# Nodeable Stream Analytics (Juggaloader)

**Updated: August 24, 2012**

**Table Of Contents**

[Nodeable Stream Analytics (Juggaloader)](#h.bxlnhqbrllt8)

[Introduction](#h.3qg3xnlch0r1)

[Message Overview](#h.1sejqjuudju0)

[Structure of the Data](#h.5wisvejy7eno)

[Algorithms](#h.c587po5kn02)

[Computations](#h.gai0on5sa6is)

[Generation of Digest Messages](#h.tpyop2slg0to)

[Summaries](#h.8emc0uv4c0ew)

[Anomalies](#h.axijeypeeirm)

[Correlation](#h.g9sjo15gu49i)

[Causality](#h.h8bu84beh21f)

[Predictive Analysis](#h.d5gwhjqxf41s)

[Technology](#h.gv5lscaabc8z)

[Appendix](#h.dqbkbr5pz64n)

[Terminology](#h.ep0j3ry8n36n)

[Low-Level Design Notes](#h.z3myz7jitgxl)

## Introduction

This document describes the high-level design of the framework for generating in-stream digest messages and statistical [reports](https://docs.google.com/a/nodeable.com/document/d/1Z1VQxLACdJPoYhC2OZ0P9MFxI7BPuFaVXafKKM9dq2Q/edit)[[1]](#footnote-1) from the raw data events streaming into Nodeable. No assumptions are made about implementation details in this design.

The system will perform stream analysis on incoming events in order to support the generation of Insight messages that:

1) **summarize** trends in raw events such as counts, averages, diffs, max/min

2) report on statistical **anomalies** divergent from those trends

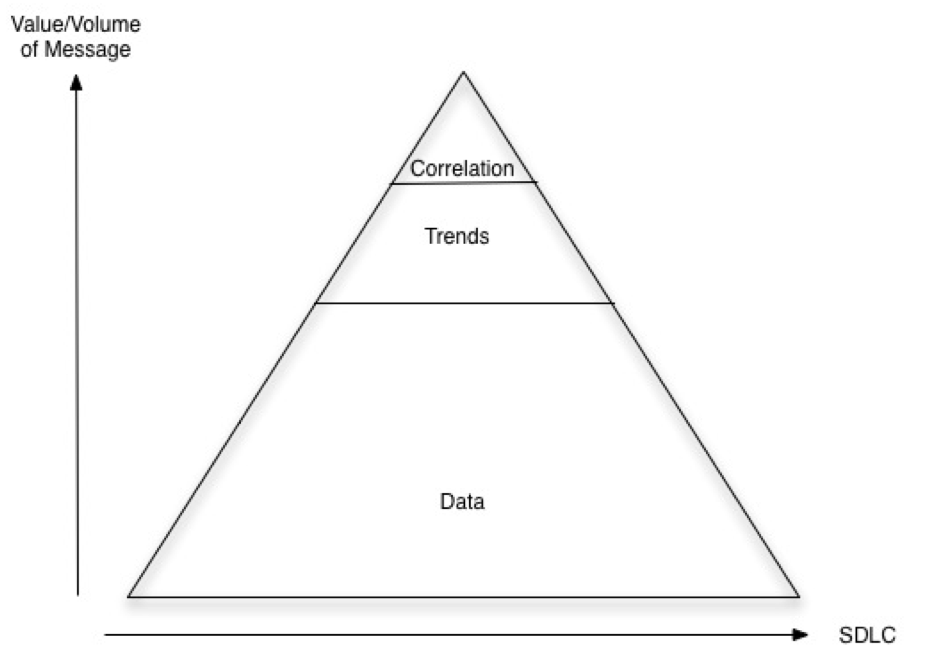
3) **correlate** anomalies across event sources and

4) determine **causality** of correlated events

5) do **predictive** analysis for various features of the data

## Message Overview

In the most general sense, an input event stream can be any time series sequence of numerical values that arrive somewhat regularly. They can be plotted as a function with an x-axis of time.



In the Nodeable app, input events can be of various types such as AWS, Github, JIRA, RSS, and custom connectors. For example, for an AWS connection the input event might include the number of raw inventories or instances that are running at any given moment.

For each type of these events (the y-axis), the following features will be calculated on the fly and stored:

1) The raw value of y

2) The first derivative of y with respect to time, dy/dt

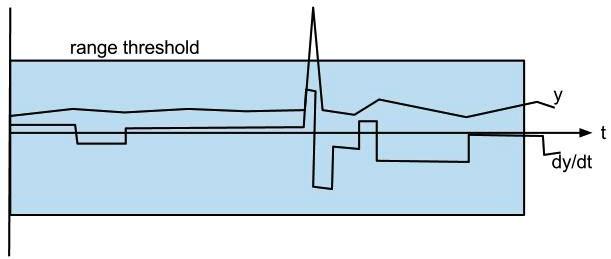
3) The incremental (windowed, decayed) average and standard deviation values of y

4) The average value of dy/dt

5) The min, max of y, dy/dt

6) the diff from the previous time sample

The calculation of most of these features will depend on the frequency at which the events arrive, and this could vary depending on the type of event.



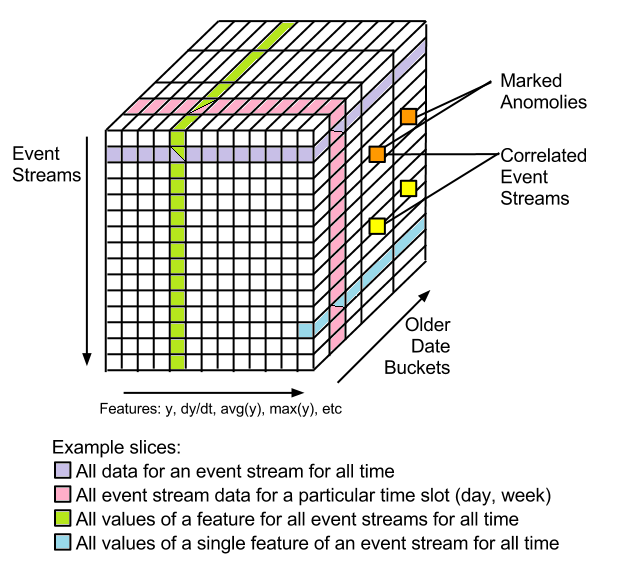
The following table shows values calculated for these features for various examples of event streams: # of instances, CPU percentage, app ajax calls/second. In this example, they are each from a subset of 3 samples that arrived at 5 second intervals.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **t** | **y** | **dy/dt** | **avg(y)** | **avg(dy/dt)** | **min y** | **max y** | **diff** |
| #instances | 10:11am | 5 | 0.003 | 2.5 | 0.003 | 0 | 5 | 0 |
| #instances | 10:16am | 6 | 0.97 | 3.1 | 0.789 | 0 | 6 | 1 |
| #instances | 10:21am | 6 | 0.004 | 3.2 | 0.013 | 0 | 6 | 0 |
|  |  |  |  |  |  |  |  |  |
| CPU % | 10:11am | 75% | 0.0 | 75 | -0.02 | 71 | 75 | 0 |
| CPU % | 10:16am | 89% | 2.0 | 75.02 | 1.003 | 71 | 89 | 14 |
| CPU % | 10:21am | 75% | -2.0 | 75.01 | 1.002 | 71 | 89 | -14 |
|  |  |  |  |  |  |  |  |  |
| ajax calls/s | 10:11am | 104 | 8 | 101.02 | 0.342 | 43 | 167 | 4 |
| ajax calls/s | 10:16am | 97 | -14 | 100.65 | 0.213 | 43 | 167 | -7 |
| ajax calls/s | 10:21am | 132 | 70 | 103.40 | 0.432 | 43 | 167 | 35 |

Each of the features are stored and persisted with the raw sample as they arrive, but can be recalculated from a stream of raw samples alone if necessary. Each of these features can be incrementally derived on the fly by looking at only the past few samples. And therefore does not need to scan all the data again or utilize an offline batch process to generate the necessary results. This type of design is sometimes called a [“streaming algorithm”](http://en.wikipedia.org/wiki/Streaming_algorithm)[[2]](#footnote-2).

## Structure of the Data

In addition to the features being stored for each individual sample, there are samples taken as roll-ups for time periods of various levels of granularity: hourly, daily, weekly, monthly, etc. This allows features of interest to be detected and reported to users if they manifest themselves at any of these levels. The same algorithms for detecting and reporting features on a per-sample basis can be reused at a higher granularity such as weekly or monthly by simply treating those higher level time scales as samples.



The above diagram is a way to visualize the structure of the feature data being collected. For each event stream listed vertically, the features described above would be populated horizontally in the columns. Each item along the third t-axis (into the page) contains the same information as the table of the first two dimensions, but for previous samples going back in time.

The size complexity of the data collected is S x F x T where S is the number of event streams being sampled, F is the number of derived features (7 in the example above) and T is the number of samples collected for each stream. Dividing T by the number of samples per unit time yields a figure for the amount of data generated per unit time.

The T dimension allows for a segregation of data to be dispatched to offline processing tasks in the background such as map/reduce jobs distributed to a cluster of nodes.

## 

## Algorithms

This section gives a rough overview of the streaming algorithm for storing the samples, calculating their features, and detecting other features of interest that should be reported to the user.

As the Nodeable system receives event data pushed into it by various worker components that poll external interfaces such as Github and AWS, it persists the events into the database along with the newly calculated features relevant to the new events. (This task can be deferred to other worker components that generate features on already saved events).

A function F is responsible for deriving the features to be persisted. For it to be a true streaming algorithm, F only has access to the new sample “e”, and an in-memory set of parameters “S” that encode the state of all the previous samples it has seen for the stream so far. F will also update that state to Snew reflect the newly absorbed sample. This means:

(data to insert, Snew) = F(e, S)

The function F needs to do the following for the specified feature:

* **y** - the sample itself, just return it, store y in S as yLast
* **dy/dt** - a discrete first derivative which is simply (y - yLast / Δt) where Δt is the difference in timestamps between the current sample and yLast’s. The values yLast and the timestamp of yLast are fetched from S (or Δt might be a fixed interval for this particular flavor of event stream).
* **avg(y)** - the average of y calculated incrementally:

e = y

avg(y) = S.yAvgLast + ((e - S.yAvgLast) /(S.n+1))

S.yAvgLast = avg(y)

// S.n = S.n + 1 - do this after the avg(dy/dt) calculation

* **avg(dy/dt)** - the average of dy/dt calculated incrementally:

avg(dy/dt) = S.yPrimeAvgLast + ((dy/dt - S.yPrimeAvgLast) /(S.n+1))

S.yPrimeAvgLast = avg(dy/dt)

S.n = S.n + 1

* **diff** - a byproduct of the dy/dt calculation above
* **min** - the smallest numerical value that has appeared in this stream, S.min = min(e, S.min); return S.min
* **max** - the largest numerical value that has appeared in this stream, S.max = max(e, S.max); return S.max

So the values stored in S for each stream (not for each sample!) “in memory” during runtime are:

* **n** - the total number of samples seen so far in this stream
* **yLast** - the last sample seen
* **yLastTimestamp** - the arrival time of the last sample seen
* **yAvgLast** - the last average calculated for y
* **yPrimeAvgLast** - the last average calculated for dy/dt
* **min** - smallest value seen in the stream so far
* **max** - largest value seen in the stream so far

The state S can be reconstructed from a reboot by looking only at the last sample’s features. Optimizations can be done to trade-off space for recovery time (and feature calculation math) by not storing the min/max/diffs and averages for each sample.

## Generation of Digest Messages

This section describes the policies for deciding what [kinds of messages](http://nodeable.jira.com/wiki/display/PRDs/Potential+Nodebelly+Messages)[[3]](#footnote-3) to generate and when to insert them into the Nodeable Insight stream.

### Summaries

As mentioned in the Introduction of this document, these messages summarize trends in raw events such as counts, averages, diffs, and max/min values.

### Anomalies

Any event that is statistically divergent from a norm or trend will be reported on via an anomaly message. Trends will be detected using low-pass filtering on abs(y) so that blips aren’t noticed. Then, sudden changes in y, dy/dt, or values for either outside of an expected range (possibly thresholded by user feedback) will be reported.

### Correlation

When a message of interest of any type is generated (aside from periodic ones) the sample at which point the message was generated is marked. Later on, an offline map/reduce process can perform distance calculations on the position of the marks relative to marks in the other event streams. The smallest distance between adjacent sets of marks of two or more streams signifies a correlation between events in the two streams.

### Causality

By modeling the delay of time between a correlated set of events between two streams and comparing that delay against what (a human curated) expected delay would be for each individual stream, the Nodeable system could detect and report the causality between correlated event sources instead of just the fact that they are correlated.

### Predictive Analysis

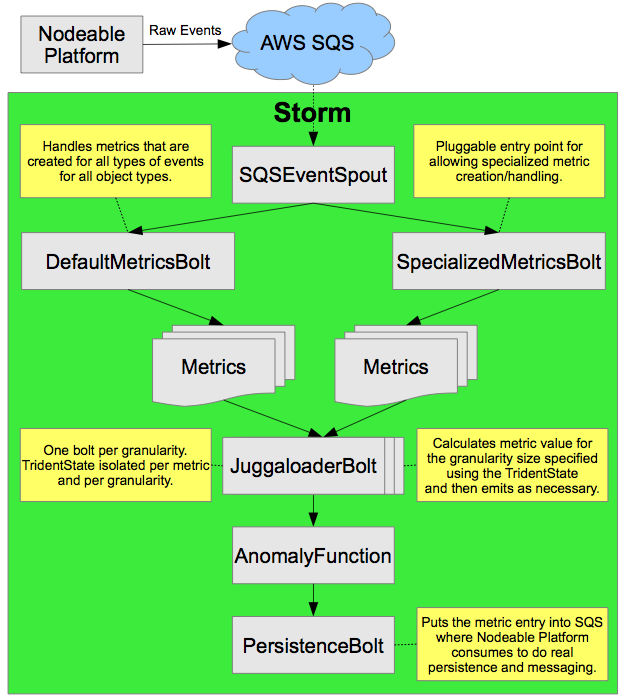
TODO

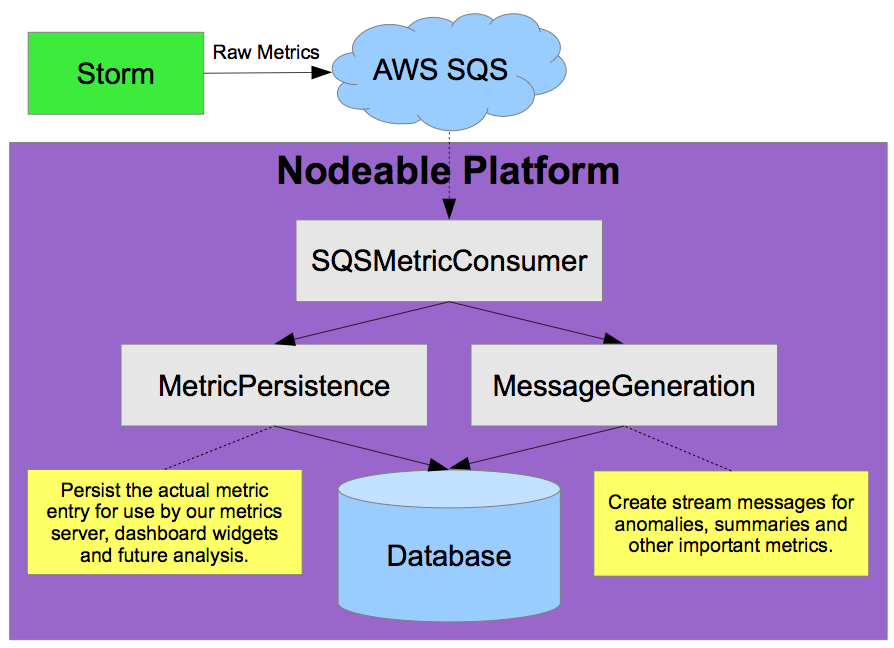
In the future, we plan to include user feedback signals as part of the Insight messages that are generated. Users can interact with current messages and use positive and negative signaling (e.g., up/down arrows, volume knobs against the standard deviations, range thresholds for y and dy/dt, or let us know when two events are actually correlated).

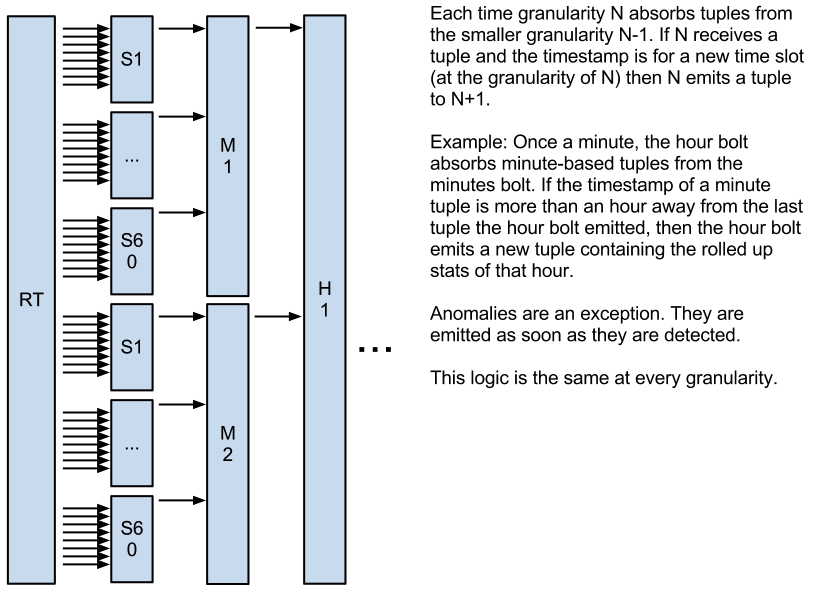
The metrics that the Nodeable app is currently tracking are listed under the [whitelist documentation](https://docs.google.com/a/nodeable.com/spreadsheet/ccc?key=0AgLFivNh9bBsdGNZcmZxWmdHdEdlRXpPbXBOMll4b3c#gid=6).

## Technology

Visual representations of the technology that powers Nodeable’s analytics are provided below:







# Appendix

## Terminology

* **Events / Event Streams / Samples** - a sequence of time series data of a particular type, usually numerical, that arrives into the Nodeable system incrementally. An example is the CPU utilization in percent of a particular instance on AWS. Usually maps to what we call a SobaObject in our shema but actually a single SobaObject could produce several event streams (CPU, Memory, etc from a single AWS instance)
* **Features** - simple numerical values derived from the raw event stream values. These are stored as samples arrive and are used to detect higher level features of interest to users and generate messages reporting them.

## Low-Level Design Notes

Proposed format of the eventStream Collection:

|  |  |
| --- | --- |
| \_id | the mongodb ID |
| type | raw, min, hour, day … year |
| ts | timestamp |
| value | the raw value (aka y from formulas above) |
| (features) | dy/dt, avg(y), avg(dy/dt), min, max, diff from above |
| mark | set when this event caused a statistically significant trigger/threshold to occur |
|  |  |

process current sample into updated state S and new feature value

emit

**Blurbs**

The Juggaloader is a generic real-time analytics engine used to filter raw input streams with lots of events into few messages of interest (aka “Nodeables”) that should be surfaced to users.

More signal less noise

* Input is raw event streams of (schemaless) numeric data
  + eg: CPU%, I/O latency, ping times, # of commits, # of lines of code changed
* Output snapshots of statistical features describing the streams
  + At various time scales
  + Messages injected into users’ streams describing interesting trends at a high-level

(show cube diagram here modelling the conceptual structure of the data)

Design Goals

* Simplicity - keep it simple initially, algorithms used allow for transparency and can answer questions about "what caused this message to appear in my stream"
* Generality - Detect a set of trends on data sources with as few assumptions about those sources as possible
* Continuous computation - Real-time stream algorithms instead of batch map-reduce tasks are needed so that messages can be surfaced to users in real-time
* Mechanism vs Policy - The Juggaloader implements the "how" using simple statistics on large sets of data, other components decide "what" / "when" to surface messages

Types of “Nodeables”

* Summaries - Periodic statuses and stream summaries over windows of time
* Trending - mean/min/max, first few derivatives, detect statistical anomalies / significant events
* Correlation - Compute aggregate distances between significant events across streams
* Causality - Which stream and event within it is the root cause of significant events in other streams?

(show the pyramid diagram here)

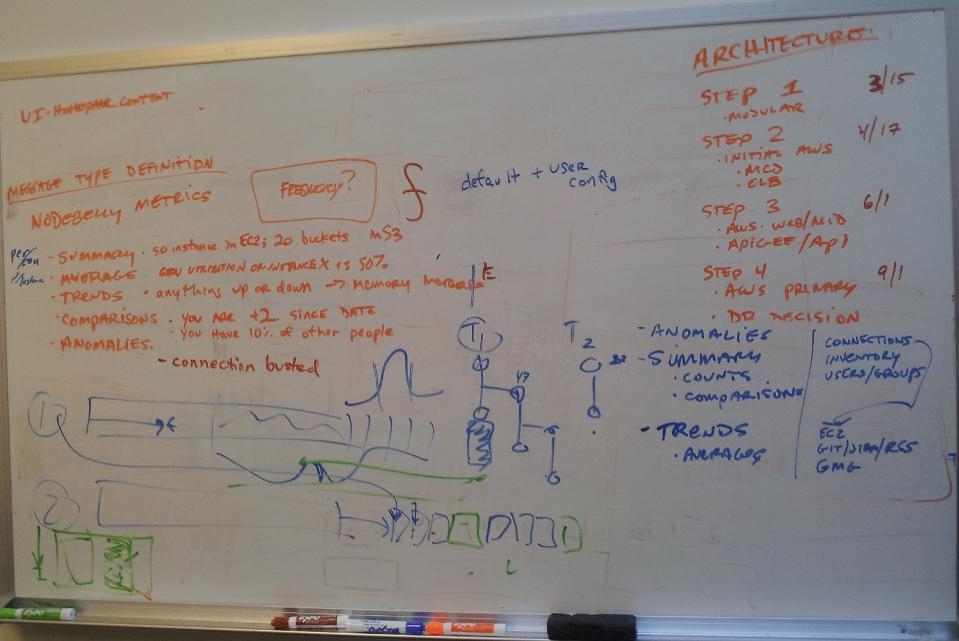
Later:

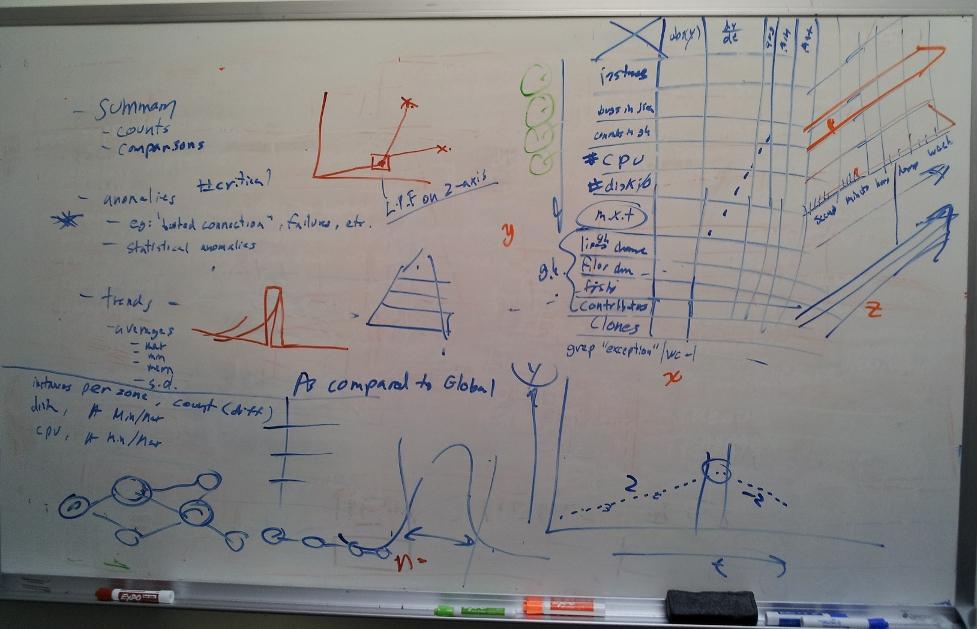
* Prediction - Use ML techniques to detect patterns that can predict future events with high probability
* Support for classifiers of non-numeric data instead of only numeric data streams

**Annoyances / Open Questions**

* can we get rid of type-specific K's (this is a pain for GMG data)
* Functions that continually increase should use the first derivative instead of standard deviations from the mean
  + Low pass filter
  + Ki +
* Take outliers out of the mean calculation so that they don’t move the average too high/low to get a real “normal” range
  + Conduct a check and flag data points that are 2-3 standard deviations away from the mean
  + Valid pattern recognition: detecting whether outliers are the new “normal” (based on how the future trending) or not
    - Ex: every Monday server1 always gets hit hard with requests
    - Ex: JIRA tickets in queue increase from 20 to 200 (after increase of support group) but now “normal” range of tickets in queue is around 200
  + Storm
    - Re: week/month/year.. each smaller-time bucket essentially summarizes a bunch of samples into a single summary which can be thought of as samples being fed to the next higher time-bucket
    - So as long as we get the right algorithms for one time frame, i believe they generalize to the rest (i hope)

**Whiteboard Photos**





TODO:

mechanism for comparison to global averages, “As compared to global”

feedback from user affects it how?

Do any mapreduces have to even happen since we keep running features?

scale by:

deferring feature generation to later after the event sample is persisted

offloading higher level timeframe stream analysis to other nodes

Existing Packages

TODO - research existing packages that might do this

**References**

[1] <http://en.wikipedia.org/wiki/Streaming_algorithm>

[2] <http://jvminside.blogspot.com/2010/01/incremental-average-calculation.html>

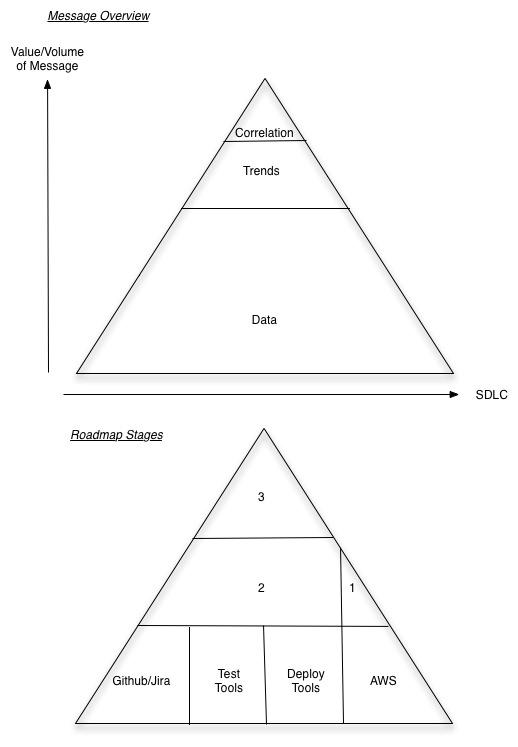
[3] <https://docs.google.com/a/nodeable.com/document/d/1Z1VQxLACdJPoYhC2OZ0P9MFxI7BPuFaVXafKKM9dq2Q/edit>

[4] <http://en.wikipedia.org/wiki/Juggalo>

[5] <https://nodeable.jira.com/browse/SOBA-1893>

[6] <https://nodeable.jira.com/wiki/display/SOBA/Metrics+Gathered+within+Nodeable>

[7] <https://nodeable.jira.com/wiki/display/SOBA/Juggaloader>



1. <https://docs.google.com/a/nodeable.com/document/d/1Z1VQxLACdJPoYhC2OZ0P9MFxI7BPuFaVXafKKM9dq2Q/edit> [↑](#footnote-ref-1)
2. <http://en.wikipedia.org/wiki/Streaming_algorithm> [↑](#footnote-ref-2)
3. <http://nodeable.jira.com/wiki/display/PRDs/Potential+Nodebelly+Messages> [↑](#footnote-ref-3)