Technical Disclosure Commons

Defensive Publications Series

August 26, 2019

SERVICE REQUEST SPEND BASED DATA-DRIVEN TECHNICAL DEBT CHARACTERIZATION IN COMPLEX SOFTWARE SYSTEMS

Abhishek Pathak

Kaarthik Sivakumar

Laxmi Mukund

Ashok Gowda

SERVICE REQUEST SPEND BASED DATA-DRIVEN TECHNICAL DEBT CHARACTERIZATION IN COMPLEX SOFTWARE SYSTEMS

AUTHORS:
Abhishek Pathak
Kaarthik Sivakumar
Laxmi Mukund
Ashok Gowda

ABSTRACT

Techniques are described herein for a data-driven Technical Debt (TD) analytics platform that allows executives and product development teams to optimally track, manage, and repay TD early in the software development life cycle. This may improve team productivity, address the problem of software TD holistically in the code-to-customer lifecycle, and yield long term benefits. Thus, the platform provides additions to savings, improved customer experience, and enhanced serviceability.

DETAILED DESCRIPTION

Complex software systems cater to a wide range of products, often having millions of lines of code and integrated with different models of operating systems (e.g., non-preemptive, multithreaded, multi-core, etc.). These software systems offer highly complicated feature sets with real-time functionality and high availability running constantly at varying performance rates. Further contributing to this complexity are the features that are continuously added by thousands of developers generating hundreds of thousands of Lines of Code (LoC) every week. The maintenance, quality, and reliability of such systems require intricate processes in place. Reducing the Technical Debt (TD) and improving quality becomes a challenge.

Without knowledge of and attention to the underlying characteristics of such highly complex software entities, traditional attempts to improve quality and reduce TD actually backfire. Instead of improving the state of the system, it reduces the serviceability and maintainability of the software system and lowers quality in unforeseen ways. This also increases cost and business risk in spite of developer efforts to improve code quality and reduce service cost.

The methodology for characterizing code level TD as described herein is provided as follows. As a preliminary matter, TD is defined as the service spend due to quality tagged Service Requests (SRs). This is a departure from current approaches in which TD remains a vague and heavily abstract term. The spend on each SR may be calculated and identified by Customer Experience (CX) models. Each SR includes a list of bugs attached thereto to help resolve the SR. Each bug that is in a Modified, Resolved or Verified (MRV) state is analyzed further for its attachments to retrieve the list of files that were modified in order to tend to that bug. The cost of each bug attached to the SR may be obtained by aggregating the cost of each SR linked to the bug. Next, a cost modelling technique may be used to generate a split cost to assign to each file modified for tending to a given bug. The cost modelling technique may employ a static analysis warning count and coverage to factor and assign the split cost. This split file-level cost may serve as the dependent variable in training one or more Machine Learning (ML) models.

Various feature/predictors may be identified at the file level, including:

1. Code Level: Halstead effort, Halstead errors, cyclomatic complexity, statement path count, number of forward edges, operand count, operation count, block count, backedge count, path count.
2. Code Coverage: covered lines of code, total lines of code, coverage.
3. Static Analysis Warning Count: fixed, dismissed, outstanding, and total counts.
4. Bug: various tags in bugs are retrieved to categorize the bugs based on their functional impacts (e.g., code quality, security, maintainability, serviceability, etc.).
5. Dynamic Analysis and Call Graph Metrics: obtained from periodic static analysis runs.
6. Source Control Management (SCM) and Code History
7. Telemetry: Syslog-based telemetry may be used to arrive at execution counts at the source code component level or file/function level. This may add a separate dimension (e.g., a dynamically sampled weight as a multiplication factor to the TD in the piece of code) to filter and report a

stronger TD for heavily executed pieces of code and a weaker TD for lightly executed pieces of code.

Feature selection and dimensionality reduction may be performed using the Spearman's Rank correlation coefficient between each predictor and the dependent variable. Correlation coefficients may be ranked between each predictor at file level and SRs associated therewith. Using the deep learning method of Generative Adversary Networks (GANs), a mapping may be established between the features and the dependent variable. This method may be enhanced to employ reinforcement learning with the end goal of minimizing quality issue related service requests to account for the multi-stage nature of software processes.

A list of all analyzed files from periodic baseline runs may be obtained via static analysis. This may serve as a better input than source code pulled to obtain a list of files. Using this prediction provided by the ML model, the cost of modifying any given file may be predicted based on the feature vector of its predictor (e.g., static analysis warnings count, cyclomatic complexity, etc.). This cost may be aggregated at component, manager, director, vice president, product, product family, and organization level to identify the TD at each level.

TD trends may be reported based on the trajectory / temporal state variations of the feature/predictor vector. This shows improvements/degradations over time and prediction trends. The optimization method of stochastic gradient descent or Nesterov accelerated gradient may be used to enable optimization in weights in the deep learning process to provide the feature vector that minimizes the TD in a given file or component.

In a further extension, a user interface may be provided with sliders whereby an executive may change the slider to view how reducing each predictor would change the TD in a given source component (e.g., reducing cyclomatic complexity by 125 may yield some change in TD). This may be derived using the ML predictive model. Furthermore, the TD slider in the user interface may enable executives to see the changes in the feature vector required to achieve that goal. This may be derived using an optimization algorithm on the deep learning process. This may help meet organization policy and goals of TD management (e.g., reducing and limiting TD per file to $1000).

Separate cost modelling may also be performed to ascertain the phase propagation of cost. This may indicate, for example, the cost to fix a software defect related to buffer overflow and how it changes if fixed during development, code commit time, post commit baseline static analysis warning scan, pre-release or post release.

This may enable characterization of the cost to fix a defect (as opposed to leaving the defect unfixed), and the Return on Investment (RoI) from fixing it. This enables profit and loss style reporting to aid executives in decision making as to where to direct resources to best align with business risks and targets at the organization level. This may also help prioritize the repayment of TD in product code.

For newly written code, the aforementioned indicators of quality may be used to provide predictions in the form of leading indicators of quality by predicting the TD introduced in the developer's code while the code is being written, for the change-set, and then provide recommendations to reduce the TD before code with high TD is committed to the product base.

A Representational State Transfer (ReST) Application Programming Interface (API) server(s) capable of handling a large number of parallel requests may be established to provide the TD vector for any source file. This may aid in understanding before-and-after scenarios and articulating benefits of branch level refactoring efforts.

Figures 1-3 below illustrate charts/diagrams that show the results and deliverables in a live dashboard. In particular, these charts/diagrams explain the code level debt characterization expressed as a monetary value in terms of dollars ($).

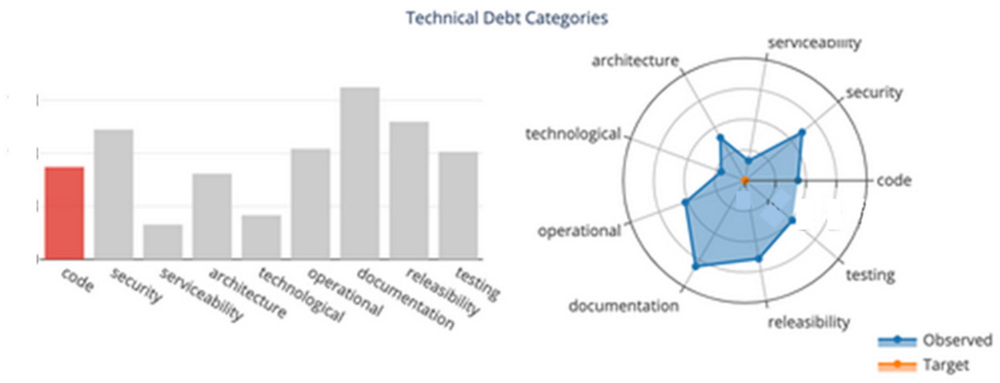


Figure 1

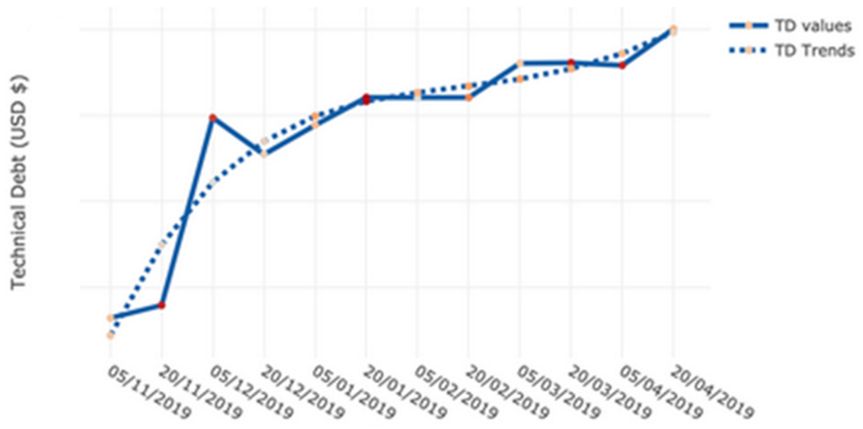
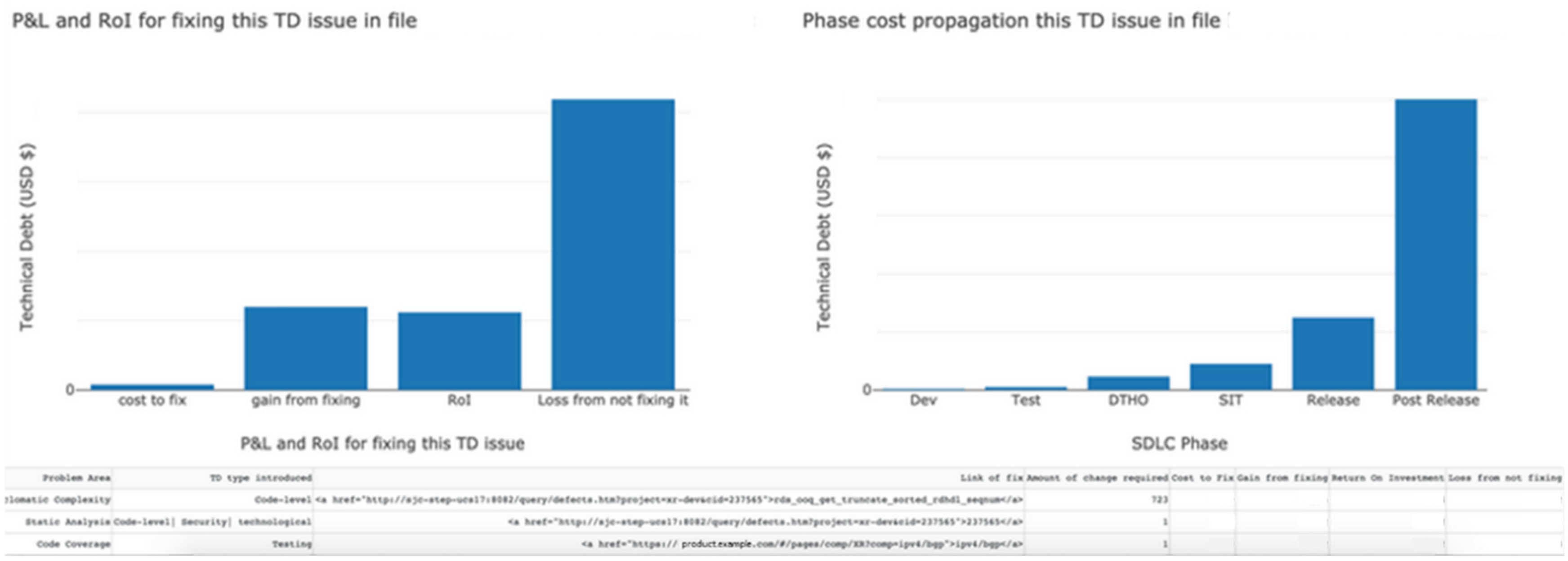

TD values
TD Trends
Technical Debt (USD $)
05/11/2019
20/11/2019
05/12/2019
20/12/2019
05/01/2019
20/01/2019
05/02/2019
20/02/2019
05/03/2019
20/03/2019
05/04/2019
20/04/2019

Figure 2



Problem Area	TD type introduced	Link of fix	Amount of change required	Cost to Fix	Gain from fixing	Return On Investment	Loss from not fixing
:lomatic Complexity	Code-level	<a href="http://sjc-step-ucs17:8082/query/defects.htm?project=xr-dev&cid=237565">rds_ooq_get_truncate_sorted_rdhdl_seqnum</a>	723				
Static Analysis	Code-level| Security| technological	<a href="http://sjc-step-ucs17:8082/query/defects.htm?project=xr-dev&cid=237565">237565</a>	1				
Code Coverage	Testing	<a href="https:// productexample.com/#/pages/comp/XR?comp=ipv4/bgp">ipv4/bgp</a>	1				

Figure 3

Figure 4 below illustrates a sample organizational chart view of TS costs mapped onto an organizational network. The TD numbers are based on a small representative sample.

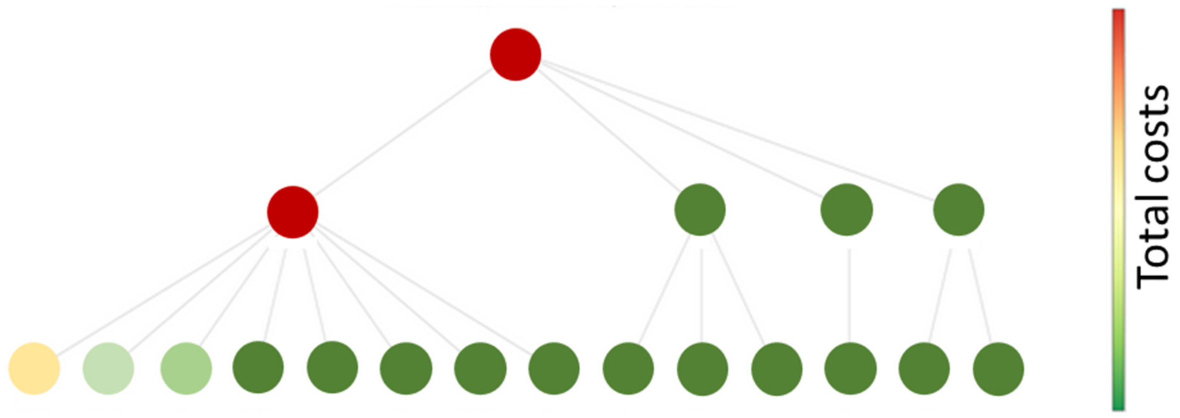


Figure 4

Figure 5 below illustrates a geographic spread chart of TD based on customer geography.

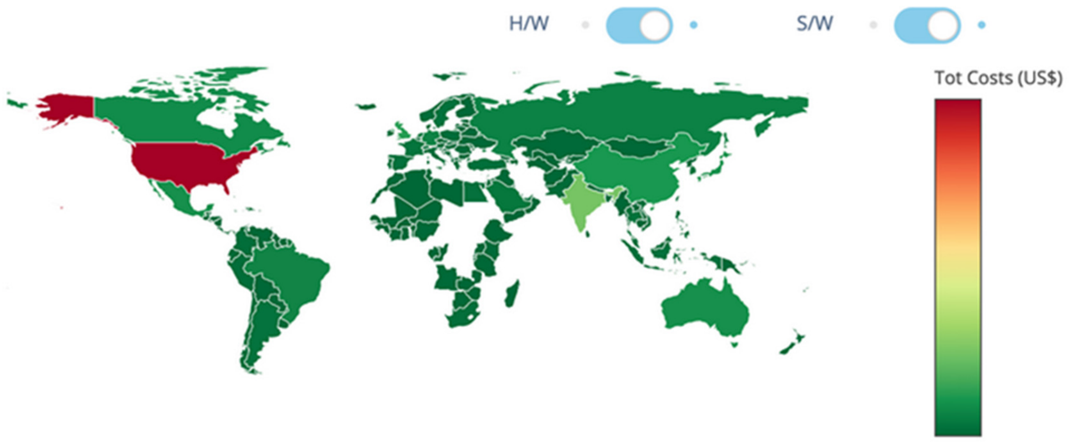


Figure 5

Figure 6 below illustrates an example chart showing the product TD quarterly trend.

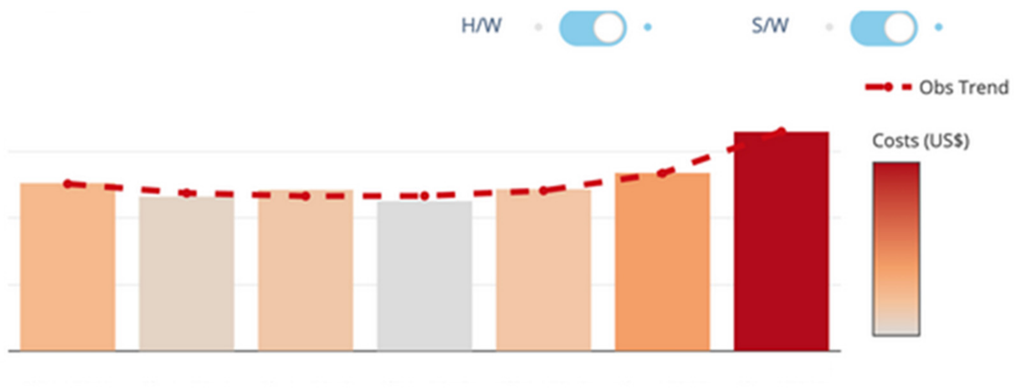


Figure 6

Techniques described herein may address the problem of TD mitigation through a developed model and associated software processes to take SR spends and various predictors contributing to the TD. This may enable newly written code to provide prescriptive insights as the code is being written.

The platform has several contributions to TD characterization, measurement, and management. One contribution is the establishment of a fully connected data-driven pipeline connecting code, static analysis at check-in time, static analysis during periodic (e.g., monthly) baseline runs, bug tracking systems, SRs, service contracts, service spends in terms of dollars, telemetry (where possible), and the customer. Thus, the platform may fully connect the code to the customer.

Others contributions include the mathematical model of characterization of TD along with a formal definition of TD and feature selection for various predictors. Furthermore, various traditional indicators of code quality and TD may be correlated. The correlation studies and analysis amongst different predictors may be correlated with an outcome (e.g., TD). Also, a basic recommender system to optimize gains from repaying TD with respect to developer efforts may improve prioritization of TD mitigation and resolution to promote lean development. These techniques promote a "shift left" mentality wherein resolving TD earlier is easier, cheaper, and more effective than post mortem style analyses. In addition to characterizing code level TD directly in terms of dollars, the platform may also address security concerns.

There is a desire in the TD community to model costs and tie TD to business risk. Tying TD to business risk and cost in terms of dollar values is deeply desired but not readily available as a debt analytics platform. Executives and developers gain by performing cost-benefit analysis / profit and loss accounting style metrics / RoI associated with a given changeset (e.g., fixing a static analysis warning immediately as opposed to fixing it later). TD should be treated as a collaborative industrial research problem between software economics/econometric/financial cost-modelling and software process research.

In summary, techniques are described herein for a data-driven TD analytics platform that allows executives and product development teams to optimally track, manage, and repay TD early in the software development life cycle. This may improve team productivity, address the problem of software TD holistically in the code-to-customer lifecycle, and yield long term benefits. Thus, the platform provides additions to savings, improved customer experience, and enhanced serviceability.