Real-time Failure Prediction in Online Services

Mohammed Shatnawi
Microsoft, Redmond, WA, and
Simon Fraser University, British Columbia, Canada

Mohamed Hefeeda
Qatar Computing Research Institute
Doha, Qatar

***Abstract*—Current data mining techniques used to create failure predictors for online services require massive amounts of data to build, train, and test the predictors. These operations are tedious, time consuming, and are not done in real-time. Also, the accuracy of the resulting predictor is highly compromised by changes that affect the environment and working conditions of the predictor. We propose a new approach to creating a dynamic failure predictor for online services in real-time and keeping its accuracy high during the services run-time changes. We use synthetic transactions during the run-time lifecycle to generate current data about the service. This data is used in its ephemeral state to build, train, test, and maintain an up-to-date failure predictor. We implemented the proposed approach in a large-scale online ad service that processes billions of requests each month in six data centers distributed in three continents. We show that the proposed predictor is able to maintain failure prediction accuracy as high as 86% during online service changes, whereas the accuracy of the state-of-the-art predictors may drop to less than 10%.**

I. INTRODUCTION

Online services have Service Level Agreements (SLAs) covering various aspects of the service such as reliability, response times, and up-times. For example, Amazon has a stated up-time of 99.95% SLA, and 3Tera has a 99.999% availability SLA. The cost of not meeting these SLAs is not only low customer satisfaction, but also a heavy price tag due to fines and loss of business. It is estimated that the annual downtime cost of IT systems in North America is about $26.5 Billion [1].

A failure is an observed deviation by the user of the service from the expected behavior. The user can be human or another computer system. Failure management, a term we use to refer to all aspects of dealing with failures, plays a key role in the reliability of online services. It includes failure monitoring, prediction, detection, root-cause analysis, all the way to failure handling by prevention and/or hiding. We focus in this paper on failure prediction. If failures are predicted correctly and in time, corrective measures can be taken to prevent or hide them from being observed by users.

There are several approaches to create an online service failure predictor. These include statistical methods like Bayesian, decision rules methods like decision-trees, artificial intelligence methods like neural-networks, and cluster analysis methods like clustering algorithms [2], [3]. The predictor generation entails: (1) generation of data about the system; including its inputs, working environment, and outputs, (2) collection of this data into containers like log files to be used later, (3) pre-processing and analysis of the data to exclude extraneous data, and organize the remaining impactful data into usable data models like dimensional models [4], (4) designing of prediction algorithm(s) and system(s), (5) training the predictor, (6) testing it, and (7) deploying it in production to be used in failure prediction in the online services. These steps are lengthy, complex, and time consuming [5], [6]. Thus, they usually take place before the run-time lifecycle of the online service. Also, the resulting predictor is built for certain system configurations and working conditions like the amount of available resources and their performance characteristics, e.g., the number of routers and their throughput, and the number of database systems and their sizes. If these configurations change, the predictor may no longer be accurate. For example, the predictor could be designed for a database system that meets its throughput SLA when it encounters up to N concurrent requests per second, and it fails beyond that. If the system administrator adds a performance enhancing database cluster to the database system where now it can handle up to $3 \times N$ requests per second and still meets its throughput SLA requirements, the predictor would likely continue to predict failure if the requests per second approaches N not $3 \times N$. In this paper, we call these predictors **static predictors**, because they do not adapt to changes in the service functionality, the system resources, or other changes.

Furthermore, data mining techniques used to generate failure predictors require high volumes of data to reach high prediction accuracy [5], [7], [3]. Current data mining techniques de-emphasize the system under study, by treating it as a black box, and focus on its inputs, outputs, and working conditions to build the failure predictor models. These characteristics of current data mining techniques make them not suitable for real-time creation and updates. Static predictors work well within environments that do not change often; such as transportation systems like airplanes and navy ships, engineering systems like factories and assembly lines, and software systems like games. On the other hand, modern online services lack such stability over time at many levels including functionality, designs and implementations, and service hardware provisions to accommodate the changing user requirements and loads over time. The ever-changing landscape of online services, coupled with requirements such as continuous up-times, make the use of static failure predictors challenging and less efficient.

We investigate a new approach to failure prediction in online-services during their real-time lifecycle that overcomes the problems noted above. Real-time refers to the runtime lifecycle of the service, where the service is in production and is being used by real customers. We use synthetic transactions during the service real-time lifecycle to generate current data about the service. This data is used in its ephemeral state to build, train, test, and maintain an up-to-date failure predictor. We evaluate the effectiveness of the proposed approach on a large-scale enterprise backend ad service. The service handles over 4 billion ad requests a month. We show that during the production phase where the service goes through changes, our approach is able to maintain high prediction accuracy of about 86%, whereas the prediction accuracy of current state-of-the-art predictors may drop to less than 10%. The recall of the data generated by our synthetic transactions is 100%. Recall in this context refers to the percentage of the generated data that is relevant and used. In contrast, the recall in the production logs is less than 2%. In addition, we show that we can update the predictor in real-time in less than 7 minutes; this includes generating data, creating the predictor, training it, and testing it. On the other hand, building a failure predictor using typical data mining techniques for the same service by using production logs requires about 5 weeks of production running and logging, and it takes more than 17 hours of pre-processing, training and testing.

The contributions of this work are (1) a novel approach to build real-time failure predictors, (2) a light-weight data mining algorithm for failure predictors in online services, and (3) the actual implementation and deployment of the proposed approach in a real online service environment.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents our approach. Section 4 describes the evaluation of the proposed approach. Section 5 has the conclusions and suggested future work.

II. Related Work

We summarize the current approaches for creating failure predictors based on production logs. Then we discuss the need to have failure prediction in online services, and the key efforts done there.

A. Handling Production Logs

Quite a few research efforts emphasize the problems encountered in building accurate data mining models based on production logs. First, logs are complex and hard to mine [5]. Second, data in the logs may not be sufficient for data mining [5], [7]. Third, pre-processing the logs to get them to a state where they are usable in prediction models is tedious and expensive [5], [8]. Snyder et al. [7] argue that the insufficiency of log data causes problems for mining them because data in the logs are extraneous, and it is hard to identify the relevant pieces that are needed in the data mining process. Xu et al. [5] describe the voluminous nature of data in production logs, and show that logs are not actually helpful in many cases, because of their large volume. Chen et al. [9] study the application of machine learning to logs of faulty executions to predict the root cause of failures. Their Pinpoint model requests paths in the system to cluster performance behaviors, and identify root causes of failures and anomalous performance. Pre-processing of system logs to prepare them for analysis and mining is studied by Salfner et al. [10].

The research threads above describe the problems with production logs that make them hard to work with, especially in real-time. They aim at alleviating some of the problems and symptoms, but come short of reducing the cost of processing production logs to levels that are suitable for real-time analysis and mining. These limitations show the need to produce a real-time predictor that does not depend on production logs.

B. Online Service Failure Prediction

Data mining and machine learning techniques used in creating failure predictors require data to be in some structure [2], [3]. Many efforts have focused on enhancements of production logs by adding structure through the use of meaningful logging [11]. Zheng et al. [8] suggest event categorization and filtering of logs to overcome their lack of structure and lack of usability for data mining. Xu et al. [5] attempt to identify problems with production logs of distributed systems, and suggest methodologies to enhance the performance of mining the logs by automatic matching of log statements. Cohen et al. [12] describe how failure prediction models are built to identify and study the root-causes of failures. They propose techniques to categorize the faulty execution results found in the logs, before building failure prediction models based on them. They also propose the use of indexing and clustering of system histories to correlate with failures. Leners et al. [13] propose an algorithm to improve availability in distributed systems by using failure informers. The failure informer is a reporting service that is built based on mining and analyzing the system messages in the logs.

In addition to root-cause analysis, failure predictors aim at analyzing the execution path structures that lead to failures. This is done by using instrumentation data from online servers to correlate bad performance and resource usage. Sambasivan et al. [14] implement path tree comparisons as means of predicting the paths that lead to failures. Realizing that failure is the norm in massive online service infrastructures, Hystrix of Netflix [15] aims at isolating points of access to remote systems, services and 3rd party libraries, to stop the cascading of failures. This adds resilience to services in distributed systems. Hystrix uses real-time monitoring, and acts on failures after they are detected. A few efforts attempt to build predictive models based on execution analysis by replaying the debugging information in logs [11], [16], [10]. Li et al. [17] propose WebProphet and the use of parental dependency graph to encapsulate web object dependencies to implement webpage load predictions. Viswanathan et al. [18] develop semantic framework for data analysis to enhance the performance of networked systems, and logging is the mechanism of collecting data about the system.

We note that all efforts related to our work of failure prediction depend on some form of logging for later analysis, and act on failures after they happen. Considerable effort is put to enhance the performance of mining the logs. Our approach has a key difference from existing approaches to enhancing online service reliability. We use data in real-time, because the cost of working with logs is too high for real-time processing.

III. Proposed Approach

Predictors are designed to predict the outcome of an event [2], [3]. In online services, events represent a variety of measurable aspects and characteristics of the service, such as the response time of the service, the availability of the service, and the number of packets routed correctly in a given time. The dynamic data prediction approach we propose is usable with any of these events. For the rest of the paper, we use the term **failure prediction** to indicate predicting when the outcome of an event does not meet its SLA. For example, if the event of interest is response time, then failure prediction means predicting the cases where the operation under study does not finish within the SLA.

We define two concepts: **Local System** and **Scenario**. Local System refers to the stack of software, hardware, and operating system used to perform a specific functionality in the online service. Scenario refers to the collaboration of the set of local systems that are used to implement an end-to-end (e2e) user scenario. As an example, assume the online service of interest is a retail service, where customers buy sports products. An example of e2e scenario is the process of finalizing the online purchase through a checkout process. Assume the event of interest is response time. The response time SLA for the e2e transaction could be 300ms; the purchase process meets its SLA if it completes in less than 300ms, and fails otherwise. Assume that the checkout process makes the following calls behind the scene: check cart contents, check product availability, calculate total price, validate credit card, submit the transaction, return to user, and update all related systems such as inventory, credit cards, and logging systems. The collaboration of these local systems to implement the purchase process represents the checkout scenario.

A. Testing in Production

Before we delve into the details of our approach, we describe Testing in production (TiP). TiP is a set of software testing methodologies that utilizes real production environments in a way that leverages the diversity of production, while mitigating risks to end users [19], [20], [21]. Enterprise service providers such as Facebook, Google, Microsoft and Yahoo use TiP to perform functional, stress and performance, as well as A/B testing in real-time [21]. The synthetic transactions used in TiP generate loads that are marked with special moniker(s) so that they are distinguished from real transactions, and do not interfere with the service destructively, or result in an incorrect state of the system like product inventory reduction due to test purchases. Synthetic transactions in TiP do utilize the service resources, and this puts an impact on the service. Service designers account for such an impact due to the importance of TiP; without it the service is flying blind [19], [21]. We utilize TiP principles and infrastructures as the platform for our suggested approach. No production code is instrumented

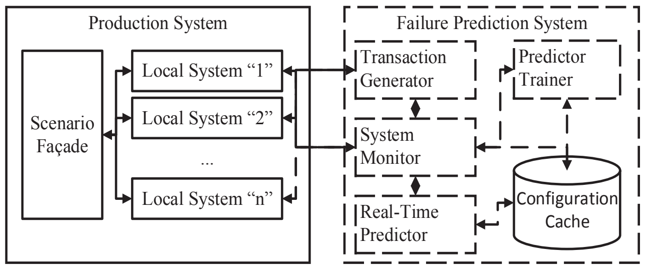

Fig. 1. Interaction between proposed failure predictor and production system.

to generate the data. Data is generated through TiP synthetic transactions. This is an advantage of our approach, because we do not impact the production code, and thus no extra testing is needed. Also, updates to the failure predictor do not require production code update or redeployment. We use TiP to cover functional, performance, and data failure modes.

B. Overview of the Proposed Dynamic Failure Predictor

Our proposed dynamic failure predictor is suitable for the dynamic nature of online services' functionality, environments, and elasticity of their loads over time. It can be used for systems with static environments as well, but it is superior in systems with dynamic situations. The proposed approach can be summarized in the following steps:

- Step 1: Using synthetic transactions, local synthetic transactions (LSTs) and scenario synthetic transactions (SSTs), to execute the local systems of the online service in a way that mimics user behavior, and constitute a complete e2e scenario such as a user buying a product online.
- Step 2: Collecting the synthetic transactions inputs, local systems outputs, e2e scenario outputs, and events of interest. This data is collected into in-memory constructs with a predefined dimensional model [4].
- Step 3: Using the freshly collected data from running the synthetic transactions to correlate the local systems inputs, local systems outputs, the output of the e2e scenario, and the event of interest.
- Step 4: Building a real-time predictor from the collected data and the correlations found in the previous steps, and training and testing it in real-time.

The failure predictor is used in production as long as it has high prediction accuracy. We can measure its prediction accuracy since we know the output of the e2e scenario and the output of the event of interest from running the synthetic transactions as well as real transactions in real-time. If the accuracy of the predictor drops below a threshold for a given period of time, we rebuild a new predictor, train it, and test it.

Figure 1 depicts the relationship between a production system and the failure prediction system used to generate and maintain the predictor. The production system is comprised of multiple local systems that constitute one e2e scenario. The failure prediction system is an independent product that runs in the same environment where the online service runs. It has access to the same resources as the production service, but is not part of the online service code. Updates and re-deployments of the predictor code can be done anytime without interfering with the service production system.

C. Generating Real-time Synthetic Transactions

The transaction generator, in the failure prediction system in Figure 1, mimics the user scenario by making all the local system calls that comprise the scenario. The system monitor collects the responses of each LST, the event of interest, and other information about each local system such as the number of running tasks, and the used resources in the system.

The algorithm we propose to generate synthetic transactions is as follows. The transaction generator makes LST calls using test loads, which are similar to real loads. The real user behavior (loads and call distributions) are found from the logs of previous deployments of the service, or from field/market studies about the service. For example, it would be known based on what is found in previous production logs, or estimated from market research, that the average customer of a retail online store buys 5-10 items a time, and that the service gets around 500 concurrent users at peak times. So if local system 1 is an addProductToCart() function, the average expected products to add are 7, and the average expected concurrent function calls are 100. Then the LSTs made by the predictor transaction generator start with these loads, and progressively add more loads until the failure causing loads are found. The test loads are executed on the current system, and so generate current information that represents the current state of the system.

The LST calls are made at equidistant time series; once every N seconds. The value of N is configurable, depending on the service, and it varies during execution time, increases or decreases, depending on the state of the service. If the service is churning and failures are happening more, then N is reduced to get a better pulse of the system. The event is captured after each of these tests, and its value is compared to the result of the prediction. If the accuracy of the predictor starts to drop, the tests are done at a higher rate to generate data to build a new predictor. The set of tests performed every N seconds need not be exactly like those of customer behavior. However, if the service at hand requires an exact replica of the user behavior, then parts of previous production logs representing that behavior can be replayed as suggested by [11], [16], [10].

D. Collecting the System Data

The system monitor collects the data generated by running the LSTs around the failure points found by the transaction generator. The data is passed to the predictor configuration cache, as shown in Figure 1. The data includes the current load and state of the system like the number of tasks, and the used resources like CPU and memory utilization. The data is collected and hosted in memory in an array with a dimensional model schema [4]. Dimensional modeling refers to a set of concepts and techniques used in data analysis which provide insights into the cause-effect relationships between entities. Data is organized into two sets, a set of measured or monitored data, called **facts**, and a set of parameters, called **dimensions** that define, impact, and control the facts.

For this work, the fact is the event of interest that is to be predicted, such as response time. The dimensions are the other sets of data that impact the event, such as:

- The **LST** that is called: this allows us to know which test is run.
- The **time** that LST is called: the time stamp is what allows us to relate and study the cause and effect in the system calls, i.e., at this time there are this many function calls and this many tasks, which caused this response time.
- The **number of tasks**: processes or jobs in the system.
- The **resource (CPU, compute servers, and memory) utilization**: although this may be seen as a measure, it plays the role of a dimension that impacts the event.

Dimensions					Fact
Dim1 LST	Dim2 Tasks	Dim3 CPU	Dim4 Memory	Dim5 LST time	Fact1 SST time
1	53	19%	35%	23	191
2	53	17%	35%	17	191
3	53	18%	35%	31	191
4	53	11%	35%	19	191

Table 1 Sample of Local System Response Time SLA.

Table 1 presents a sample of such a dimensional model. Note that we use surrogate keys [4] to represent non-measured and non-numeric values. A surrogate key is a unique identifier of an entity; it can be an integer and it is not derived. This makes a table with smaller footprint in memory, only integers are used, and enhances the performance of operations done on it.

E. Design of the Real-time Predictor

The predictor trainer, in Figure 1, is the module that creates, trains, and tests the proposed predictor. This happens on the test platform not the production platform. A predictor, fundamentally, is a classification system [2], [3]. It defines and monitors boundaries of working conditions that result in an event. It predicts the outcome of an event based on the system working conditions and loads. Our proposed light-weight predictor defines the independent variables of the local systems that impact the event of interest; the independent variables are the ones that can be controlled. The independent variables we propose to use are the number of active tasks, jobs and processes, in the system. We identify the dependent variables that control the output of the event to be the resource utilization; the compute and memory resources. They are dependent on the independent variables, but their values impact the event of interest.

Our classifier follows a logical expression model that defines the safe working boundary conditions for each of these variables by their upper values. For example, the event of interest would succeed if:

- Condition1:
  - IndependentVariable11<L11
  - ...
- Condition2:
  - DependentVariable21<M21
  - ...
- ConditionN:
  - IndependentVariableN1<PN1
  - ...

As an example, Condition1 may have two independent variables and their values that cause failure as follows: concurrent-processes <15 and Number-of-routed-packets <100. At the time of running the LSTs, the values of concurrent processes, active packets, LSTs inputs, and event value are captured from the system. They are then fed to the predictor. If the value of concurrent-processes is 13, and number of active packets is 95, the predictor predicts that the operations over the coming period of time will be successful, but issues a warning to the load-balancing-system to reduce the system loads as it is approaching failure points. The prediction value (success in this case) is compared with the actual event values that are captured from the system, ground truth, over the coming period of time. This allows measuring the accuracy of the predictor as well as false positives and negatives. We explain how to find the variable values that result in event failures in the next section.

F. Training and Testing the Predictor

Our approach uses synthetic transactions to execute the system by controlling the load to produce failures. It captures the transactions inputs, the results from the transactions, the event value, and the used loads. We can search for the classifier values that result in failures in the captured data. This is an advantage of our approach, as the findings are based on the ground truth.

The Real-Time Dynamic Failure Predictor (RTDFP) algorithm we devise, to find the classifier values that result in failures, is as follows:

1) Make a new set of LST calls with increasing intensity until the LSTs start to result in violating the SLA of the event.
2) Test the system using LST calls with loads around the failure causing loads.
3) Capture the LST inputs, system states, independent and dependent variable values, and event value into an array with a dimensional schema similar to Table 1.
4) Use search algorithm, e.g., A* search, to find the classifier values that correspond to the event failures.
5) Store these values for each local system in an array with a schema similar to that shown in Table 2. We call this table the Local System Predictor Configuration Table (LSPCT).
6) Repeat these steps, 1 through 5, and capture T instances of Table 2 until the classifier values reach a steady state; a good way to determine that is by using standard deviation for the variable values. The system continues to be in change mode until a steady state in these values is reached. The number of table instances, T, depends on the system characteristics. Some systems reach a steady state faster than others, and may require a low T value like 5 tables. Other systems with higher instability characteristics may require more tables. It is up to the system designers to define T. The T tables extracted from running sets of LST tests constitute a rolling window, i.e., the $T+1$ new LST tests will push out the results of the first LST run so that the tables have the LST test sets 2 to $T+1$.

When a steady state in the classifier values is reached, a new instance of Table 2 is created by averaging the values found in the T tables created during the training steps. This instance is called the predictor configuration table (PCT). These average values, in the PCT, are the values of the classifier variables that result in failures.

To test the predictor, we run a new set of LSTs around the failure points and capture the event value for each. We test the predictor accuracy by comparing its predictions against the event value for each test. If the accuracy of the predictor meets the goal, it is ready for use. If not, we repeat the training steps. When the predictor passes testing, the values in the PCT are stored in the predictor configuration cache, and are used as the classifier values that would predict failures in the system. Procedure 1 illustrates the RTDPF algorithm to train the proposed predictor.

Local System	Independent Var1 e.g. Tasks	Dependent Var1 e.g. CPU	Dependent Var2 e.g. Memory
1	153	68%	73%
2	149	71%	68%
3	223	67%	71%
4	196	70%	72%

Table 2 Sample Configuration of Local System Predictor.

Note that other techniques, such as regression [2], [3], can be used to find the predictor variable values that result in failure. However, we believe that searching the results of the synthetic transactions has better results as the values are found based on the ground truth coming fresh from the system.

The accuracy of the predictor is continuously monitored, by the system monitor in Figure 1. If the predictor is successful at predicting the event, the predictor is kept. If not, the system is tested until it reaches a steady state, before deciding to create a new predictor. Note that during system changes, the old predictor is kept in use until the new predictor is created. When a new predictor is created, the predictor trainer updates the predictor configuration table, which updates the dynamic real-time predictor without any impact on the service.

IV. Evaluation

We validate our approach on a large-scale enterprise online ad service, used in Microsoft, that works as an ad request

Procedure 1 RTDFP Algorithm

```
PREDICTOR TRAINER
 1: function TRAINPREDICTOR
 2:     $Counter$ = 0
 3:     $T$ = NumberOfTables2                          ▷ configured value
 4:     $FailurePoints$ = FindEventFailurePoints()
 5:     while No-Steady-State-in-Classifier-Values do
 6:         RunTestsAroundFailurePoints();
 7:         Search for values that cause failure;
 8:         Store values into ($Counter$ % $T$) of Table2;
 9:         Compute steady-state-in-classifer-values;
10:         Increment Counter;
11:     end while
12:     RunTestsAroundFailurePoints();
13:     Test Classifier Values;
14:     if ClassifierValueTest Succeeds then
15:         $PCT$ = Average Table2 $T$ Instances;
16:     end if
17:     if ClassifierValueTest Fails then
18:         TrainPredictor()
19:     end if
20: end function
FIND EVENT FAILURE POINTS
 1: function FINDEVENTFAILUREPOINTS
 2:     while no-event-failures do
 3:         Increase LST loads
 4:         Run LST calls
 5:         Capture Event-Failure-Causing-Loads
 6:     end while
 7: end function
TEST EVENT FAILURE POINTS
 1: function RUNTESTSAROUNDFAILUREPOINTS
 2:     for each $n$% below Event-Failure-Causing-Loads to $n$%
        above Event-Failure-Causing-Loads do
 3:         Run LST calls
 4:         Capture LST and system values into Table1
 5:     end for
 6: end function
```

arbitration unit. The high level diagram of the service is depicted in Figure 2. The service is composed of the ad request facade, the ad request processor, the targeting system, and the response validator. The service is a large distributed system, has challenging requirements, and operates under strict SLA, making it an ideal environment for the validation of the proposed technique. It is hosted in three continents; North America, Europe, and Asia. It receives ad requests from applications running on mobile devices and PCs. It processes each ad request, determines the source, PC or mobile, and attempts to find targeting opportunities based on app categories, device types, user information if available with user permission and consent to use, and client location. It then makes a call that has the targeting information to an ad serving network. The ad serving network identifies a suitable ad to be served to the application, and sends a response back to the service. The service validates the ad response against rules about the user, the calling application, ad type, and ad size, and returns the ad response to the calling application. The response time SLA for serving an ad is 250ms; this is the event value we watch for.

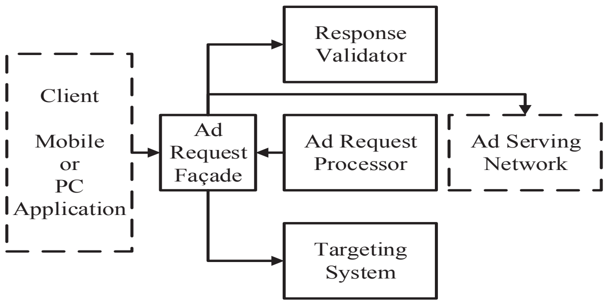

Fig. 2. The online ad service used in the experiment.

A. Implementation and Setup

We implement the LSTs for the ad service local systems and measure the SST response time for the ad request scenario. There are three local systems: ad request processor, targeting system, and response validator. For our study, we use a local ad serving network, which becomes the fourth local system.

The ad service gets about 4 billion ad requests a month. It is deployed in 6 data centers, 3 continents with 2 data centers each. The setup we have is based on the model shown in Figure 1. The transaction generator makes a call to each of the local systems and controls the various aspects of the ad request, client type, ad type, location, and user information. The transaction generator also simulates ad request calls made by multiple clients, by making simultaneous calls with different user agent information. It controls the load in two ways: the number of ad requests made by each simulated client in a given time, and the number of simultaneous ad requests representing multiple client calls. We implement a predictor instrumentation that can make up to 10,000 concurrent ad requests. By design, each user cannot request more than one ad every 30 seconds, this is a real requirement to prevent ad fraud. The maximum load that can be generated by the failure prediction system is 10,000 simultaneous ad requests every 30 seconds. This is more than 100x the expected real load of the system; so it is a good stress test. Each ad request results in about 0.5KB of data returned from the service, and this is what is captured by the system monitor into a schema similar to that shown in Table 1. The prediction system is implemented and is used for several months.

B. Data Collection

Responses to the LST calls and each local system state are collected for every LST call made to the service. These values are stored in memory arrays with the schema of Table 1, in the predictor configuration cache. In steady state, when the service is not going through any changes that warrant a predictor update, which is determined if the predictor maintains accuracy above threshold, the synthetic transactions are designed to run 100 concurrent LSTs every minute. Each LST takes an average of 25ms to complete, so running 100 tests every minute takes less than 3 seconds. Note that the LSTs are used for production testing purposes as well as for predictor verification. If the prediction accuracy drops below a threshold, we rebuild the predictor using the RTDFP algorithm. We use a high threshold of 80% accuracy, to ensure the online service maintains its failure SLA. We use the algorithms described in Section 3 to collect new data, create a predictor, train it, and test it. We find that it takes, on average, about 3-5 minutes with about 7,000 to 10,000 LSTs to generate enough failure information to start the creation, training, and testing of the predictor. Each transaction generates about 0.5KB of data, so the total data used in training is about 5MB only. In contrast, the production log has about 88GB of data a day, and requires 5-6 weeks worth of production running, as shown later in Table 3, to generate enough data that can be used to create, train, and test the failure predictors.

Using synthetic transactions for about 7 minutes we were able to generate data, create, train, and test the proposed dynamic predictor and get to accuracy close to 86%. On the other hand, we find that the average real production failure rate is less than 1%. That is an average of less than 150 failures a minute in the whole data center which has hundreds of servers. There is also the fact that production failures do not actually happen uniformly throughout the day. So from a test perspective, it can be hours, or even days, before real failures are encountered.

C. Performance Metrics

We validate the effectiveness of our approach by measuring its ability to adapt to production system changes, and still maintain high accuracy. We also compare the performance characteristics of the proposed predictor with four static predictors based on neural network, clustering, Bayesian, and decision trees algorithms. We do not implement these predictors, we use commercially available software that implement them. We choose these algorithms because of their wide use [9], [12], [5], [14] and because they represent different approaches to data mining and machine learning; they represent Artificial Intelligence, Clustering Analysis, Statistical Methods, and Decision Rules respectively [2], [3], [22].

To assess the accuracy of the proposed dynamic predictor, we use the following metrics:

- **False positives**: a prediction is called a false positive when a transaction is predicted to fail to meet the response time SLA, but the transaction meets the SLA time.
- **False negatives**: when a transaction is predicted to meet the response time SLA, but the transaction fails to meet the SLA time.
- **Accuracy**: is the ability to predict the results of the new transactions correctly; i.e., the predictor's ability to identify and exclude true errors. It is calculated as the ratio of correct predictions, which is 'all predictions' minus 'true errors', divided by all predictions.

D. Detecting Failures in Dynamic Environments

To test the predictors' ability to adapt to real-time changes, we start with a baseline configuration and configure/program the four commercial static predictors. The proposed dynamic predictor is built during the experiment. The baseline configuration is a cluster of 10 compute servers. Each compute server is a quad-core intel Xeon server with 12 gigabyte RAM. We deploy the predictors into the test client. We run the synthetic transactions, LSTs, and SST. We make the following changes: increase the available compute resources by 5 more processing servers and wait for thirty minutes, and then increase by 5 more servers for a total of 20 servers, and wait for 30 minutes. We drop the compute resources to 6 servers and wait for thirty minutes, and then drop the servers to 3 and wait for 30 minutes. During that period, we capture, every minute, the results of the LSTs and SST, event value, which is response time for the tests, and predictions made by all predictors. The event value we capture is the ground truth. We then measure the false positive rates, false negative rates, and the accuracy of all predictors to determine how they react and adapt to production system changes.

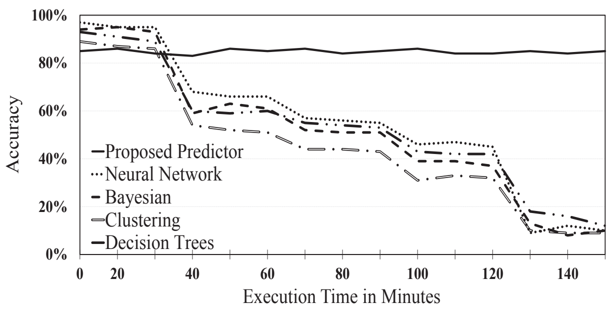

Fig. 3. Accuracy of the proposed predictor vs. current predictors in dynamic environments.

Figure 3 shows the accuracy of all predictors in dynamic environments over 2.5 hour period. Figure 4 shows the false positives, and Figure 5 shows the false negatives for all predictors. As expected, the current static predictors fail to adapt to the resource changes; they drop in accuracy after the first configuration change, which is adding 5 processing servers, and never regain their accuracy back. It is worth noting that after increasing the compute resources above the baseline, it is the false positives that are responsible for the drop in accuracy. In other words, many transactions are actually successful, but are predicted to fail. While after reducing the compute resources below the baseline, it is the false negatives that are responsible for the drop in accuracy. On the other hand, Figures 3 through 5 show that the proposed predictor is able to maintain high accuracy. It manages to maintain about the same average of false positives and false negatives post the compute resource changes. On average, it takes the proposed predictor about 7 minutes to adapt to the changes and reflect current state of the system.

The transient period, which is the period from the time we add or remove resources until steady state in the proposed predictor is reached, is on average about 7 minutes. That's how long it takes to update the proposed predictor. During the transient period, the current static predictors and proposed dynamic predictor drop in accuracy. However, since the pro-

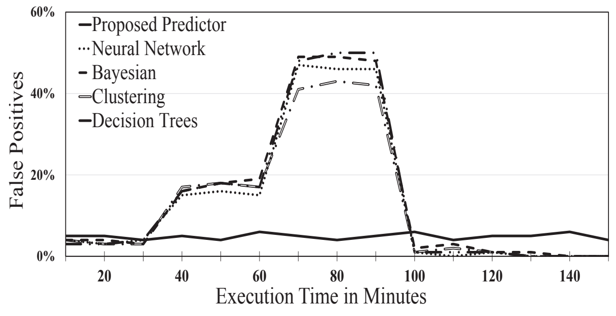

Fig. 4. False positives of the proposed predictor vs. current predictors in dynamic environments.

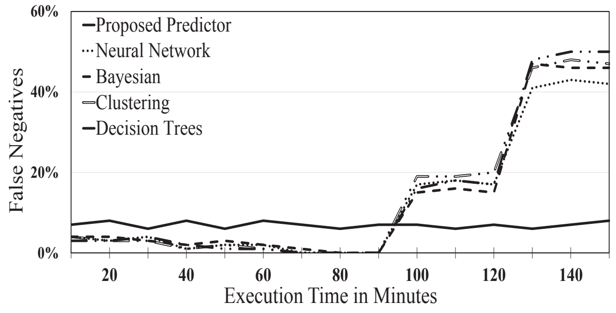

Fig. 5. False negatives of the proposed predictor vs. current predictors in dynamic environments.

posed predictor was updated after each change, its accuracy during the transient period is far better than the static ones, because their accuracy drop accumulates over time, as shown in Figure 3.

E. Receiver Operating Characteristics of the Predictors

To compare the performance characteristics of the proposed dynamic predictor with the four current static predictors that are created from real production logs we build a Receiver Operating Characteristic (ROC) curve for each predictor. The ROC curves show the relationship between the specificity and sensitivity of the predictors. This is a standard methodology for comparing predictor accuracy over a range of controlled input specificity. The following metrics are used to generate the ROC curves:

- True Positives: when a predictor correctly predicts a transaction to fail to meet the SLA time.
- True Negatives: when a predictor correctly predicts a transaction to meet the SLA time.
- True Negative Rate: is the ratio of true negatives to the sum of true negatives and false positives. This is also known as **Specificity**.
- Recall: is the ratio of true positives to the sum of true positives and false negatives. This is also known as **Sensitivity**.

Figure 6 shows the ROC curves of the predictors in their steady state, when the static predictors are still relevant to the system; note that post system changes, it is no longer possible nor meaningful to plot the ROC curves for the static

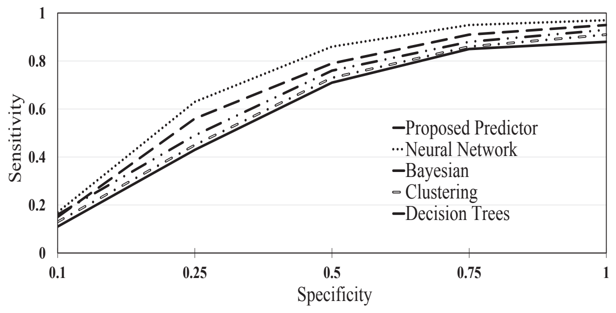

Fig. 6. Predictors ROC Curves.

predictors. The goal of this comparison is to show that the proposed predictor has comparable and viable performance characteristics to industry standard commercial predictors. The main advantage of the proposed predictor is that it takes a fraction of the time to build, train and test in real-time, whereas the static predictors require orders of magnitude more data and time to build, train, and test. Table 3 shows a comparison between the predictors in terms of their data generation and processing times, as well as data Recall.

The downside to the proposed predictor is that it requires high-level knowledge about the system to be implemented as part of its testing whereas the static predictors do not require that knowledge. We argue that this knowledge is already required by the system designers, implementers, and testers who are the intended audience of our suggested approach.

Predictor	Generation Time	Processing Time	Data Recall
Proposed	3-5 Mins	2-4 Mins	100%
Bayesian	5 Weeks	21 Hrs	0.91%
Clustering	6 Weeks	18 Hrs	0.57%
Decision Trees	5 Weeks	19 Hrs	0.68%
Neural Network	5 Weeks	17 Hrs	0.97%

TABLE 3 - Processing time and data needed by each predictor.

F. Performance Analysis over Long Period

Figure 7 shows the daily accuracy, false positives, and false negatives of running the proposed predictor for more than three months. We note the stability and consistent behavior of the predictor over that period of time.

G. Overheads

The overheads incurred by the proposed predictor fall into two categories: during steady state, and during system changes. During steady state, which is a state where the predictor's accuracy is maintained above 80%, 100 tests are run every minute for an average of about 2.5 seconds. Adding about 2.5 seconds worth of production testing constitutes less than 5% impact on each production server, which is low. The average real production requests per minute made to each production server is about 150 requests. Each request takes an average of 25ms. So the impact on production servers is low.

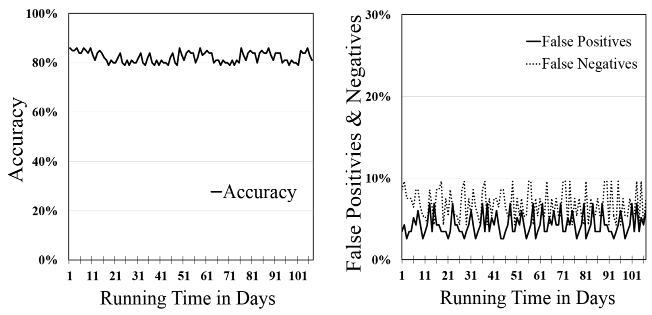

Fig. 7. Performance of the proposed predictor over 3 months.

During system changes, when the prediction accuracy starts to fluctuate and drop below 80%, we double the testing load. We run the test every 30 seconds, which results in doubling the amount of time we use the system. We use the system for about 5-6 seconds every minute, which is still low. The fact that the LSTs are part of the required production testing, and are used to accomplish other jobs than the prediction maintenance, like testing the actual functionality of the local systems, makes the investment less of an overhead. The test client servers are already dedicated to the production testing functionality, and as such they are not considered an extra cost.

There is no production code overhead special to the predictor functionality. The measurements we take, such as resource utilization and tasks measurements, are provided by the operating system of the production local systems. These are taken at equidistant time series, with or without our approach, as means of monitoring the health of production servers.

V. Conclusions and Future Work

Current approaches to failure prediction are static and can not keep up with the changes that happen during the real-time lifecycle of online services. Static predictors require massive amounts of data to rebuild, which is not possible in real-time. Static predictors have their strengths and areas of application; they do not require specific knowledge about the system, and are successful in static environments and situations. Online services, however, have dynamic situations that require them to change often.

We presented a dynamic approach to creating and maintaining failure predictors for online services in real-time which shows superior ability to stay relevant and maintains high accuracy throughout the real-time lifecycle of the online service. Using the proposed real-time dynamic failure prediction (RTDFP) algorithm, we can regenerate an online service failure predictor post system changes in a few minutes with a few megabytes of test generated data.

A future follow up is to study the ability of dynamic predictors to work with more complex cloud-based systems such as search engines that require monitoring multiple complex, and inter-dependent events at a time.

References

[1] "It channel," http://www.itchannelplanet.com/.

[2] J. Han, M. Kamber, and J. Pei, *Data Mining Concepts and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[3] M. Kantarczic, *Data Mining Concepts, Models, Methods and Algorithms*, 2nd ed. Wiley and IEEE Press, 2011.

[4] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed. Wiley Computer Publishing, 2013.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. of 22nd ACM Symp. on Operating Systems Principles (SOSP '09)*, Big Sky, MT, October 2009, pp. 117–132.

[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. of 10th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, April 2012, pp. 353–366.

[7] M. E. Snyder, R. Sundaram, and M. Thakur, "Preprocessing dns log data for effective data mining," in *Proc. of IEEE International Conference on Communications (ICC '09)*, Dresden, Germany, June 2009, pp. 1366–1370.

[8] Z. Zheng, Z. Lan, B. H. Park, and A. Geist, "System log pre-processing to improve failure prediction," in *Proc. of 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, Estoril, Lisbon, Portugal, June 2009, pp. 572–577.

[9] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *Proc. of 1st Symp. on Networked Systems Design and Implementation (NSDI '04)*, San Fransisco, CA, March 2004, pp. 309–322.

[10] F. Salfner and S. Tschirpke, "Error log processing for accurate failure prediction," in *Proc. of USENIX Workshop on Analysis of System Logs (WASL '08), in conjunction with 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[11] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. of 3rd Symp. on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006, pp. 115–128.

[12] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. of 20th ACM Symp. on Operating Systems Principles (SOSP '05)*, New York, NY, December 2005, pp. 105–118.

[13] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish, "Improving availability in distributed systems with failure informers," in *Proc. of 9th USENIX conference on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, April 2013, pp. 427–442.

[14] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, M. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proc. of 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, March 2011, pp. 43–56.

[15] "Real-time monitoring," http://techblog.netflix.com/2012/11/hystrix.html. and https://github.com/Netflix/Hystrix.

[16] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Detecting liveness bugs in systems code," in *Proc. of 4th USENIX Symp. on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, April 2007, pp. 243–256.

[17] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y. Wang, "Webprophet: Automating performance prediction for web services," in *Proc. of USENIX Symp. on Networked Systems Design and Implementation (NSDI '10)*, San Jose, CA, April 2010, pp. 143–158.

[18] A. Viswanathan, A. Hussain, J. Mirkovic, S. Schwab, and J. Wroclawaski, "A semantic framework for data analysis in networked systems," in *Proc. of 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, March 2011, pp. 127–140.

[19] L. Riungu-Kalliosaari, O. Taipale, and K. Smolander, *Testing in the Cloud: Exploring the Practice*. IEEE Software, 2012.

[20] E. Elliot, *Testing in Production A to Z - TiP Methodologies, Techniques, and Examples*. IEEE Software, 2012.

[21] "Testing in production," http://blogs.msdn.com/b/seliot/archive/2011/06/07/testing-in-production-tip-it-really-happens-and-that-s-a-good-thing.aspx.

[22] F. Provost and T. Fawcett, *Data Science for Business: What you need to know about data mining*. O'Reilly Media Inc., 2013.