320314

Achieving 99.99% uptime - a tale of Observability

Alexander Bicalho
Autodesk Forge – Run Platform Area Lead

Learning Objectives

- Observability concept and definition
- Describe and implement SLI, SLO and Error Budgets
- Describe and implement Synthetic Tests
- Describe and implement custom metrics, logging, tracing and analytics

Description

Observability is a measure of how well a system is behaving looking at its external outputs. At Autodesk we have a goal to deliver services with 99.99% uptime, and we have implemented an observability strategy that has been instrumental in achieving that goal. This class will walk you through the observability strategy we have implemented for the Forge services and will show you how to implement this strategy on your own web or cloud products and services to deliver high availability, and to quickly respond and remediate customer impacting issues. We will discuss the concepts of observability and how we implemented them with code snippets and examples. We will also cover standard observability concepts, like Service Level Agreements, Logging, Monitoring, and how you can put these concepts together to deliver a strategy that notifies your team of a service issue before your customers (internal or external) see any impact.

Speaker(s)

Alex has been working at Autodesk for 19 years. He worked as a QA Engineer in 3dsmax, AutoCAD and the Platforms Graphics SDK, later in the Cloud Platform team as a Test Engineering Manager, as an Infrastructure Engineering Manager as Product Manager for Observability, and most recently leading the Forge Run Platform Area in the Forge team. Alex believes in Automating all repetitive tasks; in building resilient, cost effective and scalable systems, and in delivering high quality services to our customers.

What is Observability?

In recent years, observability has become a prominent topic because our Cloud Applications have become more and more complex. Observability refers to the practice of using instrumentation to understand what's happening in software systems and reach conclusions about their states.

Observability Vs Monitoring

Observability uses monitoring, telemetry, analytics to reach conclusions about their state. If you consider an analogy, a telescope is a monitoring tool which allows you to 'observe' the stars. Likewise, viewing the stars sometimes is not enough, and using the telescope is not enough, for you to reach understanding about stars -> you may need other tools, like spectrographs, radio telescopes, and continuous monitoring in order to derive a complete understanding about the star being observed.

Cloud Systems Complexity

A few years back we only had Virtual machines, Load Balancers, a few storage systems for databases or files. Nowadays any cloud provider offers hundreds of different services, each with a different function, each working in a different manner to help us achieve our product goals. Such complexity means that we need to carefully define what we want to measure in order to be able to understand how our systems are behaving.
As examples, understanding the status or health of a Virtual Machine is different than understanding the health of a queue, or a function.

Cloud at Autodesk

Autodesk has been developing Cloud Applications for a few years now. Some of these services you know well, and you use them on a daily basis; others you don't even know about.
When we started with Buzzsaw there was no AWS. There were only datacenters back then. Over the years we've built new applications and services, and we have evolved together with the market. Today we have a few hundred cloud applications and services running in Windows, Linux, or serverless; written in multiple programming languages, using different database technologies.

Standardization

As we unified and standardized our practices, we rolled out a set of monitoring tools as 'standard' across a large number of services. These standards were targeted at measuring the infrastructure health, and monitored things like disk space, CPU usage, process running. This standardization helped unify our practices, but it created a side-effect: noise. It was common for these alerts to fire, a DevOps engineer gets paged to fix it, and by the time they get there the issue is resolved; or the issue is still there, but it's not urgent.
Standardization at scale came with a price -> we lost the ability to understand the intentionality of the measurements.

Tool Churn

The Cloud market has been evolving almost as fast as Moore law was changing the landscape of computers a few years back.
Here's a brief timeline of when some services were released:

Service	*Release date*
AWS Elastic Compute	August 2006 (limited)
Google App Engine	April 2008
AWS Relational Database Service	October 2009
Microsoft Azure	February 2010
AWS DynamoDB	January 2012
AWS Lambda	November 2014
Kubernetes 1.0	July 2015

Much like these services have changed and evolved over time, observability technologies have evolved. Internally we adopted new services and apps fairly frequently, always trying to chase the 'next big thing' or the holy grail.
This tool churn was damaging us more than it was helping us. It added to the noise and increased the complexity of the tools and systems. After a few years, we had multiple overlapping tools, and no solution in sight for our noise issues.

Observability Strategy

In early 2019 we formed an Observability team within Forge and gave this team the task to come up with a strategy around Observability with the following expected outcomes:

- Improve Signal to Noise ratio
- Reduce Alert Fatigue
- Distinguish between Detection and Diagnostics
- Measure end user Happiness
- Balance between Features and Reliability development

Based on these outcomes, the team created a High-Level strategy based on 3-lines of defense:

- SLI, SLO and Error Budgets
- Synthetic Tests
- Custom Metrics and Analytics

SLI, SLO and Error Budgets

Our first line of defense is focused exclusively on detection. It is aimed at measuring the customer's happiness by measuring a few indicators as close as possible to the customer, and setting appropriate objectives that will define a balance between happy and unhappy customer. The Error Budgets will determine how frequently and how much our customers are unhappy, thus defining when to alert.
This line of defense will indicate that 'something is broken' somewhere in our systems, but won't necessarily tell us what is broken, or why.

SLI (Service Level Indicator)

Service Level Indicators help us measure something quantifiable about a service that is provided. For instance, request latency is a common SLI, as it's measuring how long a request takes to be processed. Higher latency indicates that the customer needs to wait longer for a request, thus impacting that customer's experience.

SLO (Service Level Objective)

Service Level Objectives are target values for a service measured by an SLI. An example would be setting an SLO where the average requests per minute will be answered with a latency of 200ms or less.

Error Budget

Google describes Error Budget as "*how your business decides to trade off reliability work against other feature work when the SLO indicates a service is not reliable enough*". Mathematically speaking, it's a measure of how much you've breached your SLOs over time.
Another way to understand error budgets is by looking at the commonly defined SLAs – for instance, if a service has an SLA of 99.9% then the service can be down for 0.1% of the time. In 1 hour that's 3.6 seconds. In 1 month, that will be 43 minutes and 49.7 seconds.
Hint: https://uptime.is/ is a great tool to compute downtime budgets based on availability

Finding the right SLO is not a trivial task, but it's very important. It becomes an easier task by answering the question: "What value will make my customers unhappy?".

Synthetic Tests

Synthetic tests are testcases designed to simulate your user's interaction with the system. They can be API calls that replicate a given workflow, or tests that exercise the user interface to simulate a workflow.
These tests can be of different complexities – some of these tests, denoted Health checks are simply validating that your service exists and responds. Others can perform a complete user's workflow, like logging in, uploading a file, waiting for it to be processed and validating the output of processing.
Some synthetic tests are used as SLO, as they can be important to measure customer happiness.

Some caveats are the fact that Synthetic tests are expensive to run, they cannot run frequently, and they cannot truly represent your customer's experience.
As an example, imagine a cluster of 3 web servers where 1 web server is unhealthy. Your synthetic checks have 1 change in 3 to reach the failing web server. On the other hand, if you have a latency and availability SLO, you will see 33% of the calls failing, and you'll see a high consumption of Error Budget.

Custom Metrics and Analytics

There are many more interesting metrics you can collect from a running service that serve as the indicators which will help you better understand and diagnose issues with a service. These metrics are usually meant for business or debugging analysis and are usually not used for SLOs because they may be removed from the customer or may not be statistically accurate.

Custom metrics are also not used for alerting or paging, because while they are showing an issue, they are not specifically a symptom of a customer facing issue, and in many cases they're not symptoms of an issue at all.

For instance, a file translation service could measure queue, upload, download and processing times. While these are great metrics to have, upload, download and processing times will vary based on the complexity of the content, so deriving SLOs for them would be hard to do. Nonetheless, you would want to have those metrics, as you can correlate them with other facets to analyze when an issue happens. This is exactly why these metrics are targeted towards analytics, rather than SLOs and Error Budgets.

These metrics are also where you would put those standard infrastructure sensors – CPU, Memory, Disk space. You want to have them as Custom Metrics, collecting data and generating tickets into the system, which can be used if someone needs to debug or further analyze issues. You also want to make sure you are collecting these metrics intentionally, not 'by default', and make sure that you're looking at when and what matters.

Logging

Logs are a great way to collect data about a running service. It allows developers to generate data about running methods, functions, which are stored in a central location for searching.
Logging has several pros and cons, highlighted below:

Pros	*Cons*
Supports free-form text	Customer, Personal or sensitive information can be logged by accident
Supports unstructured format	Unstructured data is hard to search
Custom Metrics can be derived from logs	Higher cost compared to metrics solution
	High cost due to data volume

Given the higher cost of logging compared to metrics or other observability solutions, there are a few recommendations to make logging more cost effective:

- Use proper log level – DEBUG, INFO, WARN, ERROR
- Use a Feature Flag to switch Log level for a running service
- Use Structured Logging instead of free-form text
- Do not use Logging for Metrics, APM, etc. Use the metrics system for those tasks

Implementing SLI, SLO and Error Budgets

There are various types of applications and services. Each would require a different set of SLIs and SLOs. It's possible to group them and categorize them by class. For instance, a load balanced application or service would yield SLIs by analyzing its Load Balancer logs. A Data processing application would have metrics such as data quality or freshness.

Load Balancer based SLIs

Most of our applications are load balanced, have a load balancer or API Gateway which provides an entry point to the application. These load balancers are not Virtual Machines – they are infrastructure provided by the Cloud Providers and that means they usually have a very high availability and are the perfect entry point for us to measure our SLIs.

Our implementation is based on AWS load balancers. Load Balancer's logs provide logs in a space delimited format, and these are the first 12 fields:

```
timestamp elb client:port target:port request_processing_time
target_processing_time response_processing_time elb_status_code
target_status_code received_bytes sent_bytes  "request"
```

Here's an example log entry:

```
http 2018-07-02T22:23:00.186641Z app/my-loadbalancer/50dc6c495c0c9188
192.168.131.39:2817 10.0.0.1:80 0.000 0.001 0.000 200 200 34 366
"GET http://www.example.com:80/ HTTP/1.1" "curl/7.46.0" - -
arn:aws:elasticloadbalancing:us-east-2:123456789012:targetgroup/my-
targets/73e2d6bc24d8a067
"Root=1-58337262-36d228ad5d99923122bbe354" "-" "-"
0 2018-07-02T22:22:48.364000Z "forward" "-" "-"
```

With that in mind, it would be possible to use the ELB logs to gather statistics about the `elb_status_code` values, thus measure an SLI for failure rate. It would also be possible to gather statistics about the `request_processing_time` and measure the SLI for latency.

There are different ways to implement this process. One suggestion would be to use Cloud Watch Logs for ALBs and then parsing those logs to create metrics. Here's an example from AWS SDK Docs:

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/Counting404Responses.html

Load Balancer Logs in AWS are provided once a minute, so these logs and metrics are provided minute by minute.

Sample SLIs to measure

For Load Balancer SLIs it's important to measure the following metrics:

- Availability by Error Code – typically the ratio of HTTP 5xx divided by all of the requests
- Request Latency

It's very likely that your web service will offer multiple different API paths. There may be cases where you would want to omit a specific path from your metrics, or you will want to specify different objectives per path.

Here are some examples:

- /health should not be measured
- A GET api to retrieve a single object will have a different latency profile when compared to a GET api that retrieves a full page

In our implementation, we allow the developers to specify which paths we measure for SLI, and we then create metrics for each path, as well as an overall metric for the service.

Setting good SLOs

As mentioned previously, an SLO should be a measure of your customer happiness. If you consider the SLIs you are measuring for your load balancer, here are some example SLOs:

- 99.9% availability as measured by response error code
- A P99 of 300ms for the /login* API
- A P99 of 300ms for the /getitem API
- A P99 of 2000ms for the /getallitems API
- A P50 of 300ms for the /getallitems API

Note: P<nn> is a statistical measure that specifies that <nn> percent of the items will fall within the specified metric. Ex.: P50 of 300ms means that 50% of the calls are expected to be less than 300ms.

Hint: This article talks about setting good SLOs:
https://cloud.google.com/blog/products/gcp/building-good-slos-cre-life-lessons

Our metrics will yield a given number which is above or below the goals we have set. If the number is above our goal, then the goal was not met.

Calculating the Error Budget

In the example above, the metrics are collected once a minute. As the metrics are aggregated, they will yield either a pass or fail. If one metric fails, it will record a failure for that minute.

Error Budgets are measured over time and each different target will have a different budget. As mentioned earlier, a target of 99.9% availability yields a budget of 43 minutes and 49.7 seconds per month. If your application fails for 4 minutes consecutively, you have consumed 9% of your error budget for that month. If you measure Error Budgets weekly, then 4 minutes = 36.5% of your error budget.

Alerts based on Error Budget

The Google Site Reliability Workbook describes 4 key attributes to evaluate an alerting strategy:

1. Precision: The proportion of events detected that were significant. Basically, a signal to noise ratio.
2. Recall: The proportion of significant events detected. Detection rate.
3. Detection time: How long it takes to detect and notify
4. Reset time: How long after incident resolution does the alert stop firing. Long reset times increase the noise in the system.

Our recommendation is to setup alerting based on the following formula: `Target Error Rate >= Burn rate * 0.001 (SLO threshold)`

> Burn rate is how fast, relative to the SLO, the service consumes the error budget. If yours has a burn rate of 1, it means its consuming error budget at such a rate that you will be left with 0 budget at the end of SLO calculation window. Hence, for a service with 99.9% SLO, a constant error rate of 0.1% will result in burn rate of 1.

Logging

Logging is one of the three pillars of observability, along with *distributed tracing*, and *monitoring*. When it comes to reactive debugging in real time, logging is an invaluable tool.
It's fairly common to see logs evolving from `printf` that developers used when unit testing their applications locally. As they migrate to a production environment, printf is replaced by a logging call, and those logs are then sent to the server.

Logging levels

As traffic increases in the application, more and more logs will be generated. That often translates to difficulties in searching for the data. As more data is logged, the amount of data stored and indexed increases, making logging more expensive, and searches more time consuming.
Logging levels are meant to assist in separating the classes of logs. It's common to see 4 levels of logs:

- DEBUG
- INFO
- WARN
- ERROR

Debug logs are very verbose, and error logs should be rare, in comparison. In Production, your default log level should be WARN, meaning that INFO and DEBUG logs will be ignored, but WARN and ERROR will be processed.
Hint: Use Feature Flags to switch the log level of your application, making WARN the default log level, but allowing your team to switch to INFO or DEBUG temporarily for troubleshooting if needed.

What and How to Log?

When using Printf for debugging, it's fairly common for developers to log "Start", "Middle" and "End" actions. This allows them to see if the code reached a given line, and then to log the results/values of the actions.
When in Production, there are many servers and many threads running for each server, so the logs would look like this sequence: `start, start, start, end, middle, start, middle, end, start, middle, end, middle, end.`
As you can imagine, searching and identifying the proper log entries or level would be very hard.
Ideally, your logs should wrap an entire function, or a section of that function, and should log the entire function or section at once, not using a start/middle/end set of logs.

Structured Logging

Logs should be standardized and structured among all products and services. This allows you to search for HTTP_METHOD in any of your services, instead of searching for httpMethod in service A, http_method in service B.
Here's a good article about Structured Logging:
https://www.honeycomb.io/blog/structured-logging-and-your-team/

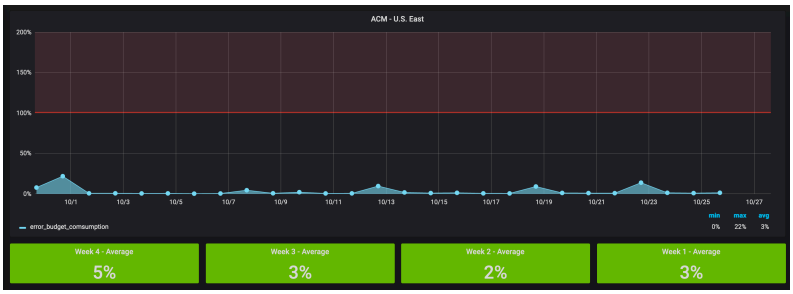
Hint: The 10 Commandments of Logging is a great article about how to deploy logging:
http://www.masterzen.fr/2013/01/13/the-10-commandments-of-logging/

Logging for Analytics

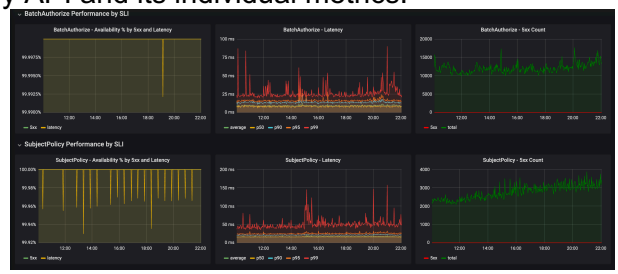
Since logs are structured text that can be searched, it's very common to see teams using logging for analytics and custom metrics. As an example, if you're logging the upload and download file size, then you can create searches and dashboards for those. While that works, that's a few orders of magnitude more expensive than simply using Custom Metrics – any kind will do.
First, these logs will be of type INFO, which should not be used frequently. Second, your metrics system is designed to store, display, dashboard those -> search is not designed for that.

Where we are in our Observability journey?

We have rolled out SLI, SLO and Error Budgets across all Forge Products and Services. Each team has setup their own SLOs and we are showing their 28-day error budgets in a large screen in the office. Here's a sample dashboard for one of our services:



The detailed Engineering dashboard shows every API and its individual metrics:



Deprecating old systems

As we onboard teams into the SLI/SLO dashboards, teams are also encouraged to review their existing monitoring strategies and migrate from the old systems into the modern SLI/SLO and Custom Metrics system.
This allows us to provide a better customer service, reduces our costs, and allows monitoring to be more intentional.

Logging Strategy

Our journey to standardize logging is starting this quarter, and we will be reconciling 2 different logging strategies into one, as well as creating a set of policies that define how logging should be used:

- Log Quota per service per day
- Mandatory Structured Logging
- Cost attribution

Bibliography

- Google SRE Handbook: https://landing.google.com/sre/workbook/toc/
- Rob Ewaschuk. My Philosophy on Alerting. Retrieved April 19, 2019, from http://files.catwell.info/misc/mirror/rob-ewaschuk-google-sre-philosophy-alerting.pdf
- Alexis Lê-Quôc, Monitoring 101: Alerting on what matters, Retrieved April 19, 2019, from https://www.datadoghq.com/blog/monitoring-101-alerting/
- Structured logging: https://www.honeycomb.io/blog/structured-logging-and-your-team/
- 10 Commandments of Logging: http://www.masterzen.fr/2013/01/13/the-10-commandments-of-logging/
- https://stackoverflow.com/questions/2031163/when-to-use-the-different-log-levels