PREDICTING SERVER FAILURES WITH MACHINE LEARNING*

A. Brian Lai†, UC San Diego, La Jolla, USA
Andy Li†, UC Berkeley, Berkeley, USA

Abstract

Unexpected server failures incur a large cost. Using data that is continuously collected by monitoring software, we can more accurately understand the processes that each server is used for. The deviations in server performance help diagnose when servers may malfunction. We demonstrate a machine learning model that can predict whether a server fails within 60 days with high accuracy. In specific, our models predict the occurrence of hard drive failures as they constitute over 80% of all server failures within the data center.

INTRODUCTION

Server maintenance at SLAC Accelerator Laboratory has traditionally taken a reactive approach. These unexpected server failures incur a high cost. When a server failure or malfunction is detected, resources must be redirected to address the servers that require maintenance and away from current projects. We want to minimize the amount of unplanned downtime as we are often forced to wait for a replacement part to be shipped.

Recently, the Computing Division implemented the five-year cycle to move towards a more proactive approach. The five-year cycle replaces the oldest 20% of servers each year so that, eventually, no server older than five-years will be in use. This program was started because it was believed that older servers tend to fail more often than their newer counterparts. By establishing a limit on how old a running server can be, the hope is that the data center will face fewer cases of fewer unexpected server failures be able to better serve the researchers that utilize the facility while minimizing the number of unplanned person-hours.

The five-year cycle is an improvement to the "run to failure" approach, but some servers tend to fail well before and well after the five-year mark. Trying to replace servers that typically fail before the five-year mark means that these servers will not be replaced proactively through five-year cycle program, but rather out of necessity. In these cases, we must absorb the cost associated with a server failing unexpectedly and the unplanned person hours that are necessary to repair or replace the server. Replacing servers that tend to last longer than five years means that we will replace servers and spend money unnecessarily. Through this project, we continue the effort to develop a more risk based approach to replacing servers.

*Work supported by SLAC Accelerator Laboratory, Department of Energy
† b4lai@ucsd.edu, acli@berkeley.edu

BACKGROUND

Within the Computing Division, Ramon Lim has already been done to determine whether age was the only indicator of whether a server would fail.

Lim's findings determined that, in addition to age, the server's usage was a strong indicator of whether a server would fail. Intuitively, this makes sense. Given two identical machines, in which one machine consistently performs computationally expensive tasks while the other machine is often idle, we would expect for the machine that handles more computationally expensive processes to fail sooner. Under the five-year cycle, however, both these servers would be replaced at the same time.

Building off Lim's project, we approach the problem of predicting server failures by incorporating both the intrinsic machine properties as well as how the server is being used over time and to develop a more risk based approach to replacing servers and to ultimately minimize the overall cost associated with maintaining and replacing servers. By prioritizing the usage of the server, we can get a deeper understanding of the processes that each server is tasked with. This level of granularity will help predict the occurrences of server failures will help facilitate in scheduling necessary maintenance hours while minimizing the number of unplanned person hours and unnecessary maintenance hours, and determining which parts to keep in inventory.

DATA SOURCES

Ganglia

The Unix machines within SLAC's data center are monitored through Ganglia, a simple daemon that runs on each of the nodes. This software tracks metrics such as the number of bytes inputted, number of bytes outputted, and CPU load. This data gave us a deeper insight as to how the servers were being used. Presently, Ganglia is configured to measure the average each metric for every interval of 60, 1440, 10080, 40320, and 345600 seconds. Only the measurements that are stored in 345600 second, or four-day, intervals however, is stored historically while the other data is removed after a period of about four days. To capture the times when the servers encountered failures, we were limited to using the 345600 interval metric data. To pull the data from Ganglia, a few simple terminal commands can easily fetch the files where the information is stored. Generally, these commands followed the format:

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-76SF00515.

/var/lib/ganglia/rrds/<cluster>/<node-name> on gmetad

rrd file

rrdfetch

This set of commands makes it simple to retrieve the data collected by Ganglia. The data is initially pulled as a rrd, reduced resolution dataset, file. This file type, however, is not easily parsed by Python and would later require some additional work to convert them into a more readable file format.

Failure Logs

Every morning, a script is run to check the health of the servers. This script helps identify issues and malfunctions. Any problems that are identified are recorded manually on text file. In this text file, there is a brief description of what the error is, whether the error has been resolved, the server, and the date the error was detected. This text file has been maintained for over 10 years. As expected, there are a considerable number of typos, formatting issues, and missing fields. This is, however, SLAC's most accurate logs of the failures that are occurring in the data center and it gives us a timeline of when each failure occurs.

DATA PREPARATION

Convert rrd files to XML

A simple bash script can be used to convert all the rrd files pulled from Ganglia into XML files that can be easily parsed in Python. To convert one file, you can use:

rrdtool dump filename.rrd filename.xml

XML to Dataframe

Initially there were over 19,500 xml files that we needed to parse through. We later learned that some of these files were empty and others stored metrics for pieces of equipment that were not servers. For every metric of a given machine, there was a separate, distinct file. All files related to servers had file names in the following format:

file_name = server + '.slac.stanford.edu-' + ganglia_char + '.rrd.xml'

Furthermore, each file followed the same formatting. In every file, the recordings for each time interval was written to the same line numbers as the other characteristics for that particular machine. Understanding the structure of the file and how the file are named, allow us to write a simple function to extract the data from all the xml files for a particular machine and format the data into a pandas dataframe using the Python's pandas, os, and BeautifulSoup libraries. This function is shown in the Appendix.

After we process one feature, we can iterate through the other features in the feature list and merge each individual dataframe into a larger, aggregate dataframe that contains the data for all the features for that machine.

After the data for one machine is nicely formatted, we can combine the dataframes for each machine into one dataframe that represents all the servers in the data center. Instead of concatenating all these dataframes to one another, we implemented this aggregation using arrays from the numpy library as it was significantly less computationally complex. After combining the data for all the machines, we could convert the numpy array back into a dataframe.

Initial Feature Screening

Using the *describe* function we can start to get an idea of the data that is being recorded. The describe function shows us the minimum, 25th percentile, median, 75th percentile, maximum, mean, and standard deviation for each of the features in the dataframe. This statistical summary makes it easy to determine which features are constant. Instead of keeping all the features that are constant, we dropped all but one, leaving one of the constant variables to serve as a bias term. This way, our models will not require as much training time while preserving the information that is important for predicting server failures.

Handling Missing Values

We identified which columns were missing values by using pandas' *value_counts* function and comparing this with the number of samples in the dataframe. Instead of removing the sample where there were one or two missing features, we replaced these missing values with the median for that feature as the median tends to be more stable than the mean. Note that we are replacing the values of -1 as opposed to NaN because we changed the null values to -1 so that we could apply to “to_numeric” function.

df["feature_name"].replace(df["feature_name"].median(), inplace=True)

Timestamps and Time Gaps

When the data was initially pulled from the XML files and inserted into the dataframe, the dates were represented as a string in the format "MM/DD/YYYY". These values were difficult to use directly given the differences in the number of days in the month. We decided to converted this strings into numbers. More specifically, chose to represent each date as its epoch, the number of seconds since January 1st, 1970. Converting these strings to a numeric value would make it much more convenient to derive other features.

After sorting the samples in the dataframe by both machine and epoch, we noticed that there were intervals that were larger than the expected 4 days. We learned that these larger gaps were a result of the machine being moved, the machine being idle, or a combination of the two.

We set the threshold for distinguishing between the two scenarios at 60 days. It was important to come up with a heuristic to differentiate between the two because there are

marked performance differences between machine that have completely shut down to be relocated and machines that were still running, but were merely idle. This heuristic, however, can and should be improved, but ideally, to have the most accurate status, each time a server is relocated it should be logged in a database.

Using this heuristic, whenever a machine was moved we changed its name for every recording after that. For example, if a machine was named "server-A", its new name after being moved would be "server-A-1". If it was moved again, its name would change to "server-A-2". In addition to changing its name, we defined a feature that tracked whether a machine had been moved or not.

Failure Logs

A significant amount of manual work was done to clean the data that was originally in this text file. Once that text file was reasonably clean, the data was parsed into a pandas dataframe. In this dataframe, we were primarily focused on when the failure occurred and what server the failure affected. In the dataframe that contained the recordings of the Ganglia metrics, we first sorted the dataframe by machine and epoch. This way the data was formatted similarly to a time series. From there, we could write a function that took a list of the failure times for a particular machine and returned the times the time that the previous failure occurred and the time that the next failure would occur in terms of epoch. Once we derived these values, these two features were appended to the Ganglia dataframe.

Deriving Features

In addition to adding indicator features for whether a server was moved, the times of the previous and next failures, epoch, we chose to include a couple additional features.

Since hard drive failures for a given server seemed to happen in clusters. Therefore, we derived the additional features, *time_since_prev_failure* which represented the time since the previous failure. If there is some pattern for the hard drive failure clusterings, this feature would essential. Computing this feature was reasonably straightforward given that we already derived when the previous failure occurred.

By taking the difference between the time until the next failure and the sample's epoch, we could easily derive the time until the next failure.

To frame predicting server failures as a supervised learning problem, we must first define what constitutes a positive and negative observation. We chose to define a positive observation as a failure occurring within 60 days. This number of days, from an operational standpoint, is long enough such that the staff has sufficient time to order the necessary parts and to schedule the required person-hours. This is a parameter that should be explored further.

By approaching the problem this way, we can define a new variable *error_days60.* This variable is an indicator feature that a failure would occur within the next 60 days. A positive observation would be defined as an observation in which a failure occurs within 60 days.

Encoding Categorical Variables

Using pandas' built in *get_dummies* function, we one-hot encoded the categorical features such the server model.

MODEL SELECTION

Defining a good model

Given that the servers function normally for a large majority of the time, the data is heavily skewed towards negative cases, the scenario where a server failure does not occur. Thus, simply using accuracy is not a good measure of performance. Ideally, we want to minimize the number of false-negative cases in which the model predicts that an error does not experience any failures, but the server does fail. This is especially costly because these inaccurate predictions require unplanned person hours to fix the servers. False positives are also costly and should also be minimized, but are relatively cheaper. In this case, maintenance is performed unnecessarily. The cost is comparatively less, however, because hard drives are generally inexpensive. True positives and true negatives are accurate predictions and should be maximized. Taking these four metrics into account we chose to evaluate a model's performance based on its F1-score. A mathematical representation of the F1 score is shown.

$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$Precision = \frac{T_p}{T_p + F_p}$$

$$Recall = \frac{T_p}{T_p + T_n}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Implementation

We used an 80/20 train-test split. We implemented a regressor model as opposed to a classifier to represent a threat level associated with the server failing. We used a threshold level of 0.4 to map to a positive case. That is, if the threat level was higher than 0.4, we would expect the server to fail.

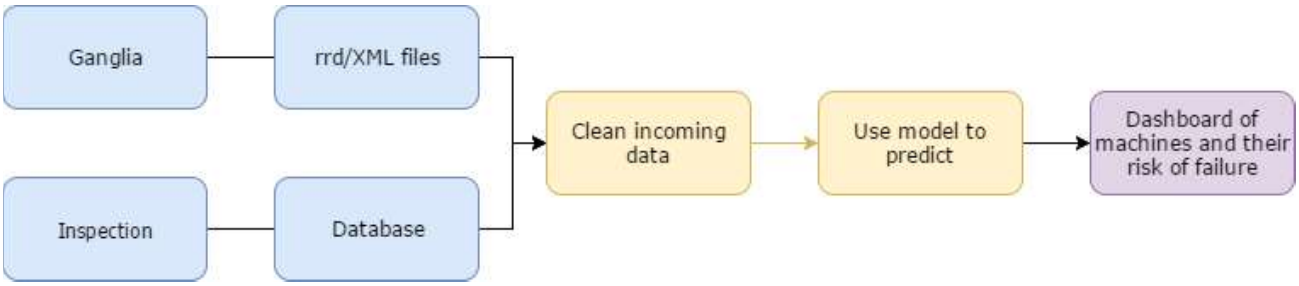

Figure 1: Simple Data pipeline

Individual Timestamps

The first approach we used was to consider each individual timestamp as a distinct sample. We could easily track which machine we were referring to as it was one of the features. The output variable would be the *error_days60* column.

Sliding Window

The next approach was to implement a sliding window approach. We aggregated the values for all the features. In addition to aggregating the values for these features, we found the variance for these values. We thought that, perhaps, if the performance fluctuated widely, it would be indicative of poor server health. We included other elements of the statistical summary such as the mean as well. We chose to implement a 16-day sliding window. In the future, however, the size of the sliding window should be explored further. Given the large number of added features, it is understandable that the training time was significantly higher. Since the performance is markedly better than the individual timestamp model, it appears that the variability of the performance of the server is a factor that should be considered when building the model.

Model	Algorithm	F1 Score
Timestamps	KNN	0.75
Timestamps	BST-DT	0.49
Sliding Window	KNN	0.82

Holdout Group

While training the previous models, we had assumed that no failure would occur for the samples that were collected the 60 days prior to when the Ganglia data was pulled. This is not the most accurate representation. Since we did not have the ground truth observation, we chose to remove this data from the dataset. When removed these sample from the dataset, the F1-Score increased significantly.

Model	Algorithm	F1 Score
Timestamps	KNN	0.943

We predict that implementing a sliding window approach and making use of the holdout group should demonstrate even stronger performance.

NEXT STEPS

We make the following recommendations to effectively utilize our model and findings and ultimately more intelligently budget for servers, allocate people-hours, and increase confidence.

Infrastructure and Processes

First, we highly recommend the development of infrastructure and processes to utilize our preliminary findings. This is necessary to automatically and proactively understand which machines may encounter errors. Without the proper processes, a staff member would have to locate and retrieve the proper data; run the necessary scripts to process the data and make predictions; present said data in a format easy for human use. With a proper pipeline, we can proactively send data about machine performance and recent failures directly to our scripts, and display the result in a pleasant way. This may take some time and can be flexible, so this can be done modularly. A sample recommended process is shown.

Automate Testing

Presently, the health of servers is checked by a script that is manually run every morning. This process can be automated. Instead of checking the health of the serves every day, if this script can be run automatically run several times throughout the day, we will also have a more precise measurement of when issues arise which can help improve the precision of the model. If this script can be automated, the findings should also be able to be written to a database. This would significantly cut down on the number hours and resources that are necessary to clean the text file that contains the failure data.

Refactor Cleaning Process

Cleaning and inputting the data should be refactored into a program that can be simply and easily run. The code from this project includes code that was used for testing and is not meant to be run. Blocks of code in the attached Jupyter Notebooks can be used in this program.

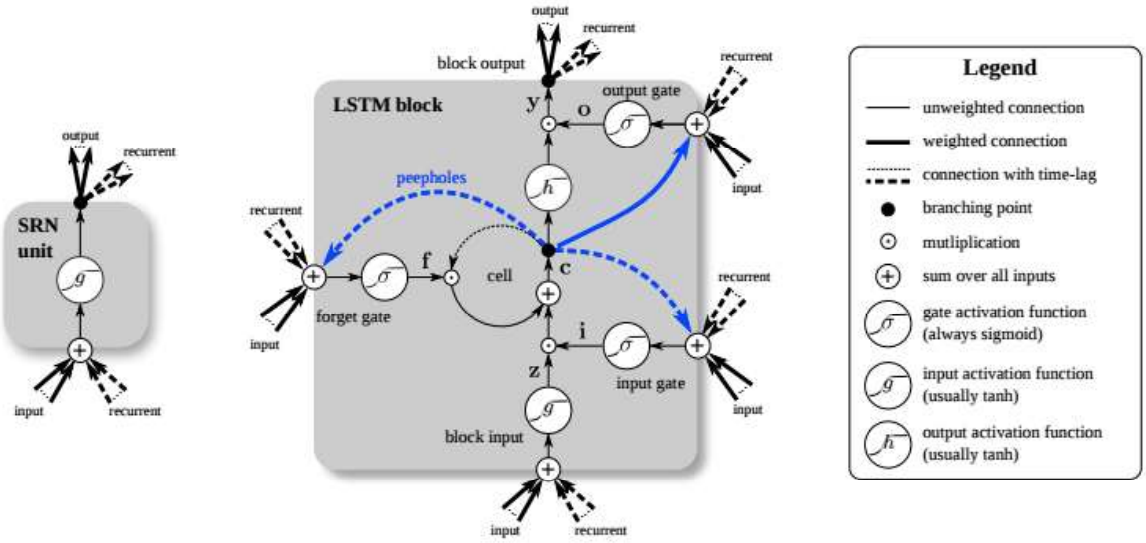

Figure 2: Detailed schematic of a RNN unit (left) and LSTM block (right) as used in the hidden layers of a RNN

Dashboard

Given the vast number of resources dedicated towards ETL and machine learning for Python, it is convenient to process and train the data in Python. An elegant solution for creating a dashboard for the failure risks for each server would be to display this information using the Django framework. By creating a dashboard, anyone, even those who are not from a programming background, can use the dashboard and devise a plan that would best utilize the budget allocated for server maintenance.

More Data

SMART (Self-Monitoring, Analysis and Reporting Technology) attributes is a set of features that are monitored by the hard drive itself and can be easily collected through other monitoring applications such as Ganglia. The infrastructure for viewing real-time SMART metrics has been implemented by hard drive manufacturer as a means of tracking the health of the hard drive. SMART attributes were initially to serve as an early detection system for hard drive failures, but an estimated 36% of hard drives still fail without any warning. We do still believe these metrics to be important factors to consider in the predictive model. A group from Google published a paper found that a small subset of features are highly correlated with hard drive failures. After showing the first-scan error, hard drives were 39 times more likely to fail within 60 days in comparison to similar hard drives servers that did not display this error [1]. Incorporating these metrics would be bolster the current machine learning algorithm and help provide deeper insights of the performance of the hard drive.

While tracking and recording SMART attributes would be useful it is important to note that the specific attributes that are tracked by each individual hard drive is determined by the manufacturer. One potentially useful feature would be the temperature of the hard drive as even a one degree difference could affect the performance of the hard drive. Although some hard drives do measure this feature, other hard drives do not. Investing in sensors may not be cost efficient, but giving preference to hard drives that do track this metric could help create a more robust predictive model.

Deep Learning

Given the amount of data we could reasonably use, we were limited to using the classical machine learning algorithms as deep learning models require significantly more data. If we collect data every minute instead of the four day averages we would have 5760 times as much data.

Not only would the more data allow us to improve the classical machine learning models and predict server failures with greater precision, we could also embark into develop deep learning models.

One method to implement deep learning is to train a simple feedforward network, but perhaps a more natural approach would be to implement a LSTM RNN (Long Short-Term Memory Recurrent Neural Networks). RNNs perform exceedingly well for sequenced data such as the time series as it "remembers" its previous states. LSTMs are a type of RNN that overcome the issues that RNNs face when dealing with a large number of time steps. If we are able to gather the data at the level of granularity of one-minute intervals, using LSTMs will be especially crucial.

Online Learning

One potential downside of tracking the usage of each server at one-minute intervals is that all this data should be stored. Given how much data there could potentially be, storing this data could become very expensive. If we were to implement an online learning model, we could continually train the model and discard the data after it has been incorporated into the model. This way, we can reap

the benefits of having a lot of data while only storing a small fraction of it. This, however, can only be used if the most has already been proven to perform well. There exist numerous algorithms that can implement this sort of learning and is often collectively referred to as "mini batch" learning algorithms.

REFERENCES

[1] Pinheiro, Eduardo, et al. “Failure Trends in a Large Disk Drive Population.” USENIX Conference on File and Storage Technologies (FAST’07), Feb. 2007.

APPENDIX

XML to Dataframe

```
def xml_process(server, ganglia_char):

    # 1. Read File
    path = 'Data/'
    file_name = server + '.slac.stanford.edu-' + ganglia_char + '.rrd.xml'
    with open(path + file_name) as data:
        xml_data = BeautifulSoup(data, 'xml')

    # 2. Get Values
    day4 = xml_data.find_all('rra')[4].find_all('row')
    day4_time = xml_data.find_all(string=lambda text: isinstance(text, Comment))

    # 3. Process Values
    time_sec = map(lambda x: x[1:].split(' ')[4], day4_time[-374:])
    value = map(lambda x: float(x.contents[0].contents[0]), day4)

    # 4. Format DataFrame
    return_df = pd.DataFrame([time_sec, value]).transpose()
    return_df.columns = ['epoch', ganglia_char]

    # 5. Clean NaN, Make Columns Numeric
    return_df.loc[pd.isnull(return_df[ganglia_char]), ganglia_char] = -1
    return return_df.apply(pd.to_numeric)
    # get all the data from all the xml files for one machine, data is packaged
    # into a dataframe
    def process_machine(machine):
        total_df = None
        for c in characteristics:
            df = xml_process(machine, c)
```

```
        if total_df is None:
            total_df = df
        else:
            total_df = df.merge(total_df)
    if total_df is None:
        return False
    return total_df
```

Converting String to Epoch

```
# define function to convert a string to epoch
from datetime import datetime

def convert_time_string(time):
    return int((datetime.strptime(time, "%Y/%m/%d") -
                datetime(1970, 1, 1)).total_seconds())
```

Identifying Time Gaps and Renaming Server Names

This process is a quite intricate. For each unique machine name, we grab all the rows in the Ganglia dataframe that measure that specific machine. Then, we sort by the epoch and make a copy of the epoch column. We stagger the two columns so that they are one time-step removed from one another. By taking the differences between the two columns, we can iterate through the column and if the difference is greater than 60 days, it will be set to 1 while the other values will be set to 0. Then, we'll use the cum_sum function this array. Using this array, we'll derive the new machine names.

```
# threshold for determining 'small' or 'large' gap in days
# curr_machine: machine that you want to examine
# should not be directly called, will be used by change names
def identify_gaps(machine, threshold=60):
    # find the rows for curr_machine
    machine = pd.DataFrame(nan_df[nan_df["machine"].str.contains(machine)])

    # sort by epoch
    machine = machine.sort_values("epoch")

    epoch_machine = np.array(machine["epoch"]) # stores the epoch times
    epoch_shift = epoch_machine.copy() # stores the shifted epoch times
```

```
    epoch_shift = epoch_shift[1:]
    epoch_shift = np.append(epoch_shift, epoch_shift[-1]+345600)

    # result represents the intervals between successive logs
    result = epoch_shift - epoch_machine

    relocated =  np.array([])
    threshold_seconds = threshold*60*60*24
    for gap in result:
        if gap > threshold_seconds:
            relocated = np.append(relocated, 1)
        else:
            relocated = np.append(relocated, 0)
    indices = indices = np.cumsum(relocated)
    return indices
    def change_names(machine, threshold):
        # create the mapping
        new_machines = identify_gaps(machine, threshold)
        curr_machine = nan_df_test[nan_df_test["machine"].str.contains(machine)]
        # create a new column
        curr_machine["new names"] = curr_machine["machine"]
        # set new names in "new names" column
        for index in range(len(curr_machine["new names"])):
            if new_machines[index] != 0:
                curr_machine["new names"][index] = curr_machine["new names"][index] + '-' +
                                                   str(int(new_machines[index]))
```

Finding the Next and Previous Error For a Machine

```
# this function returns when the previous and next failure occurs, given the time of the current sample
# and a list of failure times for that machine
def prev_next_error(row, fail_times):
    curr_time = row["epoch"]
    prev_time = -1
```

```
    next_time = -1
    for failure_time in fail_times:
        if curr_time >= failure_time:
            prev_time = failure_time
        elif curr_time < failure_time:
            next_time = failure_time
            break
    return prev_time, next_time
```

We apply this function to every row so that we can find the previous and next failure times for each sample.

Finding the Next and Previous Error For a Machine

```
# combined_df will incorporate failure data as well as ganglia data
combined_df = None
for key in ready_keys:
    df = machine_dfs[key].copy(deep=True)
    fail_times = sorted(list(hd_error_df[hd_error_df["name"]==key]["epoch"]))

    df["name"] = key
    df["nextFailure"] = df.apply(lambda x: prev_next_error(x, fail_times)[1],
                                 axis=1)
    df["prevFailure"] = df.apply(lambda x: prev_next_error(x, fail_times)[0],
                                 axis=1)

    df["timeToFailure"] = df.apply(lambda x: x["nextFailure"] - x["epoch"] if
                                 x["nextFailure"] != -1 else -1, axis=1)
    df["timeFromFailure"] = df.apply(lambda x: x["epoch"] - x["prevFailure"]
                                 if x["prevFailure"] != -1 else -1, axis=1)
    if combined_df is None:
        combined_df = df
    else:
        combined_df = pd.concat([combined_df, df])
```