Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Weber, Michael

MASTER THESIS

# The Benefits of Continuous Deployment evaluated using the JDownloader Software

Submitted on 04.07.2017

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

## Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nuremberg, 04.07.2017

## License

Nuremberg, 04.07.2017

# Abstract

Continuous Delivery and Continuous Deployment approaches have seen widespread adoption in the software industry. To harness such techniques effectively, close monitoring and detailed knowledge about the state of software in production is highly desirable.

This thesis analyzes the *JDownloader Immune System*, a real-time monitoring and error detection mechanism developed for the open source download manager software JDownloader. It describes the mathematical model for error detection based on time series analysis and Holt-Winters-Forecasting. The thesis continues to provide insight on the architecture of the immune system and shows how it provides useful information to developers and users through state dashboards.

Finally, it evaluates the effectiveness of the immune system compared to manual user reports. The thesis finds that error detection speed for severe issues is 16 times faster than through manual reports and critical bugs are more than four times more likely to be detected within the first 24 hours after their first appearance.

# Contents

## Equations

## Figures

## Tables

# 1 Introduction

## 1.1 Original Thesis Goals

The original goal of this thesis is to exemplify the benefits of Continuous Deployment techniques by presenting the JDownloader software and specifically what is called the “JDownloader Immune System” as a single outstanding example for such practices.

The thesis aims to provide a sound literature review on how continuous deployment techniques are used in similar scenarios and more specifically how the ability to release software more frequently impacts testing and software quality. The second part of the literature examines existing techniques for gathering post-deployment data (also known as software telemetry, software monitoring or application health monitoring) and identify research on systems that automatically act on the gathered data (e.g. self-healing systems).

It then tries to answer this research question:

- *How can software quality be improved when the complexity of the surrounding environment does not allow for comprehensive pre-release testing?*

The answer to this question is a description of the implementation of the Immune System at JDownloader. The thesis first describes the necessary software architecture for effectively collecting post-deployment data and then describes the mathematical model used for identifying abnormal application states.

In a last step, current defect data from JDownloader is analyzed according to quality metrics and recommendations for the implementation of such systems are presented.

## 1.2 Changes to Thesis Goals

The status of error tracking and monitoring at JDownloader before the introduction of the Immune System does not allow for the extraction and separation of meaningful data for comparison. As such, this thesis remains merely descriptive in extended parts of the research chapter.

The second change refers to measuring current JDownloader performance. Since the Immune System is currently extended, current data could not be extracted and it was agreed to use legacy data.

# 2 Research Chapter

## 2.1 Introduction

Continuous Integration, Continuous Delivery and Continuous Deployment have become ubiquitous terms among software practitioners and have changed the way software is delivered today. The ability to rapidly deploy software into production systems has profound impact on software quality. Practices to rapidly fix software bugs require effective monitoring tools to timely detect potential issues. Progress in the research regarding such tools has mainly been made in the *self-healing systems* domain.

In this paper, we describe the implications of such practices and tools by analyzing the JDownloader (JD) software, an Open Source Download Manager application. We focus on what is called the *JDownloader Immune System*, describe the implementation of the system and provide an insight on how adopting this approach impacted error detection rates and speed at JDownloader.

JD heavily relies on frequently changing external interfaces (e.g. download URLs from filesharing websites) and faces unique challenges in this regard. We nevertheless believe that the solutions proposed at JDownloader are applicable in a variety of different contexts and thus a valid research subject.

## 2.2 Related Work

### 2.2.1 Continuous Deployment and its impact on testing and software quality

One argument for the introduction of an immune system at JDownloader was the capability to deliver bug-fixes more rapidly. The increased frequency of software releases enabled by Continuous Deployment methods has sparked vivid interest in the academic community and several systematic literature reviews have been published on the topic (Mäntylä et al., 2015; Laukkanen et al., 2017; Rodriguez et al., 2017; Karvonen et al, 2017). Regarding software quality, these reviews report mixed results.

*Browsers*

Browsers are an interesting subject for comparison as they face similar issues to JDownloader. Both are desktop applications having to deal with high variation of system configurations and must ensure compatibility with many interacting plugins. Firefox[1] and Chromium[2] are large Open Source projects and have been subject to prior academic research. The Firefox browser alone has been subject to 10 individual studies focusing on software quality after changing release cycles from 12-18 months to 6 weeks (Karvonen, et al., 2017). In the case of Firefox, shorter release cycles have not had a significant impact on the number of bugs produced (Khom et al., 2012) and the software has not become less secure (Clark et al., 2014). Souza et al. (2015) find that the quicker release cycles increased re-opened bugs by 7%. Bug-fix time is significantly shorter using rapid releases according to Khom et al., 2015, however Da Costa et al (2014) report that only 2% of error fixes are integrated in the next release while 8% were fixed after one cycle, 89% after two cycles and 1% after three or more release cycles. The median delay for an error fix is 42 days, a relatively short response time compared to other open source projects (Eclipse: 112 days, ArgoUML: 180 days (Da Costa et al., 2014)).

[1] https://www.mozilla.org/en-US/firefox/
[2] https://www.chromium.org/Home

While Firefox is different from JDownloader in many aspects (larger user base, drastically more complex, large manual testing efforts before each release), the metrics used for evaluating software quality are largely identical and will serve as a basis for our evaluation of JDownloader.

*Mobile Applications*

Another area facing similar challenges to JDownloader are mobile applications. Especially Android applications deal with differing hardware, multiple operating system versions, geographic differences, multiple network providers and inexplicable 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 at customer devices that must continue to work as intended. Also, updates can only be distributed via the OS provider's tools (iOS App Store, Android Play store). This makes the update frequency not only dependent on user acceptance but also on the due diligence by those providers. In a case study of mobile software deployment at Facebook, Rossi et al. (2016) confirm that Continuous Delivery practices do not negatively impact software quality. 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 down to two (iOS) weeks or one (Android) week. Other studies (McIlory et al., 2015) suggest that more frequently updated apps correlate with higher ratings in Google's Play store.

*Web based applications*

Continuous Deployment practices emerged in the web applications domain. Changes can be made without any user interaction and companies like Amazon measure time between deployments in seconds (Vogler, 2014). The impact of Continuous Delivery on web based applications has been analyzed at Rally Software (Neely & Stolt, 2013), Facebook and OANDA (Savor et al., 2016). One extraordinary case is presented by the Guardian news organization (Goble, 2016). The Guardian claims to have abolished all but 4 automated regression tests as they deploy 400 times to production per day and fixing errors as they occur is more economically sound than running more extended tests prior to each deployment. While not yet subjected to academic rigor, the case is interesting because it is the first time we hear about reducing pre-release quality assurance for economic reasons.

*Testing with users*

Most analyzed examples strongly emphasize the importance of extensive pre-release testing through automated pipelines and manual tests. Facebook contracts a manual test team of roughly 100 people (Rossi et al., 2016) and Firefox recruits manual testers from its Open Source contributor base as well as paying teams for testing (Mäntylä et al., 2015). This approach is not feasible for all types of projects and companies. Some thus 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 (Savor, 2016; Feitelson, Frachtenberg, Beck, 2014). Jiang et al.'s work (2016) 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. Rodriguez et al. (2017) identify user involvement through canary releases and dark launches as an area with a distinct lack of research. For JDownloader, this is especially relevant because automated or manual testing is neither always economically viable nor technically possible.

### 2.2.2 Immune Systems and system health monitoring

*Self-Healing Systems*

The term self-healing system is significantly more common than “Software Immune System” in the academic literature. Ghosh et al. (2007) 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.”*

Following this definition, we consider the JDownloader Immune System to be an assisted self-healing system. They identify three main aspects for analysis in self-healing systems:

- Maintenance of System Health
- Detection of System Failure
- System Recovery

For JDownloader, system recovery requires human intervention and no automatic mechanism for system health maintenance to be in place, which is why we focus on literature regarding failure detection. Early research includes a statistical model for predictive failure detection (Hellerstein et al., 2001) and a proposal for automatic anomaly detection through gradually relaxing invariants (Hangal & Lam, 2002). Gross et al (2006) provide a framework for detecting software anomalies and identify four crucial aspects for such a system: data management libraries, statistic modeling tools, corrective action strategy support tools and an adequate software architecture. Ivan et al. (2012) describe a self-healing system for a mobile application where the architecture is similar to the one we propose for JDownloader, while Kumar & Naik (2014) extend the model towards autonomic computing. Fukuda et al. (2016) and Moran et al. (2016) continue the research on self-healing for mobile systems and provide strategies for monitoring Android applications to discover and report application crashes. Sahasrabudhe et al. (2013) describe application performance monitoring as a sequence of four steps: monitor, analysis, recommendations, action. 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 at the JDownloader example. Chen et al. (2013) and Ye at al. (2016) provide a more recent application of self-healing techniques in cloud software. Table 1 shows how data collection, fault detection, and reactive measures are handled in systems analyzed by current research.

| **Topic:** | **Approach:** |
|---|---|
| Data collection | • System data: Ye et al. (2016), Gross et al. (2006), Fukuda et al. (2016)<br>• Application data: Gross et al. (2006), Sahasrabudhe (2013),<br>• Experimental Inputs: Moran et al. (2016) |
| Fault detection | • Log errors: Ye et al. (2016),<br>• Connection threshold prediction: Hellerstein (2001),<br>• Relaxing Invariants: Hangal & Lam (2002),<br>• Performance time-series analysis (Hellerstein (2001), Sahasrabudhe (2013), Brutlag (2000), Miller, (2007)<br>• Artificial Neural Networks: Sharifi et al. (2012)<br>• Cluster-based: Wang et al. (2013) |

| | |
|---|---|
| Reactive measures | • Retry/Restart: Ye et. al. (2016), Ivan et al. (2012)<br>• Rollback: Ye et al. (2016), Chen et al. (2013)<br>• Re-route: Chen et al. (2013)<br>• Disable non-functional components: Gross et al. (2006) |

*Table 1: Approaches in self-healing systems*

Holmström & Olson (2013) and Suonsyrjä et al. (2016) correctly 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. Apart from academic research, several practitioner's tools have been developed and marketed to facilitate application monitoring, post-deployment analytics and creating dashboards. These include Nagios[3], Splunk[4], Prometheus[5], Zabbix[6] or Crashlytics[7]. Little research exists on the viability of such tools, especially in a self-healing systems scenario. Interestingly, no connection between the goal of self-healing systems and the relatively new possibility of continuously and automatically deployed software has been analyzed in academic research so far.

*Mechanisms for error detection and prediction*

Silva (2008) describes four approaches to detect errors in deployed software systems. Systems-level monitoring, failure detection at the application layer, error detection by log analysis, and remote detection of user failures. However, no consistent explanation what behavior is defined as abnormal is given. In this paper, we build on techniques originally developed for network monitoring by J.D. Brutlag (2000) and Evan Miller (2007) using the Holt-Winters Forecasting algorithm. Szmit & Szmit (2012) summarize more applications of this algorithm for anomaly detection in network monitoring. Sharifi et al. (2012) show how neural networks can be used to predict failures in web applications. Wang & Wan (2014) develop a self-healing model for systems of systems using stochastic differential equations and Brownian Motion. General information on statistical forecasting and error detection techniques are presented by Box et al. (2015).

## 2.3 Research Question

The related work indicates that there currently is a lack of material drawing a connection between the possibilities offered by (assisted) self-healing systems and software quality as perceived by the user. We thus aim to provide insight into the practical use of such systems in combination with Continuous Deployment practices to demonstrate the impact on software quality by answering the following research question:

- *How can software quality be improved when the complexity of the surrounding environment does not allow for comprehensive pre-release testing?*

We see that the impact of rapid releases on software quality has been applied in practice and analyzed thoroughly, while the concept of self-healing software has remained mostly academic.

[3] https://www.nagios.org/
[4] https://www.splunk.com/
[5] https://prometheus.io/
[6] http://www.zabbix.com/
[7] https://try.crashlytics.com/

## 2.4 Research Approach

Our research subject, the JDownloader software, is a single outstanding case (Shaw, 2003) for a real-world application of an assisted self-healing system. We mainly draw from an earlier description of the implementation by Rechenmacher (2014).

We first provide a description of the mathematical model applied for detecting anomalous behavior of the system. We describe how the Holt-Winters Forecasting algorithm is applied to analyze time-series data for each individual plugin.

Afterwards, we describe the architecture of the JDownloader Immune System. We show the interaction between data collection modules and data evaluation modules in a plugin based system.

Finally, we analyze two distinct metrics to describe the effectiveness of the JDownloader Immune System:

- Number and severity of errors detected by the immune system compared to manual user reports
- Time between first error occurrence of the issue and reporting by the immune system compared with user report speed for severe errors

Unfortunately, there is innsufficient data for the behavior of JDownloader before the introduction of the Immune System. As such, these statistics remain merely descriptive and cannot be tested against a null hypothesis describing the status before the introduction.

## 2.5 Evaluation of case

### 2.5.1 An Overview of the JDownloader Software

JDownloader is an open source software used for downloading content from various sources. It consists of a JDCore application maintained by a team of core developers and 1,230 plugins for different hosting websites developed and maintained by the Open Source community. It uses crawler plugins to scan the URL of hosting providers for downloadable content and hoster plugins for downloading. As many hosting sites change their URL structures or access models frequently, testing each plugin in advance is neither technically possible nor economically feasible. This sparked the need for a mechanism to quickly and automatically detect non-functional plugins and led to the development of the JDownloader Immune System.

### 2.5.2 Model used for predicting anomalous behavior

In this section, we describe the mathematical models used for analyzing the state of each individual hoster plugin. First, the JDownloader Immune System continuously calculates a fitness value for each individual plugin and from this fitness time-series generates two trend lines for evaluating the current plugin fitness: The Trend Indicator Line (TIL) and Perfect Level Detection (PLL).

Figure 1 provides an example for this behavior. The blue graph represents raw fitness data, the green graph shows the smoothed time series after calculating exponential moving averages (EMA). The red line (labeled “Error Trigger #1”) indicates the Trend Indicator Line (TIL), while the light red line (“Error Trigger #2) represents the long-term trend shown by the Perfect Level Detection (PLL).

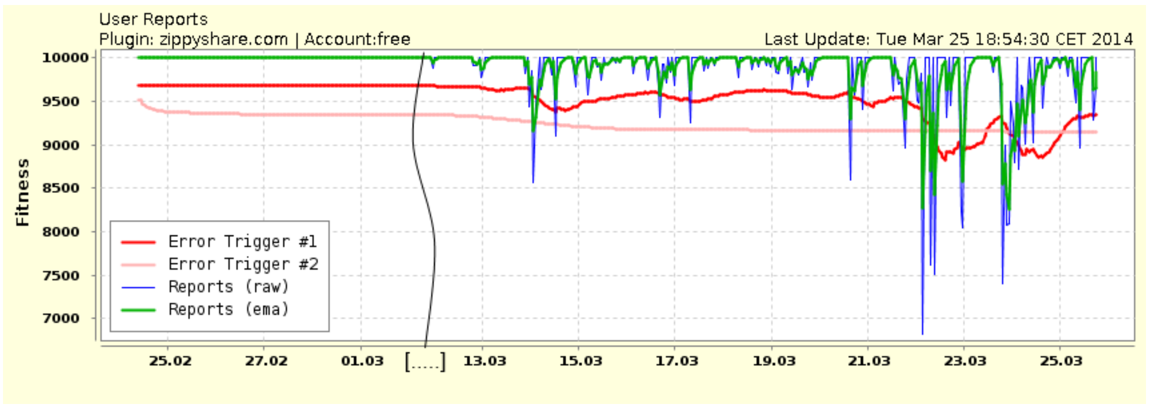


*Figure 1: Exemplary Fitness Line, Trend Indicator Line and Perfect Level Line from the zippyshare.com plugin (Rechenmacher, 2014)*

### 2.5.2.1 Fitness value

Application fitness is the fundamental metric for describing the current state of an individual JDownloader plugin. To calculate the fitness value, we track the number of download attempts in each period

*Equation 1: Basic usage definition*

$$usage = \sum_{t=0}^{1\ hour} Download\ attempt$$

and the number of failed download attempts (errors).

*Equation 2: Basic error definition*

$$errors = \sum_{t=0}^{1\ hour} Failed\ Download\ attempt$$

We then divide the number of failed download attempts within a given period (typically one hour) by the total number of download attempts in the same period to get a basic fitness value.

*Equation 3: Basic fitness definition*

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

Since errors can have different causes unrelated to the functionality of a plugin, we differentiate between network errors and plugin errors represented by corresponding error codes. For visualization purposes, each value is multiplied by 10,000. A value of 10,000 thus represents perfect fitness while 0 represents no functionality at all.

*Equation 4: Network fitness definition*

$$networkFitness = 10{,}000 * \frac{usage-(results_{CONNECTION\_ISSUES}+results_{CONNECTION\_UNAVAILABLE})}{usage}$$

To correctly calculate the overall fitness value, we look at *plugin fitness*, defined by plugin usage without error codes, and *finished fitness*, the ratio between all successfully finished downloads and the total usage. Plugin fitness is defined as

*Equation 5: Plugin fitness definition*

$$pluginFitness = 10{,}000 * \frac{usage - results_{PLUGIN\_DEFECT}}{usage}$$

*Finished fitness* correspondingly is

*Equation 6: Finished fitness definition*

$$finishedFitness = 10{,}000 * \frac{results_{FINISHED}}{usage}$$

The two values are then weighted and merged into the overall *fitness* indicator. Since *plugin fitness* directly relates to plugin defects compared to overall successful downloads, it is a significantly better indicator. It is thus weighted with a factor of nine, compared to one for the *finished fitness* factor. This leads to the following fitness factors:

*Equation 7: Plugin fitness factor*

$$pluginFitnessFactor = \frac{9}{\frac{pluginFitness}{2{,}500} + 0.5}$$

*Equation 8: Finished fitness factor*

$$finishedFitnessFactor = \frac{1}{\frac{finishedFitness}{2{,}500} + 0.5}$$

With these factors in place, overall fitness can be calculated.

*Equation 9: Complete fitness definition*

$$fitness = \frac{pluginFitnessFactor * pluginFitness + finishedFitnessFactor * finishedFitness}{pluginFitnessFactor + finishedFitnessFactor}$$

The next step is to determine whether the calculated fitness for a plugin is acceptable or requires intervention by a developer. As acceptable fitness levels can vary depending on the plugin, we dynamically calculate two indicator lines: the Trend Indicator Line (TIL) and the Perfect Level Detection (PLL). Should the fitness value be above both indicator lines, a plugin is considered fit and does not require attention. If the fitness value drops between the two indicator lines, the plugin status is changed to problematic and might need inspection by a developer. If the fitness value drops below both lines, a plugin state is considered anomalous.

#### 2.5.2.2 Trend Indicator Line

The Trend Indicator Line (TIL) is constructed by performing a series of mathematical operations on the time-series fitness data for a specific plugin with an observation interval of one hour. The length of the time series contains 200 usage events or a maximum period of 168 hours (one week).

The first operation is the calculation of a moving average for the time series.

*Equation 10: Moving average formula*

$$ma_{n_{MA}}^{offset}(t) = \frac{1}{n_{MA}} \sum_{i=1}^{n_{MA}} x(t - n_{MA} + i + offset)$$

This equation contains the dynamic model parameter n$_{MA}$ and is determined by dynamically increasing the parameter.

*Equation 11: Moving average dynamic model parameter*

$$(avg_{usage}) = 6 + f(avg_{usage})$$

The function $f(avg_{usage})$ assigns higher values for plugins with low usage to achieve better smoothing.

*Equation 12: Smoothing for low usage numbers*

$$f(usage) = \frac{2{,}500}{usage + 5}$$

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

In the second step, the smoothed curve is subjected to another smoothing round, this time applying an Exponential Moving Average algorithm thus giving more recent data higher weight.

*Equation 13: Exponential moving average formula*

$$ema(t+1) = \beta * x(t) + (1 - \beta) * ema(t)$$

$\beta$ defines the speed of decay for old values. It is calculated using the dynamic parameter $n_{EMA}$ which represents the hours passed and is used to calculate the next prediction.

*Equation 14: Decay rate for old values*

$$\beta = \frac{2}{n_{EMA} + 1}$$

leading to the final equation

*Equation 15: Exponential moving average with decay rate*

$$ema_{n_{EMA}}(t+1) = \frac{2}{n_{EMA} + 1} * x(t) + \left(1 - \frac{2}{n_{EMA} + 1}\right) * ema_{n_{EMA}}(t)$$

The dynamic parameter $n_{EMA}$ is calculated by starting with a baseline of 180 hours and adding the average usage for a certain time-period multiplied by a factor of two.

*Equation 16: Dynamic model parameter for exponential moving average*

$$n_{EMA}(avg_{usage}) = 180 + 2 * f(avg_{usage})$$

In the last step, the allowed deviation is subtracted from the trend line. The allowed deviation $\Delta f$ consists of 10% of the average fitness and 50% of the moving variability within the last twelve hours.

*Equation 17: Allowed deviation from the trend line*

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

Moving variability calculates the deviation at each point in the time series by subtracting the moving average at a certain point from the respective value.

*Equation 18: Moving variability formula*

$$mv_{n_{VA}}^{offset}(t) = \frac{1}{n_{VA}} \sum_{i=1}^{n_{VA}} |x(t - n_{VA} + i + offset) - ma_{VA}(t - n_{VA} + i + offset)|$$

#### 2.5.2.3 Perfect Level Detection

Perfect Level Detection (PLL) defines a second approach for evaluating the typical fitness of a functioning plugin. For the calculation, it uses the exponential moving average algorithm described in the section above, followed by a distribution function that detects high concentrations of fitness values within a time series. It assumes that this distribution peaks indicate the ideal fitness value for a plugin to operate. The distribution function looks like this:

*Equation 19: Distribution function for values of the time series*

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

The resulting distribution graph is again smoothed using a moving average (MA) with a model parameter $n_{MA}$ of 4% and an offset of 2%. Figure 2 shows the concentration of fitness values for the fitness chart presented in figure 1.

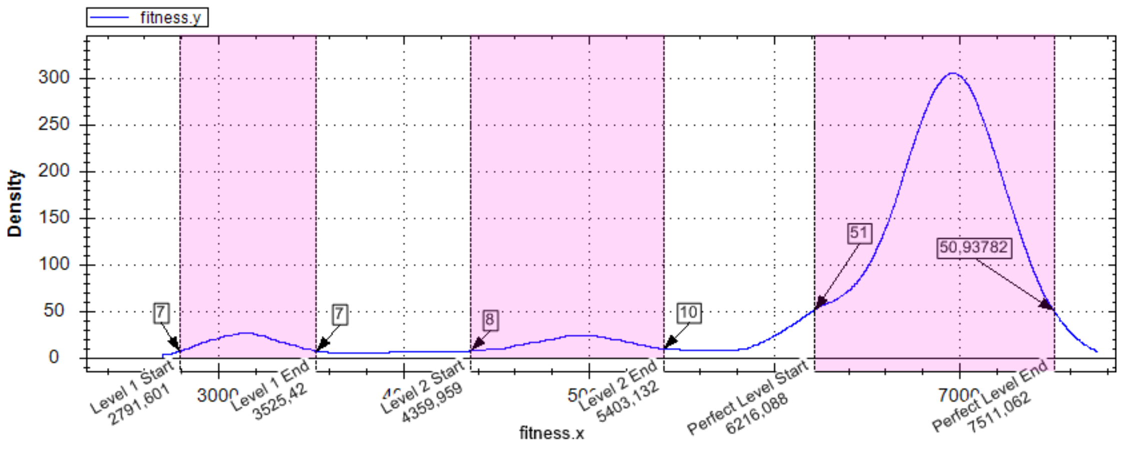


*Figure 2: Fitness value distribution and perfect level band detection (Rechenmacher, 2014)*

The threshold levels are identified by analyzing the gradient of the distribution function. The section with the highest density of fitness values is defined as the perfect level band. The lower edge of this band minus 5% tolerance is the perfect level threshold. The perfect level function pll(t) is the continuing series of all perfect level values.

#### 2.5.2.4 Anomalous Behavior Identification

Figure 3 puts these individual pieces of information together. It shows a sudden drop in the raw and smoothed fitness data. First, the Perfect Level Line (PLL) is crossed at a fitness value of 6,205.872, putting the Plugin-fitness in “problematic state”. As the fitness value continues to drop and crosses the Trend Indicator Line (TIL) at 5,669.764, the plugin is considered anomalous, an error trigger is sent and a developer is notified.

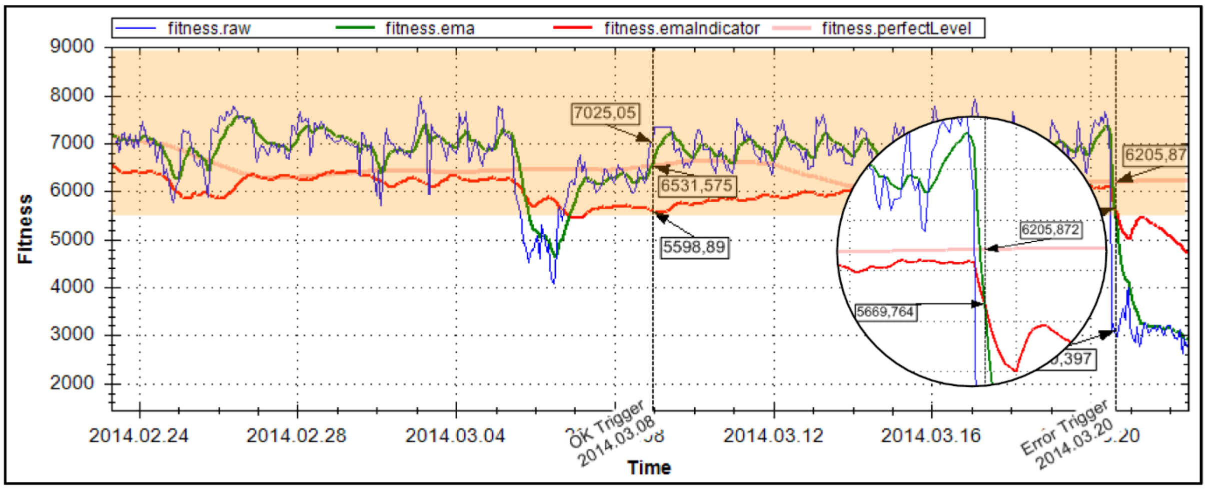
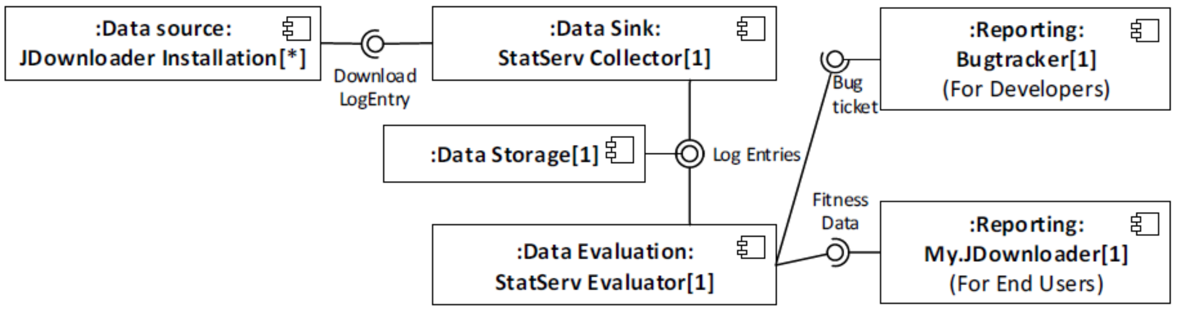


*Figure 3: Anomalous behavior detection through the JDownloader Immune System (Rechenmacher, 2014)*

### 2.5.3 Architecture and Implementation of the JDownloader Immune System

The functions described in the model section before are executed in two components within the Immune System known as StatServ: The StatServ Collector (SSC) Service and the StatServ Evaluator (SSE) Service. StatServ extended a previously existing Feedback Loop where users could manually report problems via a Community Board, a Live Chat and through a Support Desk. These reports were manually evaluated by either AppWork staff or supporters from the Open Source community. Once verified, bugs where manually entered into a bug-tracking system. StatServ currently logs 850,000 events per hour on average.



:Data source: JDownloader Installation[*]
Download LogEntry
:Data Sink: StatServ Collector[1]
:Reporting: Bugtracker[1] (For Developers)
Bug ticket
:Data Storage[1]
Log Entries
Fitness Data
:Data Evaluation: StatServ Evaluator[1]
:Reporting: My.JDownloader[1] (For End Users)

*Figure 4: Architecture of the JDownloader Immune System*

#### 2.5.3.1 Data Sources

The JDownolader Immune System continues to accept manual user reports submitted via a “Report a Problem”-button. Reports are generated as JSON files and contain the following information:

| | |
|---|---|
| **buildTime** | buildTimestamp used to identify the running JD version |
| **timestamp** | Local user time |
| **sessionStart** | Time when the current JD session was initiated |
| **linkID** | Link identifier for this JD installation |
| **host** | Top Level Domain (TLD) of the URL related to the error report. |
| **candidates** | List of plugin-candidates who could potentially have been used to download the URL in question. A *candidate* consists of a plugin-identifier (*plugin*) + subtype of the plugin (*type*) + VCS-version of the plugin (*revision)* |

*Table 2: Manual user report data*

In addition, every single (successful or unsuccessful) download attempt generates a fully automated report which, since the relevant plugin is still in memory, can aggregate more error-related information:

| | |
|---|---|
| **candidate** | Specification of the candidates list above. Since the plugin is still in memory, the “culprit” can be identified easily. Stored as *type + plugin + revision* |
| **result** | Result of the download attempt (e.g. FINISHED, PLUGIN_DEFECT, SKIPPED, …) |
| **errorID** | If a download attempt results in an error, a unique error id is stored in this attribute. When a plugin experiences a problem, an error-specific exception is thrown, the download routine stopped and a stack trace generated. The errorID represents the MD5 checksum for the stack trace to identify identical errors. |

*Table 3: Additional automated report data*

#### 2.5.3.2 StatServ Collector Service (SSC)

The SSC service serves two purposes. It efficiently collects and stores error and fitness data and polls stack traces or application logs for each error id to allow easier debugging.

SSC is an FCGI module providing a REST API and is optimized to handle large numbers of requests. It enriches uploaded log entries with the user’s country and ISP, collects log entries in memory and writes them to hard disk in batch sizes of 30,000. Evaluation is not done in real-time and raw data is saved in JSON format to allow recalculations should the model change.

If for a specific error id, a stack trace or application log is missing, the SSC adds a request for “Send Stack Trace” or “Send Full Log” to the HTTP response and – if the user agrees – adds that information to each error ID.

To ensure that each stack trace can be attributed to the correct error id, line numbers are removed from the stack trace and replaced with the actual line of code. To do this, the SSC service is connected to version control and - via the *buildtime* and *candidate.revision* attribute - is able to retrieve the relevant source code file for each line in the stack trace.

Full application logs contain all relevant information about the specific JD session, including environment information such as the operating system, java version, launcher version, and other. Since the full session is recorded, the information stored in these application logs is highly valuable for debugging. Logs also contain sensitive information like premium accounts or URLs and are thus stored on a separate server.

#### 2.5.3.3 StatServ Evaluator Service (SSE)

The StatServ Evaluator component serves the purpose of processing the calculations outlined in chapter 2.5.2 It consists of three main modules: The Aggregator, the Analyzer and the Reporter.

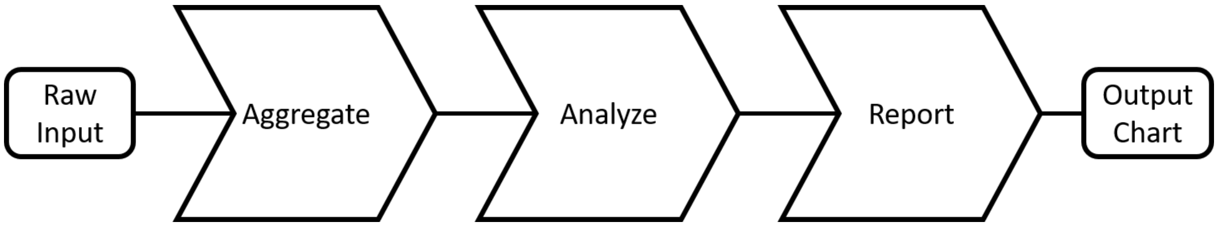


*Figure 5: StatServ processing sequence*

*The Aggregator module*

The Aggregator collects and aggregates data based on the following information triple: Plugin (P) – Account (download mode) (A) – Source (downloaded from) (S). Practical examples are

- “youtube.com” (P) - “account.free” (A) - “total” (S): For all downloads done by the “youtube.com” plugin
- “premiumize.me” (P) - “account.multi.premium” (A) - “rapidshare.com” (S): for all downloads from rapidshare using a premium account through the “premiumize.me” plugin.

For these Plugin-Account-Source-Collections (PASCs), raw log entries are aggregated to a fitness time series with an interval of one hour, while the last entry is aggregated every 15 minutes. This allows problem detection to happen more quickly without compromising overall system performance. Every two hours, this data is written into a single ChartData object. Since a period of two hours is not sufficient for low-usage plugins, the period is gradually increased until the total usage reaches 200. If remains below this value for 168 hours (one week), the time series is ignored.

From the resulting list of ChartData objects, the aggregator module calculates the fitness metrics described in section 2.5.2 to determine the overall fitness of a plugin. In addition to the automated fitness calculation, user reports are stored and calculated in a separate reportsFitness value.

*Equation 20: Reports fitness formula*

$$reportsFitness = 10{,}000 - \frac{10{,}000 * reportCounter}{0.86\% * usage}$$

*The Analyzer module*

The Analyzer module calculates the TIL and PLL indicator lines and compares the current fitness value against those thresholds. It iterates through all ChartData objects stored for each PAS-combinations and creates the trend lines. The TIL/PLL analysis is also performed for the Network Fitness and Reports Fitness time series.

For the Trend Indicator Line (TIL) and Perfect Level Detection (PLL), dynamic parameters are determined as described above while calculations for reportsFitness vary slightly. The module then compares the most recent EMA for a particular PASC and determines whether anomalous behavior is present and a developer must be notified.

*The Reporter module*

Irrespective of error state, the reporting module generates two types of reports. A detailed status report for each plugin is posted to open source issue tracking and project management tool Redmine[8]. The State Overview contains the following information:

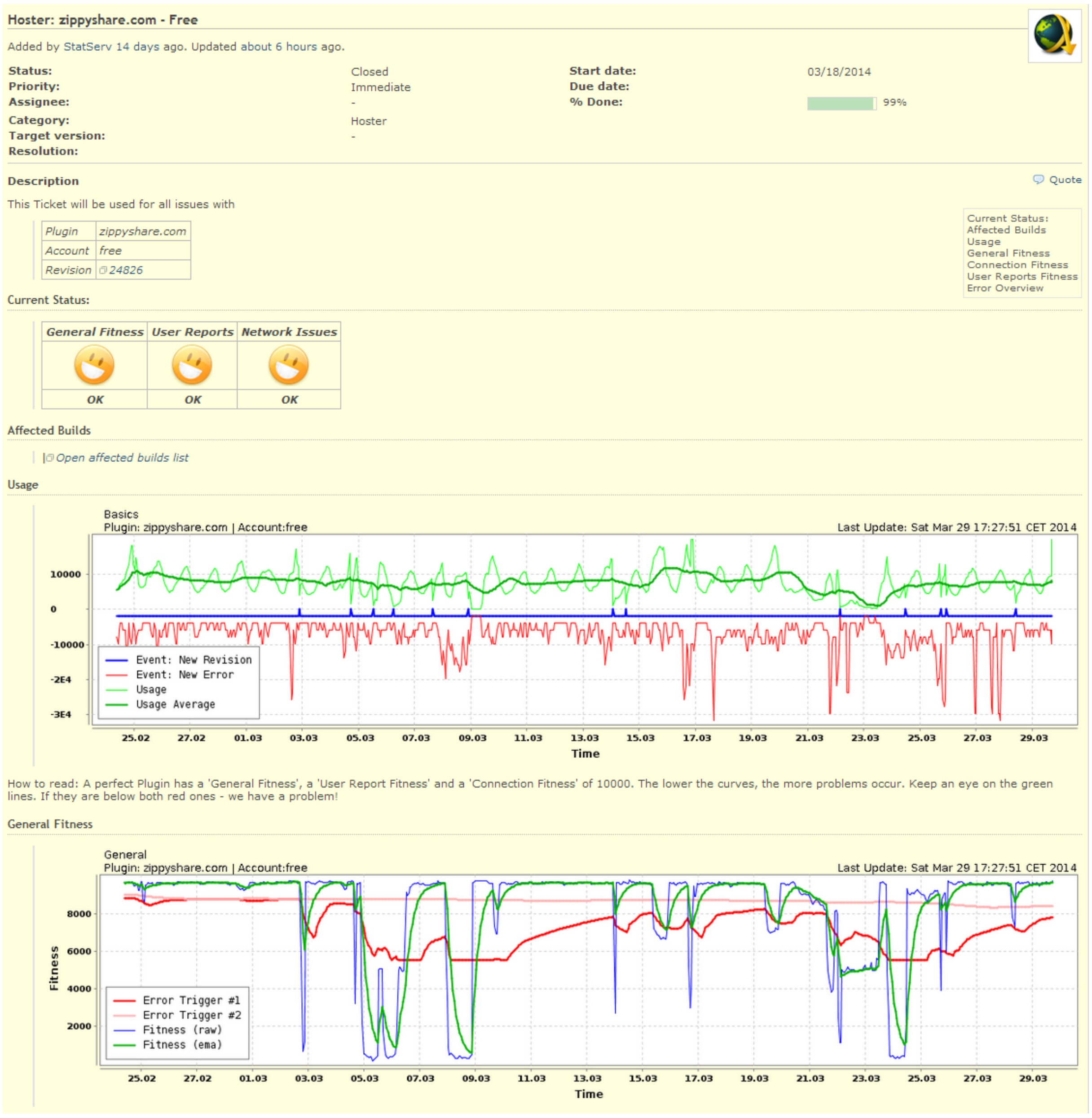
| | |
|---|---|
| **Status** | Working PASCs have the Status “closed” which is changed to “new” when a PASC stops working |
| **Priority** | Depending on the ratio between low fitness and high usage |
| **Revision** | Last revision of the plugin in version control |
| **Current Fitness States** | A quick emoticon-based overview of current fitness states |
| **Affected Builds** | Time stamps of currently active builds |
| **Usage Chart** | Graph showing usage, revisions, and error events over time |
| **General Fitness Chart** | Graph plotting the raw fitness series, EMA, TIL, and PIL |
| **Network Fitness Chart** | Graph plotting raw networkFitness series, EMA, TIL, and PIL |
| **User Reports Chart** | Graph plotting the raw reportsFitness series, EMA, TIL, and PIL |

[8] http://www.redmine.org/

| | |
|---|---|
| **Current Error Overview** | Overview of currently unresolved errors relating to this PASC |
| **Current Error Chart** | Graph displaying introduction of new revisions, errors, and fixes over time |
| **All Time Error Chart** | Like the current error chart but plotted over all revisions of a certain PASC |

*Table 4: Plugin-Account-Source-Collection (PASC) state overview*

Figure 6 shows an example for the dashboard created presenting overview information as well as usage and general fitness for the Zippyshare.com plugin.



Hoster: zippyshare.com - Free

Added by StatServ 14 days ago. Updated about 6 hours ago.

| | | | |
|---|---|---|---|
| Status: | Closed | Start date: | 03/18/2014 |
| Priority: | Immediate | Due date: | |
| Assignee: | - | % Done: | 99% |
| Category: | Hoster | | |
| Target version: | - | | |
| Resolution: | | | |

Description

Quote

This Ticket will be used for all issues with

| | |
|---|---|
| Plugin | zippyshare.com |
| Account | free |
| Revision | 24826 |

Current Status:
Affected Builds
Usage
General Fitness
Connection Fitness
User Reports Fitness
Error Overview

Current Status:

| General Fitness | User Reports | Network Issues |
|---|---|---|
| OK | OK | OK |

Affected Builds

Open affected builds list

Usage

How to read: A perfect Plugin has a 'General Fitness', a 'User Report Fitness' and a 'Connection Fitness' of 10000. The lower the curves, the more problems occur. Keep an eye on the green lines. If they are below both red ones - we have a problem!

General Fitness

*Figure 6: Exemplary PASC dashboard for the zippyshare.com - Free PASC (Rechenmacher, 2014)*

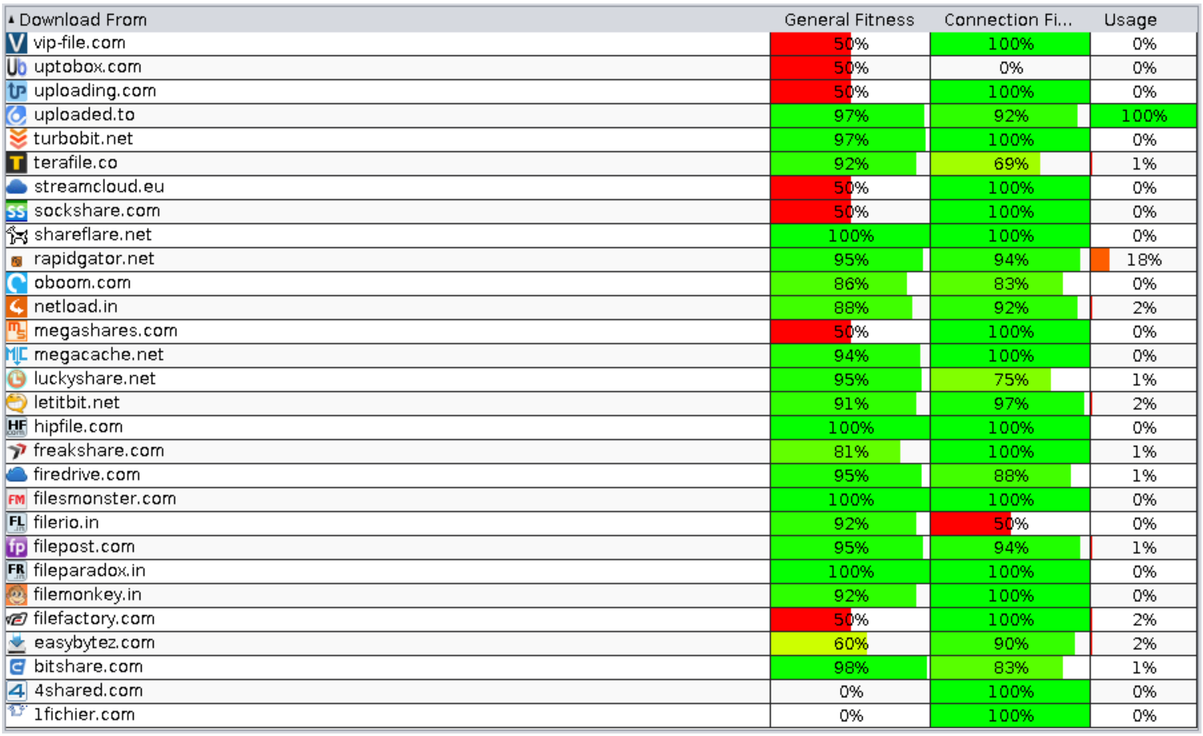
In addition, for each recorded errorID (see Table 2) an error issue is automatically opened in Redmine by the SSE service. These errors are linked to their respective PASC state overviews and vice versa. These records contain the following information:

| | |
|---|---|
| **Status** | "New" for still occurring errors, else "Closed" |
| **Priority** | Increasing priority with higher reporting frequency |

| | |
|---|---|
| **Revision** | Last revision of the plugin in version control |
| **Related Issues** | List of different error IDs with identical cleaned stack traces (these are automatically created because of changed line numbers in the stack trace) |
| **Stack Trace** | Stack trace including code line numbers |
| **Cleaned Stack Trace** | Stack trace with code snippets instead of line numbers |
| **Application Log IDs** | Log IDs collected for this error |
| **Error Chart** | A graph that shows when the error first appeared and if it still exists |

*Table 5: Error record data*

Apart from the developer-oriented PASC status pages, there is a dashboard aimed at users that provides a high-level overview on general and connection fitness for plugins that can be selected by the users. Figure 7 is an example for this type of dashboard.



| ▴ Download From | General Fitness | Connection Fi... | Usage |
|---|---|---|---|
| vip-file.com | 50% | 100% | 0% |
| uptobox.com | 50% | 0% | 0% |
| uploading.com | 50% | 100% | 0% |
| uploaded.to | 97% | 92% | 100% |
| turbobit.net | 97% | 100% | 0% |
| terafile.co | 92% | 69% | 1% |
| streamcloud.eu | 50% | 100% | 0% |
| sockshare.com | 50% | 100% | 0% |
| shareflare.net | 100% | 100% | 0% |
| rapidgator.net | 95% | 94% | 18% |
| oboom.com | 86% | 83% | 0% |
| netload.in | 88% | 92% | 2% |
| megashares.com | 50% | 100% | 0% |
| megacache.net | 94% | 100% | 0% |
| luckyshare.net | 95% | 75% | 1% |
| letitbit.net | 91% | 97% | 2% |
| hipfile.com | 100% | 100% | 0% |
| freakshare.com | 81% | 100% | 1% |
| firedrive.com | 95% | 88% | 1% |
| filesmonster.com | 100% | 100% | 0% |
| filerio.in | 92% | 50% | 0% |
| filepost.com | 95% | 94% | 1% |
| fileparadox.in | 100% | 100% | 0% |
| filemonkey.in | 92% | 100% | 0% |
| filefactory.com | 50% | 100% | 2% |
| easybytez.com | 60% | 90% | 2% |
| bitshare.com | 98% | 83% | 1% |
| 4shared.com | 0% | 100% | 0% |
| 1fichier.com | 0% | 100% | 0% |

*Figure 7: Overview fitness dashboard (Rechenmacher, 2014)*

### 2.5.4 Evaluation of the effectiveness of the JDownloader Immune System

As mentioned earlier, comparisons of the JDownloader software before and after the introduction of the immune system cannot be easily made as old versions continue to be used in parallel and error reports cannot be easily attributed to each version. Nevertheless, statistical data proves the usefulness of the immune system. We identify two relevant observations: StatServ dramatically increases the visibility of errors in rarely used plugins and it significantly shortens the time until severe errors are detected.

*Amount of Errors detected*

In the observation period, the JDownloader Immune System created 103,231 unique error IDs from 2,800 PASCs that were condensed to 3,534 relevant errors in the bug-tracking system. Out of these 3,534 relevant errors, only 50 have been assigned a priority of normal or higher amounting to only 1.4% of errors. We thus conclude that the JDownloader Immune System performs exceptionally well for detecting rare error conditions and edge cases that users seldom experience and report.

*Error Detection Speed compared to User reports*

More interesting than identifying errors hardly anyone ever encounters is the speed at which severe errors are detected. To evaluate this, we manually collected the 41 most severe bugs over several weeks and measured the response times through traditional channels (report functionality within the application and support forums) and via the StatServ Immune System. Table 6 compares the likeliness for a bug to be reported through the StatServ appears to be 16 times faster for the most critical issues.

| | **StatServ** | **Board Reports** |
|---|---|---|
| Reported within 15 hours probability | 50.4% | 13.6% |
| Reported within 24 hours probability | 91.3% | 21.3% |
| Average report duration | 10 hours 9 minutes | 6 days 21 hours 49 minutes |

*Table 6: Error detection metrics*

Since these numbers only refer to the most critical issues, we expect the difference in detection speed to be even more significant for less common issues that often are not reported at all by users.

## 2.6 Limitations

### 2.6.1 Case unique in some aspects

While we believe that a lot of lessons can be learned from the JDownloader example, it is unique in some aspects. Many issues detected by the immune system are caused by changes to the interfaces of the hosting providers. These changes cannot be tested in advance which makes reactive self-healing particularly attractive in comparison to pre-release testing. We do not find many scenarios in the current research where preventive testing of key functionality is not adequately possible.

### 2.6.2 Only a single case

We would have expected to find research or at least data on similar approaches to validate our findings. Since such research apparently does not yet exist, we are limited to present this example as a single outstanding case for the implementation of a Software Immune System.

### 2.6.3 Lack of quantitative data for comparison

This paper could have been enhanced by the presence of data on JDownloader's record for detecting errors prior to the introduction of the Immune System. The Immune System has been introduced with the JDownloader 2 beta Version and as JDownloader 1 and 2 continue to exist simultaneously, there is no straightforward way to differentiate bugs generated via the old system from bugs identified by the new Immune System. Thus, it is not possible to formulate previous system behavior as a null hypothesis and measure the actual improvement generated through the introduction of the new Immune System.

## 2.7 Conclusion

In this paper, we presented a case on how an assisted self-healing software system can facilitate Continuous Deployment and Rapid Release practices. Specifically, the immune system provides earlier failure detection (in the case of JDownloader, 16 times faster) enabling more rapid bug fixes through Continuous Deployment practices. While adequately testing software remains a vital process for software quality, we show that monitoring production software through a Software Immune System can be an additional important pillar for quality. One unique aspect of our model is that the thresholds for determining failure state are generated and adjusted dynamically across plugins and within the lifecycle of one specific plugin. We believe that this flexibility can be a useful insight and be a useful resource for further research and practical applications.

# 3 Elaboration

## 3.1 Additional relevant literature

The main section contains the most relevant literature regarding the JDownloader Immune System. However, the JDownloader Immune System could also be relevant in the context of DevOps. Should the paper be presented in this slightly shifted context, the information provided here might prove to be relevant.

### 3.1.1 DevOps

DevOps, meaning the close collaboration between development and IT operations (Dyck et al., 2015), is a relatively new trend among software practitioners and in software research. For the intended collaboration between Dev and Ops, accurate data about system behavior in production is crucial (Erich, 2014; Roche, 2013). Zhu et al. (2016) stress the importance of adequate tooling and monitoring capabilities to benefit from DevOps. The JDownolader Immune System facilitates exactly that: Quickly identifying abnormal state in production plugins allowing developers to react and fix issues rapidly.

## 3.2 Additional Comments on the JDownloader Immune System

Chapter 2 contains the generally most compelling description of the JDownloader Immune System. This section here contains logically concise snippets on niche aspects of the JDownloader Immune System that can be hot-swapped into the main text body by an editor depending on the context in which the paper will be submitted.

### 3.2.1 More information on J.D. Brutlag's Forecasting model

Most of the math presented in this paper is based on a model developed by Jake D. Brutlag (2000) for RRDTool, a system to help network technicians to detect errors. The model takes 4 distinct characteristics into account:

1. Long-term trend (e.g. server load increases over several months)
2. Seasonal cycle (e.g. daily traffic peaks in the evening)
3. Seasonal variability (e.g the average variability is not constant over a day)
4. Gradual change of 2 and 3 over time (e.g. the daily traffic peaks change due to daylight changes through the year)

From these four trends, Brutlag splits the time series into three components: The baseline, a linear trend and a seasonal trend. These components combined are used to give a prediction for the next observation interval. If the recorded data differs from the prediction by more than a pre-defined threshold (the so-called “confidence band”), an error event is triggered. Brutlag calculated an upper and a lower confidence band. For JDownloader, the upper band is not relevant (a plugin cannot work “too good”) and only a bottom threshold is determined. Evan Miller (2007) extended the model for “IMVU, Inc.”[9] to operate better for low volume time series. JDownloader's implementation is closer to Brutlag's original model.

[9] http://imvu.com

### 3.2.2 Visualization for Perfect Level Detection and Trend Indicator Line determination

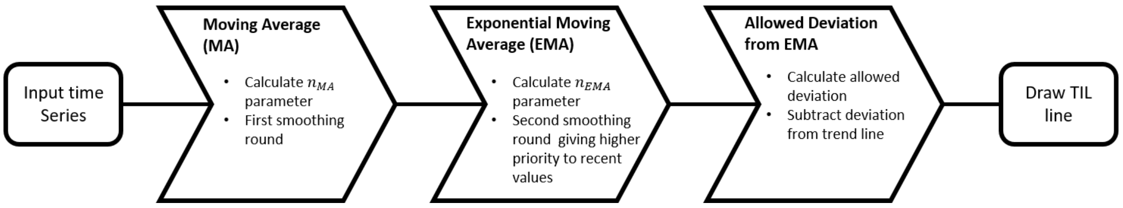

Input time Series
Moving Average (MA)
Calculate $n_{MA}$ parameter
First smoothing round
Exponential Moving Average (EMA)
Calculate $n_{EMA}$ parameter
Second smoothing round giving higher priority to recent values
Allowed Deviation from EMA
Calculate allowed deviation
Subtract deviation from trend line
Draw TIL line

*Figure 8: Trend Indicator Line (TIL) calculation*

Figure 7 summarizes the process to calculate the Trend Indicator Line (TIL) outlined in chapter 2.

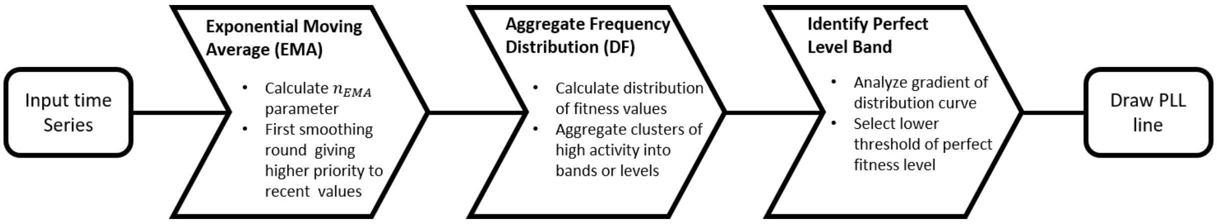

Input time Series
Exponential Moving Average (EMA)
Calculate $n_{EMA}$ parameter
First smoothing round giving higher priority to recent values
Aggregate Frequency Distribution (DF)
Calculate distribution of fitness values
Aggregate clusters of high activity into bands or levels
Identify Perfect Level Band
Analyze gradient of distribution curve
Select lower threshold of perfect fitness level
Draw PLL line

*Figure 9: Perfect Level Line (PLL) calculation*

Correspondingly, Figure 8 shows the calculation process for the Perfect Level Line (PLL).

These two graphical representations can be used to summarize the calculations without diving into the underlying mathematics and can help to easily explain the concept.

### 3.2.3 Seasonal Cycles in the data

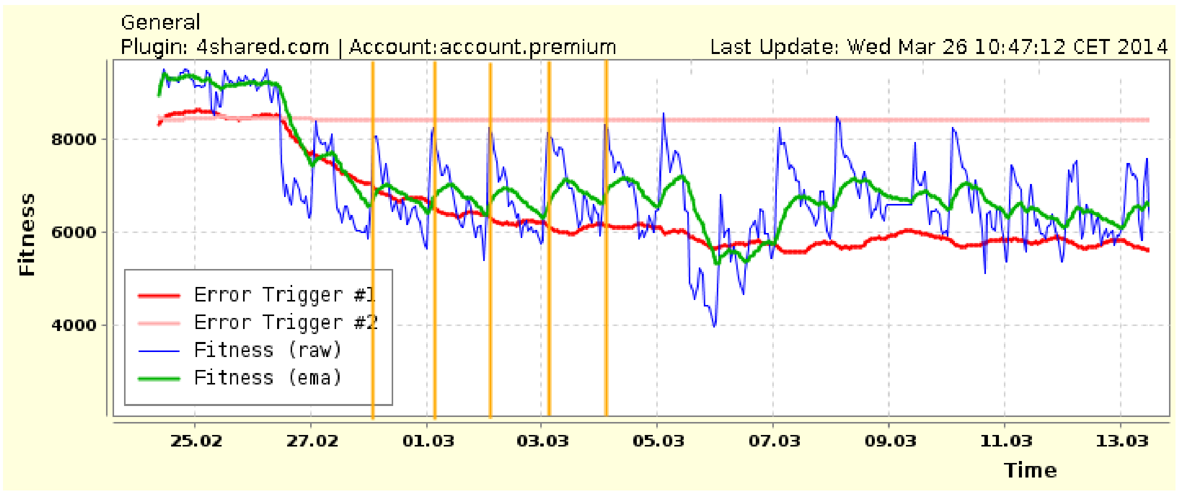


*Figure 10: Seasonal cycles in the data (Rechenmacher, 2014)*

An interesting phenomenon when calculating fitness data for a specific PASC can be seasonal cycles within the fitness time series. One possible explanation could be that certain errors only occur in specific time-zones (think government firewalls, etc.). A possible solution could be to collect separate fitness time series for each time zone and calculate differing threshold values. The downside would be additional processing resources.

# Appendix A Equations used in the JD Immune System

*Equation 1: Basic usage definition*

$$usage = \sum_{t=0}^{1\ hour} Download\ attempt$$

*Equation 2: Basic error definition*

$$errors = \sum_{t=0}^{1\ hour} Failed\ Download\ attempt$$

*Equation 3: Basic fitness definition*

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

*Equation 4: Network fitness definition*

$$networkFitness = \ 10{,}000 * \frac{usage - (results_{CONNECTION\_ISSUES} + results_{CONNECTION\_UNAVAILABLE})}{usage}$$

*Equation 5: Plugin fitness definition*

$$pluginFitness = 10{,}000 * \frac{usage - results_{PLUGIN\_DEFECT}}{usage}$$

*Equation 6: Finished fitness definition*

$$finishedFitness = 10{,}000 * \frac{results_{FINISHED}}{usage}$$

*Equation 7: Plugin fitness factor*

$$pluginFitnessFactor = \frac{9}{\frac{pluginFitness}{2{,}500} + 0.5}$$

*Equation 8: Finished fitness factor*

$$finishedFitnessFactor = \frac{1}{\frac{finishedFitness}{2{,}500} + 0.5}$$

*Equation 9: Complete fitness definition*

$$fitness = \frac{pluginFitnessFactor * pluginFitness + finishedFitnessFactor * finishedFitness}{pluginFitnessFactor + finishedFitnessFactor}$$

*Equation 10: Moving average formula*

$$ma_{n_{MA}}^{offset}(t) = \frac{1}{n_{MA}} \sum_{i=1}^{n_{MA}} x(t - n_{MA} + i + offset)$$

*Equation 11: Moving average dynamic model parameter*

$$n_{MA}\left(avg_{usage}\right) = \ 6 + f(avg_{usage})$$

*Equation 12: Smoothing for low usage numbers*

$$f(usage) = \frac{2{,}500}{usage + 5}$$

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

*Equation 13: Exponential moving average formula*

$$ema(t+1) = \beta * x(t) + (1 - \beta) * ema(t)$$

*Equation 14: Decay rate for old values*

$$\beta = \frac{2}{n_{EMA} + 1}$$

*Equation 15: Exponential moving average with decay rates*

$$ema_{n_{EMA}}(t+1) = \frac{2}{n_{EMA} + 1} * x(t) + \left(1 - \frac{2}{n_{EMA} + 1}\right) * ema_{n_{EMA}}(t)$$

*Equation 16: Dynamic model parameter for exponential moving average*

$$n_{EMA}\left(avg_{usage}\right) = 180 + 2 * f(avg_{usage})$$

*Equation 17: Allowed deviation from the trend line*

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

*Equation 18: Moving variability formula*

$$mv_{n_{VA}}^{offset}(t) = \frac{1}{n_{VA}} \sum_{i=1}^{n_{VA}} |x(t - n_{VA} + i + offset) - ma_{VA}(t - n_{VA} + i + offset)|$$

*Equation 19: Distribution function for values of the time series*

$$DF(ema(t)) = \sum_{i=1}^{\mathrm{n}} 1\{x_i = ema(t)\}$$

*Equation 20: Reports fitness formula*

$$reportsFitness = 10{,}000 - \frac{10{,}000 * reportCounter}{0.86\% * usage}$$

# References

Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time series analysis: forecasting and control*. John Wiley & Sons.

Brutlag, J. D. (2000, December). Aberrant Behavior Detection in Time Series for Network Monitoring. In *LISA* (Vol. 14, pp. 139-146).

Chen, G., Jin, H., Zou, D., Zhou, B. B., & Qiang, W. (2015). A lightweight software fault-tolerance system in the cloud environment. *Concurrency and Computation: Practice and Experience*, *27*(12), 2982-2998.

Clark, S., Collis, M., Blaze, M., & Smith, J. M. (2014, November). Moving targets: Security and rapid-release in firefox. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1256-1266). ACM.

da Costa, D. A., McIntosh, S., Kulesza, U., & Hassan, A. E. (2016, May). The impact of switching to a rapid release cycle on the integration delay of addressed issues: an empirical study of the Mozilla Firefox project. In *Proceedings of the 13th International Conference on Mining Software Repositories* (pp. 374-385). ACM.

Dyck, A., Penners, R., & Lichter, H. (2015, May). Towards definitions for release engineering and devops. In *Proceedings of the Third International Workshop on Release Engineering* (pp. 3-3). IEEE Press.

Erich, F., Amrit, C., & Daneva, M. (2014). Report: Devops literature review. *University of Twente, Tech. Rep*.

Feitelson, D. G., Frachtenberg, E., & Beck, K. L. (2013). Development and deployment at facebook. *IEEE Internet Computing*, *17*(4), 8-17.

Ghosh, D., Sharman, R., Rao, H. R., & Upadhyaya, S. (2007). Self-healing systems—survey and synthesis. *Decision Support Systems*, *42*(4), 2164-2185.

Goble, S. (2016, Ocober 7$^{th}$), *Sally Goble – The Guardian Head of Quality @Porto Tech Hub Conference 2017* [Video file]. Retrieved from https://www.youtube.com/watch?v=852OVo6HzcI

Gross, K. C., Urmanov, A., Votta, L. G., McMaster, S., & Porter, A. (2006, March). Towards dependability in everyday software using software telemetry. In *Engineering of Autonomic and Autonomous Systems, 2006. EASe 2006. Proceedings of the Third IEEE International Workshop on* (pp. 9-18). IEEE.

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

Hellerstein, J. L., Zhang, F., & Shahabuddin, P. (2001). A statistical approach to predictive detection. *Computer Networks*, *35*(1), 77-95.

Ivan, I., Boja, C., & Zamfiroiu, A. (2012). Self-Healing for Mobile Applications. *Journal of Mobile, Embedded and Distributed Systems*, *4*(2), 96-106.

Jiang, Z., Scheibe, K. P., Nilakanta, S., & Qu, X. S. (2017). The Economics of Public Beta Testing. *Decision Sciences*, *48*(1), 150-175.

Karvonen, T., Behutiye, W., Oivo, M., & Kuvaja, P. (2017). Systematic Literature Review on the Impacts of Agile Release Engineering Practices. *Information and Software Technology*.

Khomh, F., Adams, B., Dhaliwal, T., & Zou, Y. (2015). Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, *20*(2), 336-373.

Khomh, F., Dhaliwal, T., Zou, Y., & Adams, B. (2012). Do faster releases improve software quality?: an empirical case study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (pp. 179-188). IEEE Press.

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

Laukkanen, E., Itkonen, J., & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, *82*, 55-79.

Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., & Petersen, K. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, *20*(5), 1384-1425.

McIlroy, S., Ali, N., & Hassan, A. E. (2016). Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering*, *21*(3), 1346-1370.

Miller, E. (2007). Holt-winters forecasting applied to poisson processes in real-time (DRAFT). *IMVU, Inc., Oct*, *28*, 10.

Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., & Poshyvanyk, D. (2016, April). Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on* (pp. 33-44). IEEE.

Neely, S., & Stolt, S. (2013, August). Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference (AGILE), 2013* (pp. 121-128). IEEE.

Olsson, H. H., & Bosch, J. (2014). Post-deployment data collection in software-intensive embedded products. In *Continuous software engineering* (pp. 143-154). Springer International Publishing.

Rechenmacher, T. (2014). The JDownloader Continuous Deployment Immune System. Diploma Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg.

Roche, J. (2013). Adopting DevOps practices in quality assurance. *Communications of the ACM*, *56*(11), 38-43.

Rodriguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., ... & Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, *123*, 263-291.

Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T., & Stumm, M. (2016, November). Continuous deployment of mobile software at facebook (showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 12-23). ACM.

Sahasrabudhe, M., Panwar, M., & Chaudhari, S. (2013, September). Application performance monitoring and prediction. In *Signal Processing, Computing and Control (ISPCC), 2013 IEEE International Conference on* (pp. 1-6). IEEE.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., & Stumm, M. (2016, May). Continuous deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 21-30). ACM.

Sharifi, M., Ramezani, S. B., & Amirlatifi, A. (2012). Predictive Self-Healing of Web Services Using Health Score. *Journal of Web Engineering*, *11*(1), 79.

Silva, L. M. (2008, July). 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.

Suonsyrjä, S., Hokkanen, L., Terho, H., Systä, K., & Mikkonen, T. (2016, October). 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.

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*.

Vogler, W. (2014, November 12). The Story of Apollo – Amazon’s Deployment Engine [Blog Post]. Retrieved from: http://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html

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

Wang, L., Wang, H., Yu, Q., Sun, H., & Bouguettaya, A. (2013, December). Online reliability time series prediction for service-oriented system of systems. In *International Conference on Service-Oriented Computing* (pp. 421-428). Springer Berlin Heidelberg.

Ye, F., Wu, S., Huang, Q., & Wang, X. A. (2016, September). The Research of Enhancing the Dependability of Cloud Services Using Self-Healing Mechanism. In *Intelligent Networking and Collaborative Systems (INCoS), 2016 International Conference on* (pp. 130-134). IEEE.