary releases and dark launches as an area with a distinct lack of research [19].

2.2. System health monitoring

Ghosh et al. define self-healing as "[…] the property that enables a system to perceive that it is not operating correctly and, without (or with) human intervention, make the necessary adjustments to restore itself to normalcy. Healing systems that require human intervention or intervention of an agent external to the system can be categorized as assisted-healing systems" [4]. Following this, we consider the JD immune system to be an assisted self-healing system.

For JD, system recovery requires human intervention, which is why we focus on literature on failure detection. Early research includes a statistical model for predictive failure detection [7] and a proposal for anomaly detection through gradually relaxing invariants [6]. Gross et al. provide a framework for detecting software anomalies; they identify four crucial aspects for such a system: Data management libraries, statistic modeling tools, corrective action strategy support tools, and an adequate software architecture [5]. Ivan et al. describe a self-healing system for a mobile application where the architecture is similar to the one we designed for JD [8], while Kumar & Naik extend the model towards autonomic computing [11]. Moran et al. continue the research on self-healing for mobile systems and provides strategies for monitoring Android applications to discover and report application crashes [16]. Sahasrabudhe et al. describe application performance monitoring as a sequence of four steps: Monitor, analysis, recommendations, and action [21]. They present a case study of their model showcasing the use of dashboards showing information to application developers. Their notion of availability as a metric corresponds well to the notion of "application fitness" we apply to the JD example. Chen et al. and Ye at al. provide a more recent application of self-healing techniques in cloud software [2] [31].

Suonsyrjä et al. point out that collected post-deployment data can not only be used for error detection but also for creating a feedback loop regarding customer satisfaction with the software [27].

Silva describes four approaches to detect errors in deployed software systems [26]. Systems-level monitoring, failure detection at the application layer, error detection by log analysis, and remote detection of user failures. However, no consistent definition of abnormal behavior is given. In this paper, we build on techniques originally developed for network monitoring by J.D. Brutlag [1] and Evan Miller [15] using the Holt-Winters Forecasting algorithm. Szmit & Szmit summarize more applications of this algorithm for anomaly detection in network monitoring [28]. Sharifi et al. show how neural networks can be used to predict failures in web applications [24]. Wang & Wan develop a self-healing model for systems of systems using stochastic differential equations and Brownian Motion [30].

3. A Fitness Model

This section introduces a mathematical model to track JD plugin behavior over time and to assess it. Details beyond this section can be found in our technical report [18].

3.1. Anomalous Behavior

JD consists of a software core that is extend by about 1,230 so-called "hoster plugins". A hoster plugin

is a software extension (of the core) that can download files from one or more specific hosting service. The core does most of the work, but for downloading from a given website, the core calls out to the matching hoster plugin. Some hoster plugins are able to download from multiple web services. To date, there are roughly 2,800 different variants. The interface between the core and the plugins is always the same.

A JD hoster plugin exhibits anomalous behavior, if the plugin does not work as intended. Usually, the file download does not work. To a certain extent this is normal. Many external issues may lead to a failed download. Web servers may be down, firewalls may block requests, Internet Service Providers (ISP) may block access, man-in-the-middle applications may modify the loaded resource and thus break the plugin's parser, or the requested resources may not be available in the region they were requested from.

However, a plugin may also stop to work due to a bug, for example, in the plugin's parsing function for a website. This can happen, for example, if an external website changes or if a developer commits faulty code. Many of these plugin issues cannot be found by automated tests:

- Many plugins require user interaction (e.g. solving a CAPTCHA challenge, or entering a password)
- About 40-50% of all downloads require a paid premium account. Buying all these accounts for testing reasons would be too expensive.
- Downloads in free (not paid for) mode are often restricted. For some services, the user has to wait for up to 24 hours between two downloads.

3.2. Fitness function

Tracking anomalous behavior (logging plugin failures), creates a time series of successful and failed file download attempts. A time series can be split into intervals of different length. We use an observation interval of 1 hour. By tracking each download attempt and its result, usage and error values can be calculated for each interval.

$$usage = \sum_{t=0}^{1\,hour} download\_attempt_t \quad (1)$$

$$errors = \sum_{t=0}^{1\,hour} failed\_download\_attempt_t \quad (2)$$

Captured as a time series, both values show typical network metrics regularities: A trend over time, a seasonal cycle, a seasonal variability, and a gradual evolution of these regularities [1]. We next merge the *usage* and *errors* value to a single fitness value. The fitness values are calculated by dividing the amount of *error* events by the *usage* events at the given point in time *t*. If there is no known usage at point t, the fitness function is undefined and its calculation is skipped.

$$fitness(t) = 1 - \frac{errors(t)}{usage(t)} \quad (3)$$

This does not only reduce the metrics to a single value, but also eliminates all of the regularities above. The resulting time series has the following properties:

- The value is a percentage between 0% (worst) and 100% (perfect): A perfect fitness series has a constant fitness of 100%.
- **General high variability.** The usage may change spontaneously due to external influences like network problems.
- **Higher variability for unpopular plugins.** Some plugins are rarely used. The volume might be too small for a stable series.

3.3. Threshold Considerations

Detecting an error means detecting a significant drop of the fitness value. Brutlag's approach [1] of predicting a confidence band estimates an upper and lower boundary of acceptable fitness values for each point in time. This is not necessary, because there is only an error if the fitness value is below the lower boundary. Therefore, we do not use a confidence band, but rather a threshold that defines the lowest acceptable value.

We found that a static threshold is impracticable. If a plugin detects that its parser is out of date, it should throw a "Plugin Defect Exception". If there is a network issue, the plugin should terminate with a different error type. Unfortunately, plugins often don't do so, resulting in tracking the wrong error code. This is why there are plugins that work perfectly well at an average fitness value of 50%, and others that have a "perfect level" of 90%. As described in Section 3.1, errors are normal up to a plugin-specific value, and thus affect the "perfect level". We therefore need automated detection of the "perfect-level" for each plugin.

If we were to rely on the "perfect level" as a threshold, we would get many false positives caused by external problems. An unstable web service interface, for example, often results in short-time drops of the fitness metrics. To avoid false positives, a second threshold is used. It's based on an Exponential Moving Average (EMA) trend line. In contrast to the "perfect-level", this "trend indicator" slowly follows a long-term trend and is able to detect sudden trend changes even if we are already below the "perfect level".

This leads to three states:

- **Normal.** The plugin works well. Fitness is above "perfect level" and the trend indicator.
- **Problematic.** There might be a problem. Fitness is below the "perfect level" or the trend indicator.

- **Anomalous.** There is a problem. Fitness is below the "perfect level" and the trend indicator. If an anomalous state is ignored for a long time, the state will change to "Problematic", because the trend indicator will handle the error as a trend instead of an exception.

3.4. Algorithm definitions

Several algorithms are required for the model.

3.4.1. Moving Average (MA)

The MA algorithm in Equation 4 is the simplest algorithm to smooth a data series. It assigns each point in time *t* the average of all values in a time range of t_{MA}. t_{offset} can be used to shift the range in time. A t_{offset} of 0 means that all points are before or equal to *t*. *x* is the time series to smooth, e.g. *fitness(t)*, *usage(t)*, *errors(t)*.

$$ma_{t_{MA}}^{t_{offset}}(t) = \frac{1}{t_{MA}} \sum_{t_{rel}=-t_{MA}}^{0} x(t + t_{rel} + t_{offset}) \quad (4)$$

3.4.2. Moving Variability (MV)

The MV algorithm in Equation 5 extends the MA. It generates the MA for each point in time, and the average relative deviation of the current value to the MA over the time frame t_{MA}.

3.4.3. Exponential Moving Average (EMA)

The EMA algorithm is used several times in the model. In contrast to the simple MA, the EMA can be considered as a weighted moving average that uses an exponential function to give higher relevance to newer events. Because the calculation of *ema(t+1)* always requires *ema(t)*, this is an incremental algorithm. The model parameter β defines how fast the relevance of old values decay. Again, *x* is the time series to smooth, e.g. *fitness(t)*, *usage(t)*, *errors(t)*.

$$ema(t+1) = \beta \cdot x(t) + (1-\beta) \cdot ema(t) \quad (6)$$

$$\beta = \frac{2}{t_{EMA}+1} \quad (7)$$

Like in Equation 4, t_{EMA} can be seen as the number of hours passed, which are used to calculate the next prediction. This leads to the formula in Equation 8.

$$mv_{t_{VA}}^{t_{offset}}(t) = \frac{1}{t_{VA}} \sum_{t_{rel}=-t_{VA}}^{0} \left| x(t + t_{rel} + t_{offset}) - ma_{t_{VA}}^{t_{offset}}(t + t_{rel}) \right| \quad (5)$$

$$ema_{t_{EMA}}(t+1) = \frac{2}{t_{EMA}+1} \cdot x(t) + \left(1 - \frac{2}{t_{EMA}+1}\right) \cdot ema_{t_{EMA}}(t) \quad (8)$$

Because the EMA algorithm is incremental, we need a start condition for $ema(t_{-1})$. Thus, we simply define $ema(t_{-1}) = x(t_0)$, and decrease β in each round *r*, leading to Equation 9. The downside of this start condition handling is that smoothing slowly gets better from 0% for $r=1$ to 100% for $r >= t_{EMA}$.

$$\beta = \frac{2}{min(t_{EMA}, r) + 1} \quad (9)$$

An example is shown as Equation 10.

$$\beta_0 = \frac{2}{1+1} = 1; \beta_1 = \frac{2}{3}; \beta_2 = \frac{1}{2}; \beta_3 = \frac{2}{5}; \ldots; \beta_{t_{EMA}} = \frac{2}{t_{EMA}+1} \quad (10)$$

3.5. Trend Indicator Line

Several steps are required to get a trend indicator line (TIL). This section introduces the underlying model. It consists of the following steps: *f(t)* → *ma(t)* → *ema(t)* → *til(t)*.

1. Starting with the fitness time series *f(t)*, we first use the MA algorithm from Equation 4 to smooth the time series *ma(0, t_{MA}, t)*. The model parameter t_{MA} has to be defined empirically and should be higher for low usage values to increase smoothing, if there is not enough data to get a stable series.
2. Afterwards, the EMA algorithm from Equation 8 performs another smoothing round, giving more weight on the latest fitness values series *ema(t_{EMA}, t)*. Like before, the smoothing model parameter t_{EMA} should be derived from the average usage of the plugin. Because this EMA-step is a long-term trend line, t_{EMA} should cover several days.
3. Finally, the allowed deviation from the *ema(t_{EMA}, t)* trend line is subtracted from the trend line. This deviation should be derived from the average fitness value and the variability at that point in time. This approach tries to eliminate the high variance in the variability described in Section 3.2. Fitness values above 90% are considered as fine, and values below 55% indicate an error.

3.6. Perfect Level Detection

Figure 1 shows the fitness time series during a three-month measurement period in 2014.

The Perfect Level Detection (PLL) algorithm finds the highest density of fitness values. Figure 1 shows a high density of fitness values between 6,000 (60%) and

7,500 (75%). The PPL then is derived from the lower edge of this band. It consists of the following steps: *f(t) → ema(t) → DF(ema(t)) → ma(t) → til(t).*

1. At first, the raw fitness series is smoothed. An EMA with a dynamic model parameter prepares the series for the next steps.
2. Based on *ema(t)*, a Frequency Distribution Function (DF, Equation 11) is aggregated.

$$DF\left(ema\left(t\right)\right)=\sum_{i=1}^{n}1\left\{x_i=ema\left(t\right)\right\} \quad (11)$$

   This is best explained with an example: The goal is to get a function *DF(f)* for *0% <= f <= 100%*. *DF(0)* is the count of all intervals in which *ema(t) = 0*. If there are 10 intervals with a fitness value of 10, then *DF(0) = 10*. To simplify the following "hill" detection, *DF(f)* should get smoothed radically. An *MA* with a model parameter of 4% and an offset of 2% is used. Without the offset, the MA would shift the "hill". Figure 2 presents the smoothed *DF(f)* of Figure 1.
3. Thanks to the smoothing, it is now easy to find the "hills" (level bands) by looking at the gradient of *DF(f)*. To get sharper edges to the next level band, we cut all *DF(f) <= 15%* of *DF(f)*.
4. The perfect level threshold is defined as the lower boundary of the perfect level band - 5%. The *pll(t)* function is the perfect level threshold over time. Again, *pll(t)* is limited to *55% <= pll(t) <= 90%*.

3.7. Anomalous Behavior Trigger

To throw an event as soon as there is a problem, and as soon as the problem is fixed, an $ema_{dynamic}(t)$ series is derived from the raw *f(t)* series. The model parameter for the EMA algorithm has to grow the lower the average usage of the plugin is. This results in higher smoothing if the variability is high due to too few events. The $ema_{dynamic}(t)$ is compared to the *pll(t)* and the *til(t)*. Figure 3 shows a trigger point that triggered an "error occurred" event on March 20th, and a "bug fixed" event on March 8th, 2014.

4. Design and Implementation

The model was implemented as the "StatServ System" for the JDownloader 2 BETA version. During data collection, there was an average of about 850,000 logged events per hour. StatServ is written in Java. Details beyond this section can be found in our technical report [18].

4.1. Old Feedback Loop

Before StatServ, a user had several options to report problems: A community board, a live chat, and a support desk. All these are either managed by moderators from the support community or by Appwork employees. They read the posts, emails and chats, and try to gather as much information as possible about a bug. In most cases, the user is asked to provide a "JDownloader Application Log", a stack trace, test URLs, environment details, or any further bug related information.

As soon as a supporter validated a bug and collected enough information, they submit a bug to the JD bug tracker. Developers are meant to find all relevant information in the bug report. It often happens that a developer has to track down the bug report to the user in order to get more information. This process was slow and needed to be improved. The goal of the implementation is not only to find bugs, but also to provide full bug reports to the developer community.

4.2. The New "StatServ" Approach

The new design can roughly be separated into six components. Each JD installation sends log entries and error details to the "StatServ Collector" (SSC) service hosted by Appwork. The SSC service saves the data to a storage device. The "StatServ Evaluator" (SSE) application reads the data from storage, calculates the charts according to the model and maintains an issue for each found problem in the bug tracker.

4.3. Data Sources

A "StatsManager" module sends all log messages to the SSC service. Two types of data collection approaches are implemented: (1) Manual user reports and (2) fully automated reports.

4.3.1. Manual User Reports

A user can click a "Report a Problem" menu button whenever they experience a download related problem (Step 1). An overlay window will appear. This overlay follows the mouse cursor and informs the user whether they can report the currently selected item (Step 2). Another click collects all information about the position, and sends the resulting "Download Feedback Log Entry" to the StatsManager (Step 3).

4.3.2. Fully Automated Reports

In addition, the StatsManager logs every download attempt and its result directly after the plugin's routine returned. In contrast to manual reports, which can be initiated at any time, significantly more detailed information is available at this point in time, because the full plugin process is still in memory. Compared to the manual "Download Feedback Log Entry" attributes, the "Download Log Entry" contains more information.

4.4. The "StatServ Collector" Service

The central "StatServ Collector" service (SSC) can handle a large volume of Log Entries sent per second.

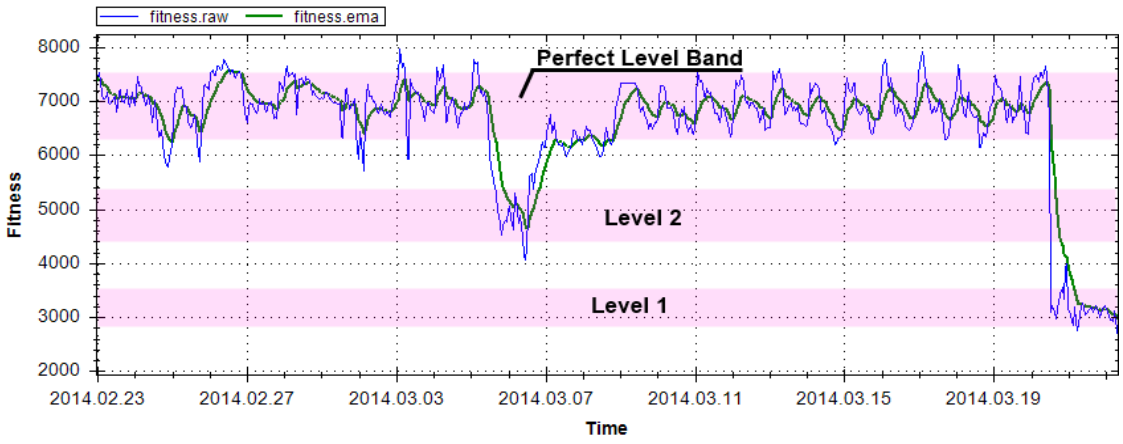

Figure 1. Fitness time series from measurement period in 2014.

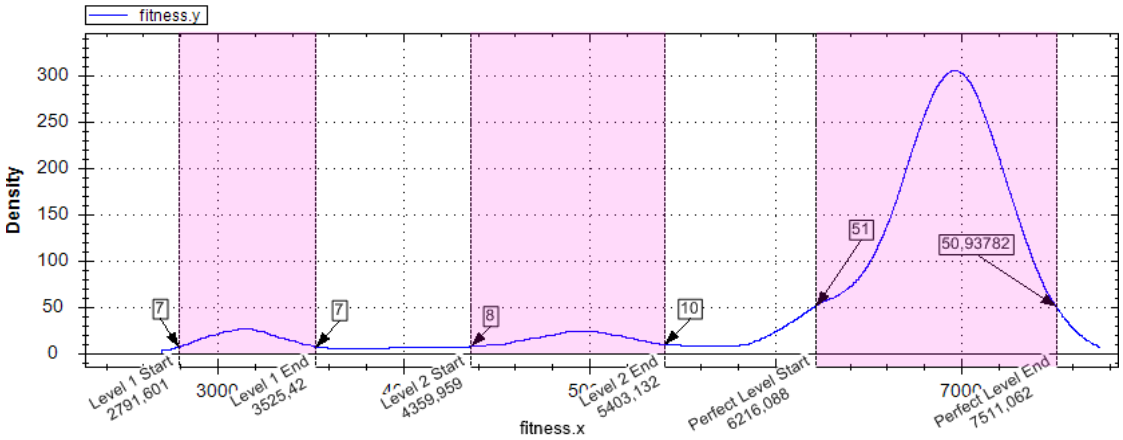

Figure 2. Smoothed distribution function with “perfect level” threshold at 6,216.

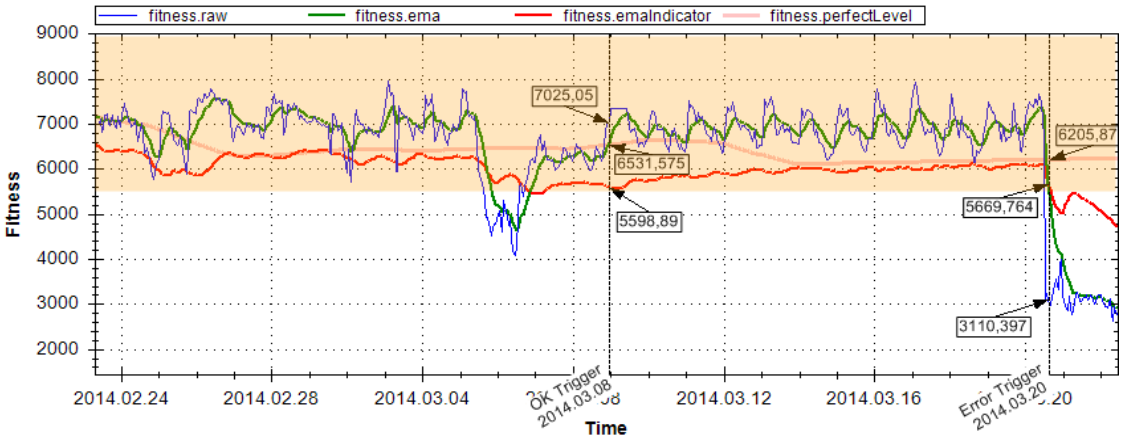

Figure 3. Fitness time series with example trigger points.

The user’s country and ISP are extracted from each IP address, and added to the Log Entries. We do not store any IPs for privacy reasons.

If a Log Entry contains an error ID, the SSC service performs a lookup to see if there is already a full stack trace, and at least one “JDownloader Application Log” for this error ID. In case either one is missing, the service adds a “Send Stack Trace” or “Send Full Log” request to the HTTP response. If the user agrees, their JD instance will then upload the requested information to the service. This way, there is a stack trace and at least one full application log for each error ID.

4.5. The "StatServ Evaluator" service

The "StatServ Evaluator" service (SSE) is designed as a separate process so that it can be restarted, updated, or killed at any time. The SSC service continues to receive and write data even if the SSE is not running. The SSE has three main modules: the "Aggregator", the "Analyzer" and the "Reporter".

4.5.1. The Aggregator

The aggregator turns the raw data into the fitness time series. The SSE looks at each "Plugin", "Account" (download mode), and "Source" (download from) Combination (PASC) separately. Typical combinations are "youtube.com"-"account.free"-"total" for all downloads done by the youtube.com plugin, with a non-paid account, or "premiumize.me"-"account.multi-.premium"-"rapidshare.com" for downloads done by the premiumize.me plugin, using a paid premium account and downloading from rapidshare.com.

For each PASC, the SSE aggregates the Log Entries to a fitness time series with an interval of 1 hour. The latest entry in this series is aggregated in a 15 minute interval. Thus, even if we use an overall interval of 1 hour, the latest entry is never older than 15 minutes. This way, problems can be found earlier. The default time frame that is used to aggregate data is 2 hours. However, there are many rarely used plugins that have low usage. To get stable fitness values, we dynamically increase the aggregation frame until the total usage is at least 200, or the maximum time frame of 168 hours (1 week) is reached. This corresponds to about 34 unique download attempts in 2 to 168 hours. A plugin that is used less than 34 times in a week is ignored.

The aggregation creates a list of "Chart Data" objects. All values are average values normalized to a time frame of 2 hours, and thus can be directly compared. All percentage metrics are in the range from 0 (0%) to 10,000 (100.00%). All metrics are set in relation to its usage to eliminate seasonal characteristics.

Network Fitness, Equation 13, describes how many connection problems occur in relation to total usage. A perfect value of 10,000 represents zero problems.

Plugin Fitness, Equation 14 shows the amount of PLUGIN_DEFECT Download Results. JD plugins are meant to throw a PLUGIN_DEFECT exception in case of parser problems.

Finished Fitness, Equation 15, captures how many download attempts finished successfully.

Finished- and Plugin Fitness, Equations 16 and 17, are merged to the General Fitness value of Equation 18, which will be subject of later chart analysis. The equation is a weighted average of both. Low fitness values get a higher relevance. The Plugin Fitness has a general 9 times higher weight than the Finished Fitness. It turned out that the Plugin Fitness is a much better error indicator, and therefore has higher relevance.

The manual user reports are grouped in an hourly interval and divided by the usage. The formulas in Equation 19 and 20 were developed in the first weeks after launching the manual reports feature in JD based on our experience with the system.

4.5.2. The Analyzer

After aggregation, there is a "Chart Data" list for each PASC stored on hard disk or in memory. The Analyzer module loops through all combinations, creates the TIL and the PLL. The TIL/PLL analysis is not only done for the General Fitness series, but also for the Network Fitness and the Reports Fitness series. The Network Fitness brings out network problems like bad connectivity to a web server. The User Reports Fitness may detect problems, even if the plugin's was implemented poorly. The model requires dynamic parameters several times to achieve better smoothing for low usage values (rarely used plugins). Equations 21 and 22 are used for this purpose.

$$f(usage) = \frac{2500}{usage + 5} \quad (21)$$

$$\lim_{usage \to \infty} f(usage) = 0 ; f(0) = 500 \quad (22)$$

TIL Model Parameter

The detailed description of the TIL algorithm can be found in Section 3. This section concentrates on specifying the dynamic model parameters.

First, we need to determine the Moving Average model parameter. We dynamically increase the parameter to get better smoothing for low usage values.

$$t_{MA}(avg_{usage}) = 6 + f(avg_{usage}) \quad (23)$$

Next, the second step in the TIL calculation is long-term smoothing using an EMA algorithm. The base parameter is 180 (hours).

$$t_{EMA}(avg_{usage}) = 180 + 2 \cdot f(avg_{usage}) \quad (24)$$

The final step subtracts the allowed deviation from the trend line. The deviation Δf is 10% of the average fitness + 50% of the MV over 12 hours. The more unstable the series is, the higher the allowed deviation is. Equation 25 captures this.

PLL Model Parameter

The detailed description of the PLL algorithm can be found in Section 3. Except for the dynamic model parameter for the EMA, all other parameters are static and have been explained in the model section.

The PLL suffers from the EMA start condition. This start condition results in less smoothing for the first $t_{EMA}(avg_{usage})$ values. Because this might result in a false

(12) $$errors_{network} = results_{CONNECTION\ ISSUES} + results_{CONNECTION\ UNAVAILABLE}$$

(13) $$networkFitness = 10000 \cdot \frac{usage - errors_{network}}{usage}$$

(14) $$pluginFitness = 10000 \cdot \frac{usage - results_{PLUGIN\ DEFECT}}{usage}$$

(15) $$finishedFitness = 10000 \cdot \frac{results_{FINISHED}}{usage}$$

(16) $$pluginFitnessFactor = \frac{9}{\frac{pluginFitness}{2500} + 0.5}$$

(17) $$finishedFitnessFactor = \frac{1}{\frac{finishedFitness}{2500} + 0.5}$$

(18) $$fitness = \frac{pluginFitnessFactor \cdot pluginFitness + finishedFitnessFactor \cdot finishedFitness}{pluginFitnessFactor + finishedFitnessFactor}$$

(19) $$reportsFitness = 10000 - \frac{10000 \cdot reportCounter}{0.86\% \cdot usage}$$

(20) $$reportsFitness = \begin{cases} 10000 \, if \ reportsFitness > 10000 \\ reportsFitness \, if \ 0 \le reportsFitness \le 10000 \\ 0 \, if \ reportsFitness < 0 \end{cases}$$

(25) $$\Delta f\left(avg_{usage}, t\right) = 10\% \cdot avg_{fitness} + 50\% \cdot mv_{12}(t)$$

(29) $$\Delta f\left(avg_{usage}, t\right) = 3.3\% \cdot avg_{fitness} + 50\% \cdot mv_{12}(t)$$

perfect level detection we simply ignore the first $t_{EMA}(avg_{usage})$ values for the calculation of the distribution function *DF(t)*.

(26) $$t_{EMA}\left(avg_{usage}\right) = 12 + f\left(avg_{usage}\right)$$

EMA Model Parameter

To get the current state of the PASC, we need to compare the TIL and the PLL to a short-time EMA. We use the same model parameter for this smoothing as shown in Equation 26.

User Reports

The User Reports series is different, because there is significantly more automated data than manually reported data. Users usually do not keep reporting a bug as long as the bug exists. The average end-user may report a bug once, and then wait for a solution. This is probably why there are very distinct peaks. To fit the model, the parameters need to be different. All Reports Fitness values below 6,000 are seen as anomalous, and all values above 9,800 are perfect.

For TIL, we use the model parameters from Equations 27, 28, and 29:

(27) $$t_{MA}\left(avg_{usage}\right) = 20 + f\left(avg_{usage}\right)$$

(28) $$t_{EMA}\left(avg_{usage}\right) = 24 + 2 \cdot f\left(avg_{usage}\right)$$

For PLL and EMA, we use the model parameter from Equation 30:

(30) $$t_{EMA}\left(avg_{usage}\right) = 3 + f\left(avg_{usage}\right)$$

4.6. Reporting

The final job of the SSE service is to report the results of the chart analysis. There is a reporting system for the developer community and a read-only front-end for the end-users to check the plugins' status. We use the open source software Redmine for this.

5. Results Discussion

5.1. Data measurements

We measured data for three months in 2014. During this period, the JD immune system observed about 2,800 different PASCs and spotted more than 103,231 different error IDs. 560 PASCs and 3,500 related errors were classified as relevant (by usage and occurrence count) to report them to the Bugtracker. 109 PASCs are about a problematic or anomalous state. 1,782 of all reported Error Issues have been solved, identified as duplicates or rejected. Figure 4 shows the priority distribution for all Error ID issues (open and closed). Only 1.4% has a priority of normal or higher. The issues priority is directly related to the number of incoming related error Log Entries. This shows that most of the found errors happen rarely.

The PASC State Overview Issues in Figure 5 shows almost the same distribution. This correlates with the JD usage numbers. A few plugins are popular and frequently used, others are hardly used. Figure 5 shows 109 PASCs marked as “anomalous” or “problematic” by the system. They have either a very bad general fitness (below 5,500), their current fitness is below the PLL or there are significant more user reports than usual. It turned out that many of the rarely used plugins are indeed damaged and require a review. Low volume errors like this often stay undetected for a long time, because there are no reports through the old feedback loop. StatServ now revealed all these problems.

5.2. Model Review

After finding the model parameters, the model turned out to work fine. However there are a few remarkable findings. First of all, it is important to notice, that even though we eliminated seasonal cycles, some plugins show distinct daily cycles in their fitness series. An explanation may be that there might be problems, which occur in certain time-zones only. In this case, the seasonal cycle of the usage series, and the error series would be phase shifted. The resulting fitness series would then still have seasonal cycles.

5.3. Automated Mode

The fully automated error collection mode delivers fast and precise data. Moreover, it collects stack traces and application logs for each error ID. This is a significant benefit for developers. One drawback of the automated data collection is a plugin’s potentially poor code quality. Each plugin should throw correct Download Results. However, some plugins may not do this.

5.4. Manual Data Provision

The downside of automated data collection can be compensated by manual user reports. Users report errors without knowing the exact reason. They just report if something does not work as expected. That’s why the User Reports Fitness series can detect a problem even if there is not a single automated error report.

However, user motivation to report problems varies significantly. The model parameters had to be adjusted several times because the amount of reports was not stable. In addition to that, and compared to the automated mode, user reports tend to arrive with a time delay. JD is an application that usually does its job in the background. A user will report a problem as soon as he or she detects that there is a problem. This is often many hours after the actual problem occurred. Moreover, the user cannot decide why his downloads failed. They will report if the problem is actually caused by his firewall, antivirus system, or anything else. This is probably the most severe drawback of the manual reports approach. Although it can detect severe problems, we just know that there is something wrong – nothing more. This is a challenging situation for a developer, because they require as detailed bug information as possible. Therefore, we have found the manual approach to be only a useful extension to the automated mode, but not a replacement.

6. Conclusions

JDownloader (JD) is a file download manager that utilizes a large set of open source plugins, which frequently have bugs or get out of date. To improve inno-

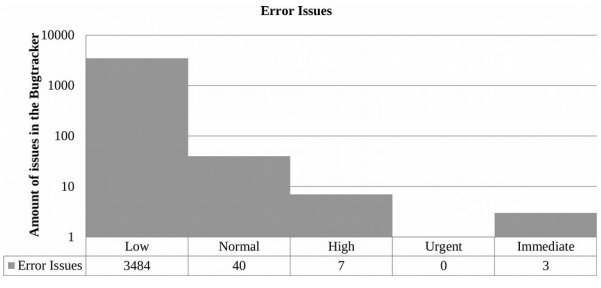

Figure 4. Distribution of error priorities.

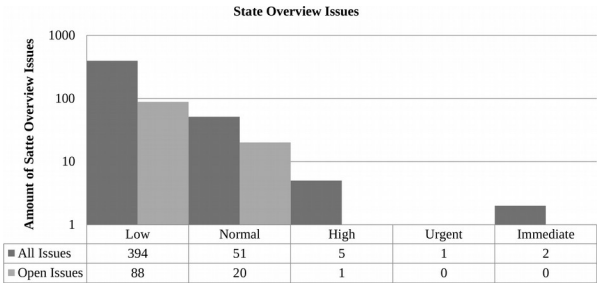

Figure 5. Distribution of state overview priorities.

vation speed, we designed the JD immune system, a system to more speedily identify malfunctioning plugins in deployed applications. We can now, within hours of the first occurrence of a plugin's malfunction, identify the severity of the malfunction and react to it.

The JD immune system helped us identify thousands of previously unknown bugs. The new system makes it possible to identify newly occurring bugs 16 times faster than before, and we show this by drawing on empirical data from JD's multi-million member strong user base.

References

[1] Brutlag, J. D. (2000). Aberrant Behavior Detection in Time Series for Network Monitoring. In LISA (Vol. 14).

[2] Chen, G., et al. (2015). A lightweight software fault tolerance system in the cloud environment. Concurrency and Computation: Practice and Experience, 27(12), 2982-2998.

[3] Feitelson, D. G. et al. (2013). Development and deployment at Facebook. IEEE Internet Computing, 17(4), 8-17.

[4] Ghosh, D. et al. (2007). Self-healing systems—survey and synthesis. Decision Support Systems, 42(4), 2164-2185.

[5] Gross, K. C. et al. (2006). Towards dependability in everyday software using software telemetry. In Engineering of Autonomic and Autonomous Systems, 2006. IEEE.

[6] Hangal, S., & Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In Proceedings of the 24th international conference on Software engineering (pp. 291-301). ACM.

[7] Hellerstein, J. L. et al. (2001). A statistical approach to predictive detection. Computer Networks, 35(1), 77-95.

[8] Ivan, I. et al. (2012). Self-Healing for Mobile Applications. Journal of Mobile, Embedded and Distributed Systems, 4(2), 96-106.

[9] Jiang, Z. et al. (2017). The Economics of Public Beta Testing. Decision Sciences, 48(1), 150-175.

[10] Karvonen, T. et al. (2017). Systematic Literature Review on the Impacts of Agile Release Engineering Practices. Information and Software Technology.

[11] Kumar, K. P., & Naik, N. S. (2014). Self-Healing model for software application. In Recent Advances and Innovations in Engineering (ICRAIE), 2014 (pp. 1-6). IEEE.

[13] Mäntylä, M. V., et al. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. Empirical Software Engineering, 20(5), 1384-1425.

[14] McIlroy, S., et al. (2016). Fresh apps: An empirical study of frequently-updated mobile apps in the Google play store. Empirical Software Engineering, 21(3), 1346-1370.

[15] Miller, E. (2007). Holt-winters forecasting applied to poisson processes in real-time. IMVU, Inc.

[16] Moran, K. et al. (2016). Automatically discovering, reporting and reproducing android application crashes. In 2016 IEEE international conference on software testing, verification and validation (ICST) (pp. 33-44). IEEE.

[17] Neely, S., & Stolt, S. (2013). Continuous delivery? Easy! Just change everything (well, maybe it is not that easy). In Agile Conference, 2013 (pp. 121-128). IEEE.

[18] Rechenmacher, T. (2014). The JDownloader Continuous Deployment Immune System. Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg: 2014.

[19] Rodriguez, P. et al. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. Journal of Systems and Software, 123, 263-291.

[20] Rossi, C. et al. (2016, November). Continuous deployment of mobile software at Facebook. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 12-23). ACM.

[21] Sahasrabudhe, M. et al. (2013). Application performance monitoring and prediction. In Signal Processing, Computing and Control (ISPCC), 2013 IEEE International Conference on (pp. 1-6). IEEE.

[22] Savor, T. et al. (2016). Continuous deployment at Facebook and OANDA. In Proceedings of the 38th International Conference on Software Engineering Companion (pp. 21-30). ACM.

[23] Schröter, A. et al. (2010). Do stack traces help developers fix bugs? In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (pp. 118-121). IEEE.

[24] Sharifi, M. et al. (2012). Predictive Self-Healing of Web Services Using Health Score. Journal of Web Engineering 11(1), 79.

[25] Cesar Brandão Gomes da Silva, A., de Figueiredo Carneiro, G., Brito e Abreu, F., & Pessoa Monteiro, M. (2017). Frequent releases in open source software: A systematic review. Information, 8(3), 109.

[26] Silva, L. M. (2008). Comparing error detection techniques for web applications: An experimental study. In Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on (pp. 144-151). IEEE.

[27] Suonsyrjä, S. et al. (2016). Post-Deployment Data: A Recipe for Satisfying Knowledge Needs in Software Development?. In Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2016 Joint Conference of the International Workshop on (pp. 139-147). IEEE.

[28] Szmit, M., & Szmit, A. (2012). Usage of modified Holt-Winters method in the anomaly detection of network traffic: Case studies. Journal of Computer Networks and Communications, 2012.

[29] Vogler, W. (2014). The Story of Apollo – Amazon's Deployment Engine [Blog Post]. Retrieved from: http://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html

[30] Wang, H., & Wan, C. (2014). Quality Failure Prediction for the Self-Healing of Service-Oriented System of Systems. In 2014 IEEE International Conference on Web Services (pp. 566-573). IEEE.

[31] Ye, F. et al. (2016). The Research of Enhancing the Dependability of Cloud Services Using Self-Healing Mechanism. In 2016 International Conference on Intelligent Networking and Collaborative Systems (pp. 130-134). IEEE.