A Real-time Framework for Detecting Efficiency Regressions in a Globally Distributed Codebase

Martin Valdez-Vivas
Facebook
Menlo Park, CA, USA
mvv@fb.com

Caner Gocmen
Facebook
Menlo Park, CA, USA
caner@fb.com

Andrii Korotkov
Facebook
Menlo Park, CA, USA
partisan@fb.com

Ethan Fang
Facebook
Menlo Park, CA, USA
ethanfang@fb.com

Kapil Goenka
Facebook
Menlo Park, CA, USA
kgoenka@fb.com

Sherry Chen
Facebook
Menlo Park, CA, USA
zhc047@fb.com

ABSTRACT

Multiple teams at Facebook are tasked with monitoring compute and memory utilization metrics that are important for managing the efficiency of the codebase. An efficiency regression is characterized by instances where the CPU utilization or query per second (QPS) patterns of a function or endpoint experience an unexpected increase over its prior baseline. If the code changes responsible for these regressions get propagated to Facebook's fleet of web servers, the impact of the inefficient code will get compounded over billions of executions per day, carrying potential ramifications to Facebook's scaling efforts and the quality of the user experience. With a codebase ingesting in excess of 1,000 diffs across multiple pushes per day, it is important to have a real-time solution for detecting regressions that is not only scalable and high in recall, but also highly precise in order to avoid overrunning the remediation queue with thousands of false positives. This paper describes the end-to-end regression detection system designed and used at Facebook. The main detection algorithm is based on sequential statistics supplemented by signal processing transformations, and the performance of the algorithm was assessed with a mixture of online and offline tests across different use cases. We compare the performance of our algorithm against a simple benchmark as well as a commercial anomaly detection software solution.

CCS CONCEPTS

• **Computer systems organization** → **Maintainability and maintenance**; *Real-time systems*; • **Software and its engineering** → **Operational analysis**; • **Information systems** → *Decision support systems*; • **Computing methodologies** → *Anomaly detection*; • **Mathematics of computing** → Time series analysis;

KEYWORDS

regression detection; systems data science; codebase efficiency; code efficiency monitoring; resource utilization monitoring; continuous push; change point detection; anomaly detection; CUSUM methods

ACM Reference Format:
Martin Valdez-Vivas, Caner Gocmen, Andrii Korotkov, Ethan Fang, Kapil Goenka, and Sherry Chen. 2018. A Real-time Framework for Detecting Efficiency Regressions in a Globally Distributed Codebase. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3219819.3219858

1 INTRODUCTION

Facebook is the world's largest social media platform, with over 2 billion people accessing the site each month [17]. As such, it runs one of the largest and most complex distributed infrastructures in the world [1, 7, 21]. At this scale, operating the site efficiently is important to sustain user growth while keeping the increasing resource demands associated with hosting new users and services within manageable levels. Code changes that degrade efficiency from a compute or memory utilization perspective for some element of the codebase are said to have caused an efficiency regression.

The Efficiency Infra team at Facebook monitors the global CPU share (gCPU) for functions and endpoints in the WWW codebase which serves the desktop version of the site as well as client apps. The team's primary charter is to detect gCPU regressions, diagnose the root cause, and work with code owners to clean up or roll back regressive code changes. Similarly, the Core Data Demand team monitors QPS patterns across functions and endpoints that run realtime queries against Facebook's social graph. Both tasks fundamentally amount to finding sustained deviations over a prior baseline across a collection of time series, and leveraging that information to triage and roll back inefficient code changes.

At the scale of Facebook's codebase, it is infeasible to manually review even a modest fraction of these efficiency-related time series. Adding to the challenge, in late 2016 Facebook moved 100% to a quasi-continuous push model [18, 19], which means the company transitioned from a weekly push cycle to pushing code changes to the site multiple times in a day. A highly precise automated regression detection mechanism is therefore not only advantageous, but complementary to the company's engineering development and release model.

Real-time anomaly detection covers a broad range of approaches that can generally be categorized into three main subdomains [8]. Point anomalies are sudden, outsized spikes or dips; pattern anomalies are a run of points which taken collectively have characteristics dissimilar to historical data; and change point anomalies are sustained level shifts in statistical attributes of the series (mean, variance, and/or seasonality). The central challenge in anomaly detection is that the definition of success for a detection framework hinges on which classes of anomalies are considered detrimental, and even within a family of use cases this can vary across different objectives. For example, the traffic team at Facebook is interested in catching point anomalies which could be an early indication of an event like a DDoS attack, whereas in the performance regression domain certain endpoints can exhibit periodic spikes as part of normal behavior. Regressions can also arise and fix themselves without any intervention from the efficiency teams, and surfacing these instances is likewise distracting and counterproductive. In our use case, we are strictly interested in catching change point anomalies and filtering out the other two anomaly modes as noise in the system.

Detecting change point anomalies is a well-studied problem [2]. At a high level, change point detection can be separated into offline and real-time or online algorithms. Offline algorithms are used in batch processing to identify change points retroactively in a historical data set. Online algorithms are oriented towards monintoring and catching change points as they occur. In practice the notion of real-time for change point detection algorithms is different and not as absolute as real-time detection for point anomalies, as some observations after the change point are required to establish the persistence of a level shift. The present system is designated as real-time in the sense of being invoked or activated at regular intervals with the utility of the operation depending on its logical correctness as well as the time in which it is performed, which coincides with the established definition of real-time computing systems [20].

Change point algorithms can further be subdivided into statistical rule based and machine learning based approaches. Machine learning based approaches project anomaly detection as a binary classification problem. While machine learning has successfully been used for other time series applications such as forecasting [15], these methods require careful feature mining and a large volume of labeled examples to perform well [24]. In real-time monitoring for efficiency regressions, there is neither a natural well-defined set of predictive features nor a prelabeled training set. At best, thousands of examples would need to be collected and labeled by hand to construct the training set required for a supervised learning model. Statistical rule based approaches can alternately be grouped into parametric and non-parametric methods [13]. Given hundreds of thousands of time series with a wide degree of variance profiles, it is difficult to justify consistent assumptions about a stationary error distribution required for a parametric model. This consequently led us to non-parametric methods such as the commonly used CUSUM algorithm [10, 14, 16, 22].

An off-the-shelf implementation for CUSUM is available in the `changepoint` package in R [13]. We ran into limitations using a traditional implementation of the CUSUM algorithm due to the assorted scale and variance profiles in the time series we monitor, and the volume of false positives that entails given the scale of Facebook's operations. Our solution relies on combining a scaled CUSUM score with signal processing operators, namely lowpass filtering and Fourier regression, to normalize the input signal. Post-CUSUM, we apply a clustering heuristic based on sampled stack traces that groups together regressions likely to be related, and singles one of them as the probable root cause.

Several other solutions have been proposed for changepoint detection. In general, simpler approaches based on exponential smoothing and ARIMA were not found to perform favorably compared to traditional CUSUM. On the other hand, more sophisticated proposals based on filtering came with limiting assumptions about the behavior of the error process or were tested on generated data [23]. The contribution of the present work however is not to elaborate on different methodologies in this space and draw a comprehensive comparison with regards to their performance, for which multiple surveys already exist [2, 8]. Instead, the aim of the paper is to present a pragmatic and scalable solution to a problem facing every enterprise codebase, substantiate that it can perform above par compared to commercial software solutions, and provide detail on how we tested and tuned the model with samples collected across a very large, unlabeled production dataset.

In Section 2, we describe in more detail the end-to-end system we designed, and how we customize the solution across different customers. Section 3 describes the multiple testing workflows we developed to assess the performance of the algorithm, and Section 4 compares the performance against the traditional CUSUM algorithm and a commercial anomaly detection software solution called Anodot. We conclude with discussion in Section 5.

2 SYSTEM DESIGN

The detection workflow is comprised of the CUSUM algorithm as well as custom signal processing operators to account for periodicity and noise effects. At the end of the section, we describe specific use cases for three individual teams at Facebook, the different challenges they posed, and how we leveraged the aforementioned set of tools to improve detection quality.

2.1 CUSUM

CUSUM (cumulative sum) is a widely-used nonparametric method for detecting change points. For a time series X_t, the CUSUM statistic S_t is derived as follows:

$$\begin{aligned} \bar{x} &= \frac{\sum_{t=1}^{T} X_t}{T} \\ S_t &= \frac{\sum_{t=1}^{T} (X_t - \bar{x})}{T} \end{aligned}$$

The minimum CUSUM statistic designates the point most likely to be a changepoint in the positive direction, and the converse also holds true. The objective then is to find a suitable threshold π to separate actual changepoints from noise.

The CUSUM statistic is scale dependent. In other words, if we uniformally scale a set of time series X_t by an arbitrary factor $\alpha > 0$, the corresponding decision threshold would have to be revised to $\alpha\pi$ to preserve the original outcome. Along similar lines, if the series in X_t have a wide range of scales, a single decision threshold might struggle balancing detection quality at different ends of the spectrum. This is the case in our application. Even though our metrics (compute and memory utilizations for functions and endpoints) are bound between 0 and 1, the utilization distirbution has a long tail, with certain functions having an order of magnitude difference in utilization compared to each other. To address the differences in scale, we added the option to normalize by a scale measure, which in practice is commonly the mean or median of the original series.

Overall, CUSUM is a highly effective method for detecting change points in instances where the variance of the series is small relative to the magnitude of the change point. This is not generally true in practice. In terms of utilizations, transient spikes are in fact expected to arise occasionally during the course of normal operation. A large isolated utilization spike in itself is unlikely attributable to an inefficient code change, but could raise a false alert especially if the jump happens to be close to the end of the time window. To illustrate, consider the series

$$\begin{aligned} X &= (0, 0, 0, 2, 0) \\ Y &= (0, 0, 2, 0, 0) \\ Z &= (0, 0, 0, 1, 1) \end{aligned}$$

X and Y both look like the same transient spike albeit occurring at different periods in the time window, whereas Z looks like a sustained increase and therefore one we would like our system to identify. Whereas the CUSUM statistic of Y is less than the CUSUM of Z, the CUSUM statistics for X and Z are equal. Whatever threshold we set, it will at best result in a false positive or a false negative (the latter being less desirable in our case).

Likewise, seasonality and periodicity effects are present in virtually any globally-distributed production system. Depending on the time of day, a diurnal or weekly cyclical pattern can easily be mistaken for a regression over a small enough time window. Pushes are not scheduled at regular intervals and cyclical peaks take place at different times of the day for different entities, obviating efforts to avoid them by scheduling the detection workflow during "off-peak" hours. The presence of daily and weekly seasonality in particular raise a practical tradeoff in setting the time window for CUSUM. While an input window on the order of days instead of hours can help expose the presence of these effects, it also requires greater resources for data querying and ingestion, which can pose a limiting factor and slow down detection times when running the workflow at scale.

Up to this point, we have only touched on running CUSUM once to detect a single change point. It could happen that a time series has multiple change points. The typical solution in this case is to package CUSUM with a recursive search algorithm like dynamic programming to find the additional change points. We chose not to look for multiple change points in our implementation. Doing so would introduce an additional layer of tuning, without clear gains to our detection rates. We also expect change points are closely tied to code pushes, therefore we do not expect to find multiple level shifts in a time window that has a single push.

2.2 Time Series Operators

As discussed above, two limitations of CUSUM center around its ability to filter periodicity and noise. The domain of signal processing offers flexible and proven solutions for handling these effects.

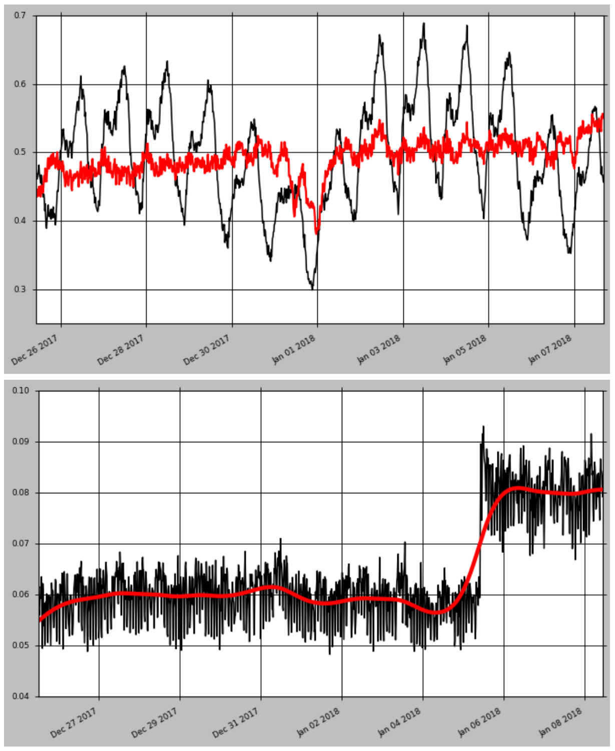


Figure 1: Time series operators help to mitigate noise and seasonality effects. The top figure shows a time series before and after seasonality is removed using Fourier regression. The bottom figure shows a noisy time series that is smoothed using a Butterworth lowpass filter. The lowpass filter preserves the general shape of the series while smoothing out spikey noise which can trigger false alerts.

Specifically, we use two classic signal processing techniques—the Fourier regression and lowpass filter—to preprocess the time series and reduce the prevalence of false alerts.

2.2.1 Fourier Regression. The Fourier transform is the most widespread and well-understood method for contextualizing periodicity in time series. Fourier regression [11] is a closely related concept consisting of a generalized linear model with Fourier terms as basis functions. For a time series Y_t, a simple Fourier regression model can be expressed as

$$Y_t = \beta_0 + \sum_{k=1}^{K} \Big(\beta_{2k-1}\sin(\omega_k t) + \beta_{2k}\cos(\omega_k t)\Big) + \epsilon_t$$

where $\beta_0, \beta_1, \ldots, \beta_K$ are coefficients to be estimated, $\omega_k = 2\pi k/M$ is the angular velocity or frequency, M is the cycle length (number of time periods per cycle), and ϵ_t is an error process. K sets the order of the model, a higher order includes more harmonics of the Fourier terms and results in a closer fit.

The above equation assumes a single value for M. In practice there may be multiple seasonality effects in a production environment. One way to check this is to examine the output of the FFT, and it is possible to extract significant cycle lengths through some automated process such as cross validation. Pragmatically however, it is reasonable to assume we will be encountering primarily, if not exclusively, diurnal and weekly seasonal cycles in our application. For time series with hourly granularity, incorporating this assumption will result in the final form

$$\begin{aligned}Y_t &= \beta_0 + \sum_{k=1}^{K} \left(\beta_{2k-1}\sin\left(\frac{2\pi kt}{24}\right) + \beta_{2k}\cos\left(\frac{2\pi kt}{24}\right)\right) \\ &+ \sum_{k=K+1}^{K+N} \left(\beta_{2k-1}\sin\left(\frac{2\pi k't}{7\times 24}\right) + \beta_{2k}\cos\left(\frac{2\pi k't}{7\times 24}\right)\right) + \epsilon_t\end{aligned}$$

where we define $k' = k - K$. We select the model order (K, N) by fitting multiple models and choosing the one with the lowest corrected AIC score [12].

Effectively the above Fourier regression assumes that Y_t can be reliably modelled by the sum of a collection of relatively high-order sine waves. If there is a trend in the underlying data, however, this assumption will lead to a poor fit. We estimate the trend by calculating daily medians for Y_t, and smoothing them using LOESS. We remove the trend by division, fit and subtract the Fourier regression from the detrended data, and add the trend back in through multiplication. In summary,

$$\begin{aligned}D_t &= \text{median}([Y_{M(t-1)+1}, \ldots, Y_{Mt}]) \\ \tilde{Y}_t &= \frac{Y_t}{\tilde{D}_{\lfloor 1+t/M \rfloor}} \\ Y'_t &= \tilde{D}_{\lfloor 1+t/M \rfloor} \times \left(\tilde{Y}_t - f(\tilde{Y}_t)\right)\end{aligned}$$

where $\tilde{D}_t$ is the LOESS-smoothed D_t, and $f(\tilde{Y}_t)$ is the Fourier regression fitted onto the detrended time series. We apply CUSUM on Y'_t to check for a change point.

We can fit the regression through ordinary least squares if the error process ϵ_t is approximately normal. Even in the case of time series with isolated, mild point anomalies, OLS tends to provide a reasonable fit. OLS is less resilient in the presence of pattern anomalies, and will overcompensate in cases where a daily peak falls unusually above or below the baseline. Since this is occassionally observed in our dataset, we opt instead to fit the regression through iteratively reweighted least squares, with a Huber loss function which is more resilient to outliers.

2.2.2 Butterworth Filter. While Fourier regression will let us estimate and remove seasonal effects, it does not help with removing noise, including isolated spikes which can lead to false positives. Signal filtering methods, in particular low pass filters, are designed for this purpose. A Fourier transform maps a series from the time domain (t, Y_t) to the frequency domain $(\omega, A(\omega))$. Once in the frequency domain, a lowpass filter can suppress higher-order frequencies that are likely contributors to noise.

A lowpass filter is characterized by a frequency response function $H(\omega)$. The filter is applied by taking $\tilde{A}(\omega) = H(\omega)A(\omega)$ over the frequency domain, and the filtered signal is returned to the time domain by taking an inverse Fourier transform on $\tilde{A}(\omega)$. From a range of options for lowpass filters, one of the most commonly used is the Butterworth filter [6], characterized by the frequency response function

$$H(\omega) = \frac{1}{1 + \left(\frac{\omega}{\omega_c}\right)^{2n}}$$

where n is called the order of the model and ω_c is the cutoff frequency. Butterworth filters are popular because they can be shown to be maximally flat in the passband. Intuitively, this means there are no ripples in the filtered output signal.

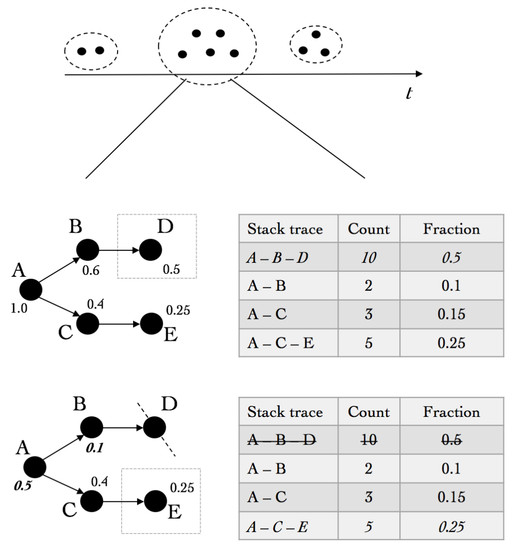
Frequencies lower than ω_c are passed through with minimal attenuation, whereas frequencies higher than ω_c are penalized at an exponential rate. n determines the rate at which $H(\omega)$ tapers off to 0. A higher n will decrease the variance of the resulting output signal. For regression detection, there are competing interests to consider when setting n and ω_c. Whereas we do want to filter noise, it is also important to avoid attenuating too aggressively so as to diminsh the magnitude of an actual changepoint, which could translate into a false negative. This resulted in our conservative default settings of $n = 1$ and ω_c equivalent to one day.

2.3 Deduplication

Functions in a codebase are heavily interdependent, so a regression in one function can trigger a cascading effect where level shifts also register on upstream and downstream dependencies. Surfacing these apparent change points adds little value to the regression investigation and remediation process. Clustering change points from similar functions together and attributing them to a single or small set of functions is helpful to reduce the volume of manual diagnostic work and address regressions at a faster cadence. We call this step deduplication as it pares down detected change points that provide redundant signal. The approach is designed around stack trace samples obtained from Xenon, Facebook's open-sourced HHVM profiling tool.

Change points can be in proximity to each other in two principal senses, in function-level dependencies and in time. The first stage of deduplication consists of bucketing proximate change points. In the current implementation, this is defined as change points that happen within 10 minutes of each other and that occur in functions that are included in a common stack trace. Within each bucket, we construct the dependency graph from the full collection of stack traces and label each node with the fraction of stack traces that include a call to the function.

Nodes are iteratively selected for removal based on fraction of exclusive stack traces in a greedy fashion, the stack traces that contain the removed node are likewise removed, and the change point magnitudes of the remaining nodes are revised down according to their membership in the removed stack traces. Nodes and stack traces are removed iteratively until the fraction of remaining stack traces falls below a tolerance threshold. All the nodes and stack traces are then reintroduced, and the procedure is repeated using a different starting point. After all potential starting points have been tried, we are left with a list of potential regression sets. The final regression set is selected based on a weighting taking into account the relative magnitude of the change points and the number of stack traces represented by the candidate regressions belonging to the set. The weighting is designed with the aim to strike a balance between avoiding attributing regressions to high level function calls (such as init) as well as very low level functions which might get called



Stack trace	Count	Fraction
A – B – D	*10*	*0.5*
A – B	2	0.1
A – C	3	0.15
A – C – E	5	0.25

Stack trace	Count	Fraction
~~A – B – D~~	~~10~~	~~0.5~~
A – B	2	0.1
A – C	3	0.15
A – C – E	*5*	*0.25*

Figure 2: Schematic overview of the deduplication step. Change points that are close in time and can be connected via a common dependency structure are grouped together. Nodes with the largest fraction of exclusive stack traces are iteratively removed in a greedy fashion, the stack traces they belong to are also removed, and the magnitudes of the remaining nodes are revised down based on the stack traces that were removed. The procedure is repeated until the fraction of remaining stack traces falls below a tolerance threshold.

more frequently as the result of a regressed parent function (like accessing a database). A conceptual illustration of the deduplication step is provided in Figure 2.

2.4 Use Cases

In this section we present some specific use cases to illustrate how the methodology described above is deployed in practice.

2.4.1 Efficiency Infra. The Efficiency Infra team monitors global CPU share (gCPU) for functions and endpoints serving the desktop site and client apps. Their primary challenge in detecting regressions is noise, which can come in the form of one-off spikes or temporary excursions that return to the baseline without intervention. Efficiency Infra also places an emphasis on detection speed, so keeping the their data ingestion footprint as small as possible is a notable objective.

Input series are smoothed using a Butterworth filter. Based on ad hoc experimentation, 3 days was selected as a minimum threshold to establish a reasonable baseline and obtain enough signal to avoid overfitting. The workflow runs several times a day, and there is no practical benefit of surfacing the same regression in mutliple workflow runs. In addition, applying CUSUM on the filtered series can distort the location of the change point, which can undermine the deduplication step and complicate any followup triage efforts. For both of these reasons, we actually run CUSUM twice: once on the filtered signal over the 3 day window, and if the CUSUM statistic clears a threshold we run CUSUM on the original signal over the last 4 hours. The second CUSUM run is primarily for more accurate reporting of the change point's time and magnitude, though we do add a second conservative CUSUM threshold in case the magnitude of the original change point is an artifact of a mild short-term trend. The overall procedure is summarized in Figure 3.

An important corner case for Efficiency Infra is when gCPU drops below the baseline and steadily creeps back up, which could indicate a remediation effort that turned out to be unsuccessful. To catch these cases, we first check for large change points in the negative direction during the first pass of CUSUM. If such a shift is detected, we truncate the historical data prior to that point, rerun CUSUM, and proceed with the rest of the analysis as described above.

2.4.2 Core Data Demand. The Core Data Demand team monitors the volume of queries (QPS) that come into Facebook's distributed data store for the social graph (TAO) [5]. Similar to the Efficiency Infra team, they analyze the global share of TAO queries for functions and endpoints (gTAO). Due to the cylical patterns of Facebook user activity which drives data demand, a large portion of the codebase exhibits strong periodicity in its TAO QPS volume.

We start by computing the Fourier transform of each time series. For time series with high amplitudes at daily and weekly frequencies, we fit a Fourier regression on two weeks of data and remove the seasonality as described in Section 2.2.1. Even after controlling for seasonality many Core Data Demand time series exhibit heteroscedasticity, so we uniformally apply a log transform to stabilize the variance of the time series. The CUSUM algorithm was run over the last 5 days of the final transformed time series. This time window was selected by analyzing the tradeoff between detection accuacy and processing time across past incidents.

2.4.3 Instagram. The Instagram Efficiency and Reliability team is responsible for monitoring the efficiency of Instagram's infrastructure. Similar to Efficiency Infra and Core Data Demand, the team monitors gCPU and TAO QPS for the Instagram codebase. The tracking system is monitoring gCPU currently, with TAO QPS to be added in the future.

Similar to Core Data Demand, the time series we monitor for Instagram suffer from strong periodicity. The system setup is correspondingly similar albeit with two notable differences. First, we did not apply a log transform since it did not show an advantage in terms of precision and recall. Second, we run the CUSUM algorithm on a shorter window of 3 days as this struck a reasonable comprise between false positives and detection speed.

3 MODEL TUNING AND EVALUATION

A significant factor to making the signal from this system actionable in practice is effective calibration on well-defined performance metrics. In this section, we define the metrics used to evaluate the detection system, the process for initial system calibration, and evaluating the comparative gains of subsequent modeling revisions following the initial deployment.

3.1 Metrics

We identified three relevant performance metrics for our system:

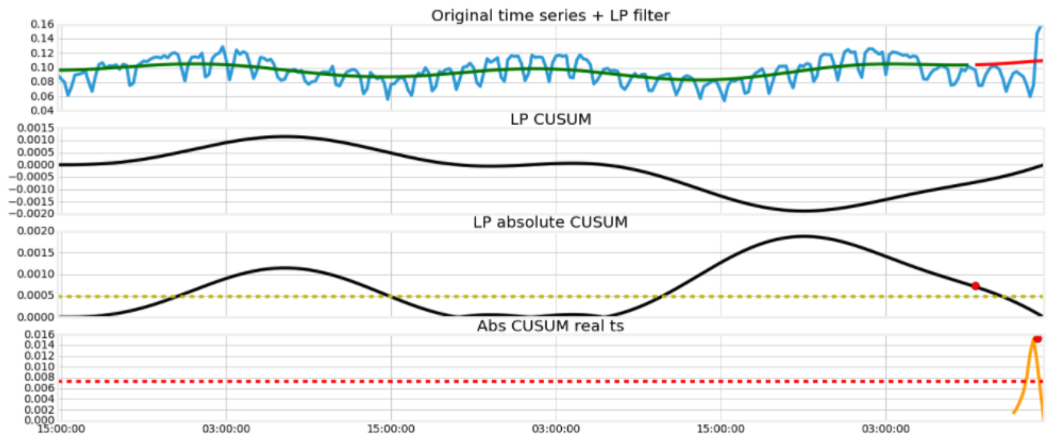


Figure 3: An illustration of the main steps of the regression detection algorithm as implemented for Efficiency Infra. The original time series is passed through a Butterworth lowpass filter (top panel). The CUSUM statistic is calculated on the filtered signal, and we check whether it crosses a threshold in the last 4 hours (middel panels). If the CUSUM statistic does cross the threshold, we run CUSUM on the original signal over the 4 hour window (bottom panel). If this second CUSUM statistic clears a tolerance threshold, the magnitude and location of the change point are logged. After all the change points are logged, they are passed into the deduplication step.

- **Precision:** What fraction of the alerts we create are actually efficiency regressions?
- **Recall:** What fraction of the real efficiency regressions do we detect?
- **Detection Speed:** How quickly do we catch a regression after it happens?

We focus on the first two metrics for a couple of reasons. Due to the number of the time series we monitor, marginal improvements to precision have a significant impact on the number of alerts we create and are reviewed manually by an engineer. On the other hand, small undetected regressions as a byproduct of lower recall would, in the aggregate, lead to a bloated codebase translating into greater resource requirements to host new users and services as well as a poor user experience overall. Having a detection framework that is high in both precision and recall is therefore highly imperative. The third metric above, while also of interest, introduces a further labeling challenge in that the exact starting point of a regression is not always precisely defined and can be open to interpretation. Some regressions do not manifest themselves as discrete jumps and instead accumulate over a short time frame, introducing a layer of ambiguity to the measurement of detection speed which is not straightforward to overcome.

3.2 Initial Calibration on a Static Training Set

Initial system setup involves determining which time series operators to use, the training window, and tuning various meta-parameters (e.g. threshold for the CUSUM statistic). This initial setup relies heavily on analyzing an assorted set of examples from prior observation. For each individual use case, we collected several snapshots of time series both with and without efficiency regressions as identified by engineers from the corresponding teams. The examples with their associated labels and annotations were uploaded to a MySQL database through a custom internal web based UI called *Time Series Keeper*, designed specifically to assemble a standardized canonical training set.

We load the examples into a Jupyter notebook and experiment through several configurations based on the series' observed attributes. During this stage, we construct Receiver Operating Characteristics (ROC) curves for each candidate setup. Similar to binary classification models, the area under the ROC curve (AUC) enables us to easily rank order different candidate setups. We identify the system configuration that maximizes AUC and deploy it into production. In addition to helping us choose the best methodology, ROC curves also help us visualize the tradeoff between accuracy and recall. We analyze this tradeoff to determine the target accuracy and recall which helps us determine the CUSUM test statistic threshold we should use in production.

3.3 Performance Monitoring Feedback Loop

The initial calibration detailed in Section 3.2 uses a hand-curated set of examples. Collecting examples in this way is limited in scale and liable to not producing a fully representative sample of all cases that would arise in production. A more reliable indication of how well the system is performing can be obtained once the detection system is online and starts producing alerts for engineers to review. The alerting system exposes functionality for the engineers to provide feedback on whether the incident represents an actual regression or various potential classes of false positives. Collecting this feedback over time results in a labeled dataset from the system output that can be leveraged to understand the performance of the methodology in practice. Using this dataset, we identify systematic patterns that generate false positives which motivate follow-up analyses on avenues to improve the methodology.

While this feedback loop can provide guidance on opportunities to improve detection precision, it is critical to assure recall is not negatively impacted in the process. This is especially true since the positive examples identified through the feedback loop are unlikely to capture all the regressions that actually transpired in production. In other words, we are limited by what we do not know, and any loss in recall we measure in the labeled set is more likely to be a lower bound on the actual figure. On the other hand, improved precision can create room to increase the sensitivity of the detection algorithm, allowing us to surface smaller regressions and improving the overall recall rate.

3.4 A/B Testing

As alluded to in Section 3.3, the samples produced through the feedback loop exhibit bias. While the feedback loop grants us some flexibility in assessing the impact of proposed revisions to the methodology or tuning parameters, it cannot indicate whether new regressions will be caught or additional false positives would be flagged as well. The only way to check against these questions is to run the existing methodology against the proposed changes on the same inputs and compare the outputs side by side.

The productionized detection system runs using Facebook's distributed computing platform *FBLearner Flow*[9]. Flow also enables functionality to define a workflow that runs the detection system in production in parallel to a proposed changeset, and apply custom aggregation procedures on the outputs of each. In this vein, we collect and compare the algorithm outputs applied over millions of time series spanning a week of runs, corresponding to a couple dozen push cycles, and create visualizations for all the change points detected by the production system but not by the proposed changeset, and vice versa. Inspecting the output and classifying them between regressions and false positives drives the discussion and informs the final determination as to whether a changeset improves detection overall, and if so the proposed changes are landed.

4 PERFORMANCE COMPARISON

To evaluate the performance of our system, we compare the output of our algorithm against a benchmark implementation of CUSUM based on the one found in the `changepoint` R package [13], as well as a commercial machine learning-based anomaly detection software called Anodot [3]. The main distinctions between the benchmark CUSUM implementation and the implementation described in Section 2 is a normalizing scaling factor and the inclusion of time series operators to control for noise and seasonality in the input time series. Anodot is effectively an ensemble model, the algorithm first performs a multiclass classification on the input time series and applies the modeling framework assessed to be most suitable for the given series profile. One of the convenient aspects of Anodot is it does not expose tuning parameters in the user-facing API, all parameter tuning is handled internally using proprietary techniques. Anodot does return a start and end time as well as a confidence score between 0 and 1 for the anomalies it detects, allowing for some filtering and latitude in modulating the sensitivity of the algorithm.

4.1 Data Collection and Testing Setup

The gCPU and gTAO metrics for functions and endpoints are derived from stack trace samples collected through Xenon, a profiling tool for HHVM. These metrics are logged and stored as time series in Facebook's internal distributed monitoring system called the Operational Data Store (ODS) [1, 4]. The series are queried from ODS using FBLearner Flow [9]. The time series transformations, detection logic, and deduplication steps are implemented in Python 3 and likewise applied in the same Flow workflow. Anodot, whose implementation is in Java, is exposed and called by Flow through an Apache Thrift API.

The main input parameters to the Flow workflow are the specification of an input window and scanning window. The input window is the full segment of the time series that is available to the algorithms for model fitting, whereas the scanning window is defined relative to the end of the input window and demarcates where the detection procedure should scan for regressions. For Efficiency Infra workflow runs, the input window is 3 days and the scanning window is the last 4 hours. For Core Data Demand workflow runs, the input window is 2 weeks and the scanning window is 5 days. Anodot and our customized CUSUM algorithm are both trained over the same input window. For Anodot, we automatically filter out any anomalies that have a start time or an end time that fall before the scanning window. The CUSUM benchmark does not have a formal modeling step, as such we only pass it the data in the scanning window since the rest of the input window is immaterial to the outcome. The post-detection actions we perform in production were applied evenly so as to not systematically favor any approach. These post-detection actions consist of running the deduplication step to distill the number of incidents into a small set of root causes, and discarding detected change points that fall below a fixed tolerance threshold.

As alluded to in previous sections, a challenge to comparing the performance of different algorithms in the present context is the absence of a labeled training set. We do have labeled time series collected through the feedback loop described in section 3.3, but these would be biased in favor of the current algorithm which would have 100% recall on those series by default. Instead, we bootstrap a test set by running the three algorithms with their default settings on the same input data and collecting the incidents they identify. We can then examine these incidents to establish whether they are regressions or false positives, and in turn produce precision and recall curves for each algorithm based on which of the test incidents it identifies.

4.2 Results

Regressions are relatively uncommon events. To include enough examples of regressions in a bootstrapped test set, it is necessary to run the detection frameworks across multiple code pushes. To this end, we collected the outputs for our algorithm, the CUSUM benchmark, and Anodot across 103 Efficiency Infra workflow runs corresponding to pushes spanning January 6–9, January 31–February 1, and February 4–7, 2018. For the Core Data Demand workflow, which has a substantially longer scanning window, we only run two instances to generate a similar number of incidents. These instances have scanning windows over January 14–18 and January 29–February 2, 2018, respectively.

The outputs of the three algorithms were aggregated and manually labeled. We consider an incident to represent a true positive

Use Case	Detection System	Incident Count		Classification		Precision
		Pre-Dedup	Post-Dedup	TP	FP	
Eff. Infra	Customized CUSUM	1,701	131	111	20	85%
	Benchmark CUSUM	29,956	352	39	92	30%
	Anodot	2,120	118	98	20	83%
Core Data	Customized CUSUM	1,793	61	53	8	87%
	Benchmark CUSUM	989	85	52	33	61%
	Anodot	8,789	42	34	8	81%

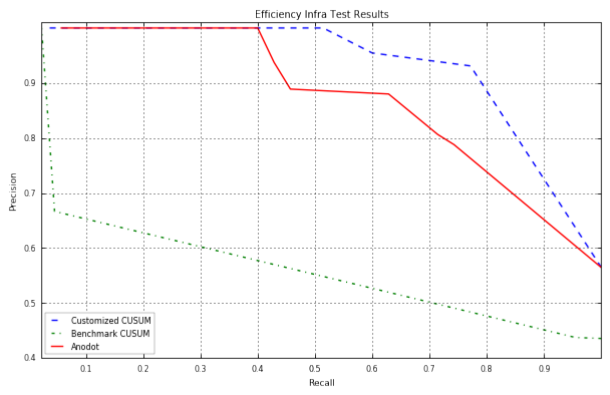
Table 1: Incidents detected by the customized version of CUSUM, benchmark CUSUM and Anodot under default settings. Deduplication reduces the number of incidents to be investigated by more than 90%. The accuracy of customized CUSUM and Anodot on root causes is comparable, with customized CUSUM producing a greater number of true positives.

if it contains an apparent increase in the scanning window that cannot be explained by periodic behaviors and also shows no indication of being followed by a proportional decrease. Incidents that happen close to the end of the scanning window and seem to coincide with the upward phase of a periodic cycle present somewhat of an ambiguous case. We classify these cases as true positive if the magnitude of the increase is more than 10% over the highest periodic peak that has been previously attained.

Table 1 shows the number of incidents, root causes, and number of true and false positives across the three algorithms with their default settings for the Efficiency Infra (gCPU) and Core Data Demand (gTAO) use cases. For customized CUSUM, the default setting refers to the CUSUM threshold that is currently running in the production version of the algorithm at Facebook. For Anodot, the default setting refers to the anomalies that Anodot flags out-of-the-box without any custom postprocessing logic besides filtering out anomalies outside the scanning window. For benchmark CUSUM, we set a relatively low threshold for gCPU which returned nearly 30,000 detected incidents. Since gTAO has a longer scanning window and therefore produces many more incidents per run, the default threshold for benchmark CUSUM was raised to bring the number of detected incidents within more interpretable levels.

For all the algorithms, the deduplication step compresses the output by more than 90%. The fact that Anodot produces many more raw incidents yet results in fewer root causes compared with customized CUSUM demonstrates the utility of deduplication. One potential explanation of this outcome is that many of the incidents Anodot detects are artifacts of cascading effects caused by a smaller set of individual regressions, whereas customized CUSUM on average flags incidents that are closer to their respective root cause. In all cases, the lower number of root causes represents a much more reasonable volume of time series for an engineer to investigate manually.

Under default settings, the customized CUSUM and Anodot algorithms show comparable accuracy overall. However, customized CUSUM returns more root causes, translating into a greater number of true positives. As mentioned in section 2.1, false negatives are considered worse than false positives for our application, since a pattern of missed regressions can lead to a slower and more resource-expensive codebase over the long run. Customized CUSUM compares favorably against Anodot for both gCPU and gTAO from this perspective. Benchmark CUSUM, on the other hand, has observably lower accuracy than either of the other two algorithms. Even after culling the root causes with the highest CUSUM statistic for the gCPU case, the accuracy of benchmark CUSUM does not go above 30%. Benchmark CUSUM performs somewhat better in the gTAO case, possibly due to the fact that the Core Data Demand series are evaluated over a longer scanning window—reducing the likelihood that false positives are detected close to the end of the scanning window as explained in section 2.1—and tend to exhibit less noise overall.



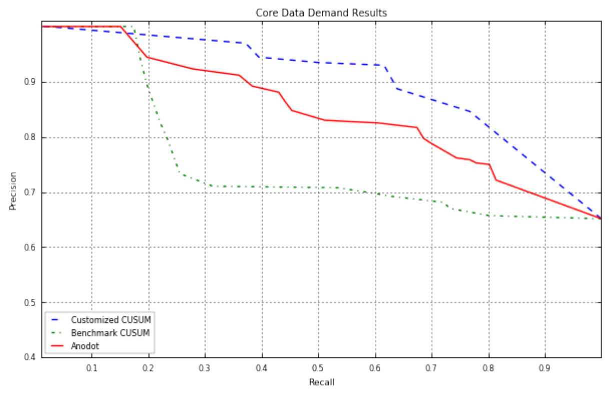


Figure 4: ROC curves comparing the detection quality of CUSUM, CUSUM with time series operators and normalization, and Anodot. Top figure is based on examples from the Efficiency Infra's detection system. The bottom figure is based on examples from Core Data Demand.

For Anodot and the CUSUM algorithms, the degree of certainty for detected incidents is contextualized by a confidence score and CUSUM test statistic, respectively. By aggregating the labeled root causes, obtaining confidence scores and CUSUM statistics for each of them, and performing a sweep on a threshold applied to these metrics, we generate the precision-recall plots for the three methodologies illustrated in figure 4. In both the gCPU and gTAO cases, the customized CUSUM implementation is shown to dominate over the other two methodologies. This implies that for a target level of precision, we can achieve higher recall with customized CUSUM compared to the other two algorithms under consideration.

5 CONCLUSION

In this paper, we described the real-time system we designed and implemented to detect efficiency regressions in an enterprise codebase. The algorithm is based on a non-parametric statistical rule-based method to detect changepoints (CUSUM). By supplementing this method with signal processing techniques, we are able to achieve detection rates that are high in accuracy and recall. We described the processes we used to tune an initial parametrization and test proposed changes to the methodology. Finally, we demonstrated that our solution outperforms a simple benchmark and a state-of-the-art commercial machine learning solution. This system is currently in production scanning millions of time series multiple times a day, and serves as the main line of defense against efficiency regressions at Facebook.

We are scaling the detection framework to monitor other operational metrics and continue to test ideas to further improve its detection accuracy. Since the original submission, we have extended the platform to monitor efficiency metrics for several other teams at Facebook. In addition, we are working on shortening the onboarding cycle for new use cases by instrumenting automated tuning. By maintaining an operationally efficient codebase, Facebook is better able to continue scaling the site to include new users and services.

ACKNOWLEDGMENTS

Eloi Morlaas assisted with visualizations and improved the algorithm's detection accuracy since the initial submission of this paper. Emil Hesslow and Jiro Sumitomo were early contributors and implemented significant pieces of this work. The system improved through many iterations thanks to feedback and data collection efforts from several engineers including Leo Jefferson Cardoso, Joy Yao, Alison Ding, Grace Law, Garik Sargsyan, Qi Zhou, Guilin Chen, Manish Khetty, and others.

REFERENCES

[1] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. 2012. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.* 35, 2 (2012), 4–13.

[2] Samaneh Aminikhanghahi and Diane J. Cook. 2017. A survey of methods for time series change point detection. *Knowledge and Information Systems* 51, 2 (2017), 339–367.

[3] Anodot. 2017. *Ultimate Guide to Building a Machine Learning Anomaly Detection System.* Technical Report.

[4] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. 2011. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11).* ACM, New York, NY, USA, 1071–1080. https://doi.org/10.1145/1989323.1989438

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13).* USENIX, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[6] Stephen Butterworth. 1930. On the theory of filter amplifiers. *Wireless Engineer* 7, 6 (1930), 536–541.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[8] D. Choudhary, A. Kejariwal, and F. Orsini. 2017. On the Runtime-Efficacy Trade-off of Anomaly Detection Techniques for Real-Time Streaming Data. *ArXiv e-prints* (Oct. 2017). arXiv:stat.ML/1710.04735

[9] Jeffrey Dunn. 2016. Introducing FBLearner Flow: Facebook's AI backbone. Facebook Engineering Blog. https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/.

[10] S. Fremdt. 2013. Page's Sequential Procedure for Change-Point Detection in Time Series Regression. *ArXiv e-prints* (Aug. 2013). arXiv:stat.ME/1308.1237

[11] Andrew C. Harvey and Neil Shephard. 1993. Structural time series models. In *Econometrics*, H.D. Vinod G.S. Maddala, C.R. Rao (Ed.). Handbook of Statistics, Vol. 11. Elsevier, 261 – 302. https://doi.org/10.1016/S0169-7161(05)80045-8

[12] Clifford M. Hurvich and Chih-Ling Tsai. 1989. Regression and time series model selection in small samples. *Biometrika* 76, 2 (1989), 297–307. https://doi.org/10.1093/biomet/76.2.297

[13] Rebecca Killick and Idris A. Eckley. 2014. changepoint: An R Package for Changepoint Analysis. *Journal of Statistical Software* 58, 3 (June 2014).

[14] V. Konev and S. Vorobeychikov. 2017. Quickest Detection of Parameter Changes in Stochastic Regression: Nonparametric CUSUM. *IEEE Transactions on Information Theory* 63, 9 (Sept 2017), 5588–5602. https://doi.org/10.1109/TIT.2017.2673825

[15] Nikolay Laptev, Jason Yosinski, Li E. Li, and Slawek Smyl. 2017. Time-series extreme event forecasting with neural networks at uber. In *Int. Conf. on Machine Learning Time Series Workshop,*.

[16] E. S. Page. 1954. Continuous Inspection Schemes. *Biometrika* 41, 1/2 (1954), 100–115. http://www.jstor.org/stable/2333009

[17] Facebook Investor Relations. 2017. Facebook Reports Second Quarter 2017 Results. Press Release. https://investor.fb.com/investor-events/event-details/2017/Facebook-Q2-2017-Earnings/default.aspx.

[18] Chuck Rossi. 2017. Rapid release at massive scale. Facebook Engineering Blog. https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/.

[19] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. 2016. Continuous Deployment of Mobile Software at Facebook (Showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/2950290.2994157

[20] K. G. Shin and P. Ramanathan. 1994. Real-time computing: a new discipline of computer science and engineering. *Proc. IEEE* 82, 1 (Jan 1994), 6–24. https://doi.org/10.1109/5.259423

[21] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10).* ACM, New York, NY, USA, 1013–1020. https://doi.org/10.1145/1807167.1807278

[22] Gabriel Tsechpenakis, Dimitris N. Metaxas, Carol Neidle, and Olympia Hadjiliadis. 2006. Robust online change-point detection in video sequences. In *In 2nd IEEE Workshop on Vision for Human Computer Interaction (V4HCI), in conjunction with the IEEE Conference on Computer Vision and Pattern Recognition.*

[23] Kenji Yamanishi and Jun-ichi Takeuchi. 2002. A Unifying Framework for Detecting Outliers and Change Points from Non-stationary Time Series Data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02).* ACM, New York, NY, USA, 676–681. https://doi.org/10.1145/775047.775148

[24] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw GPS data for geographic applications on the Web. , 247-256 pages.