Practical Monitoring

1. Monitoring Principles

1.1 Monitoring Anti-patterns

> An anti-pattern is something that looks like a good idea, but which backfires badly when applied.

Anti-Pattern #1: Tool Obsession

> When the mission is defined by running software, analysts become captive to the features and limitations of their tools. Analysts who think in terms of what they need in order to accomplish their mission will seek tools to meet those needs, and keep looking if their requirements aren't met.

Monitoring Is Multiple Complex Problems Under One Name You should use specific tools for specific purposes, not a generalist tool for specific purposes.

You should be using a configuration management tool. Agent-less monitoring is extremely inflexible.

You should use as few tools as needed to get the job done.

Adopting tools and procedures of more successful teams will not make you successful, as those tools haven't made them successful either. They use those tools and procedures because they are successful, not the other way around.

Chose your tools with care. Tools are created by teams with different goals in mind, be sure those tools fit your teams goals also.

Sometimes, you really have to build it Create small, specific tools, not big, generalist platforms.

Sometimes, the initial time invested in developing specialized tools for your specific purpose pays off in the future. An example of such a tool could be the creation of AWS EC2 instances with all your company standards automatically applied.

The Single Pane Of Glass Myth Having multiple tools feeding into a single dashboard can be ineffective. Having multiple tools feeding into their specific dashboards and then connecting the dashboards together might be a better solution.

Anti-pattern #2: Monitoring-as-a-job

Don't create specific jobs out of system monitoring. Monitoring is a skill all team members should have, and it's a task everyone in the team should participate in.

Anti-pattern #3: Checkbox Monitoring

Common anti-pattern smells:

- Recording metrics like system load, CPU usage, and memory utilization, but the service still goes down without your knowing why
- Constantly ignoring alerts, as they are false alarms more often that not.
- Checking system for metrics every 5 minutes or less
- Aren't storing historical metric data

How to remediate this:

What does "Working" really mean? Monitor that. Talk to the service app owner, to understand what "working" really means. For example, when monitoring a WebApp, check the `GET` response, latency and text to be displayed. If one of these three requirements fails, we know something has gone south.

OS metrics aren't really useful for alerting. Focus on what "working" means, instead of OS statistics. If a MySQL server is taking up all of the CPU usage, but response times are on point, then, there isn't really a problem.

Collect tour metrics more often. Monitoring often is preferred because in complex systems, a lot can change in a small period of time. Common, modern hardware can perform very well, even with more consecutive monitoring. Also, test the monitoring approach in a lab before deploying to prevent system failure due to monitoring.

Anti-pattern #4: Using Monitoring as a Crutch

Fix the problem rather than monitoring it. More monitoring doesn't fix a broken system, and it's not an improvement in your situation.

Anti-pattern #5: Manual Configuration

Automation is key. If you take a long time to add a new server to your monitoring solution, the most probable is that you will make a mistake of even don't add it at all. If, in the other hand, it takes only a couple of minutes, you can ensure the server has been correctly added to the monitoring stack and will add it.

1.2 Monitoring Design Patterns

Pattern #1: Composable Monitoring

> Use multiple specialized tools and couple them loosely together, forming a monitoring platform.

Composable monitoring can be thought of as the UNIX philosophy in action.

Components of a Monitoring Service Every Monitoring Service has to contain these 5 services.

1. Data collection
2. Data storage
3. Visualization
4. Analytics and reporting
5. Alerting

Data collection There are two models, push and pull.

Pull: A service requests that a remote node send data about itself. The central service is responsible for scheduling when those requests happen. A pull-based mechanism can be difficult to scale, as it requires central systems to keep track of all known clients, handle scheduling, and parse returning data.

Push: A client pushes data to another location. The client may do so at regular intervals or as events occur. Easier to scale in a distributed architecture, such as those in cloud environments. Nodes pushing data, only need to know where to send it.

We are interested in storing two types of data: *Metrics and logs*.

Metrics:

- Counter:

An event-increasing metric (Visitors on a website)

- Gauge:

A point-in-time value. You can store gauges and then plot their values in a graph, in order to obtain a reference in time.

Logs:

Logs are strings of text, with a timestamp, normally need parsing in order for humans to interpret the data in a meaningful manner. There are two types: *Unstructured* and *Structured*.

Unstructured:

Values are identified by order. If you are unfamiliar with the predefined order, you might have a hard time knowing what the values stand for.

```
192.34.63.77 - - [26/Jun/2016:14:06:22 -0400] "GET / HTTP/1.1"
301 184 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/47.0.2526.111 (StatusCake)" "-"
```

Structured:

Grouping of data in key-value pairs. An example of this is `JSON`. This is the same message as before, but converted to a `JSON` file.

```
{ "remote_addr": "192.34.63.77", "remote_user": "-", "time":
"2016-06-26T14:06:22-04:00", "request": "GET / HTTP/1.1", "status":
"301", "body_bytes_sent": "184", "http_referrer": "-", "http_user_agent":
"Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/47.0.2526.111 (StatusCake)", "http_x_forwarded_for":
"-" }
```

Log collection:

The easiest way of collecting logs would be to use **Log forwarding**. The OS sends logs to a remote place, instead of storing them in the local system.

Data storage Metrics are normally stored in a *Time Series Database* (TSDB). Each entry, called **datapoint** is stored in a key-value pair (The key, being the timestamp)

Many TSDB "roll up" or "age out". As data becomes old, it's resolution is compressed. For example: it might be unnecessary to store 60 second intervals of data that is 1 week old when monitoring the CPU load, so it's values are averaged or added up. Depending on the policy.

Log storage can be *Simple raw files* or *Search-engine* (Such as **elasticsearch**)

You want to use elasticsearch.

Visualization It is important to generate a frontend framework that suits your unique data visualization requirements.

The best dashboards focus on displaying the status of a single service or one product. The best dashboards are designed by people that understand the service or product.

Analytics and Reporting The most common use-case is determining and reporting on service-level availability (SLA) of your application and services.

The SLA formula is `a = uptime / total time`

Alerting

> Monitoring is for asking questions

Monitoring does not exist to generate alerts: Alerts are just one possible outcome.

Pattern #2: Monitoring from the user perspective

Even in complex systems, monitor from the user's perspective.

One of the most effective things to monitor is HTTP response codes. From there, expand to the underlying system.

Pattern #3: Buy, not build

3 groups of companies:

- Ones that use SaaS
- In-house monitoring with tools like Graphite, InfluxDB, Sensu and Prometheus
- Custom monitoring platforms (Companies like Facebook, Netflix, Twitter)

It's cheaper A great SaaS monitoring solution will cost around $9.000/year, but by comparison, a custom monitoring solution might range about $35.000 in raw engineering time plus another &18.750 per year in maintenance.

You're probably not an expert at architecting these tools Even if you are, it's also not the best use of your time. SaaS solutions allow you to buy dedicated expertise thrown at a specific problem domain.

SaaS allows you to focus on your company's product And the time it takes to set up is far less than developing a in-house solution.

No, Really, SaaS is Actually Better The only two rational objections are:

- You actually have outgrown it
- Security/Compliance reasons

Pattern #4: Continual Improvement

You are not going to build a world-class monitoring service in a day, or two years. World class tools are created through iteration and complete re-architecturing or the product.

1.3 Alerts, On-Call, and incident Management

Monitoring

> The action of observing and checking the behaviors and outputs of a system and it's components over time.

Alerts is just one way to accomplish this goal.

One way of preventing false positives is to meter the rolling average, but if we do this, we loose granularity.

We also have to determine exactly what is it that we want alerting, because our attention is limited.

What makes a good alert?

If we understand alerts as a way of communicating a vital error that needs urgent action, there are 6 keys to building great alerts.

- Stop using e-mail for alerts
- Write runbooks
- Arbitrary static thresholds aren't the only way
- Delete and tune alerts
- Use maintenance periods
- Attempt self-healing first

Stop using e-mail for alerts

- Send Response/Action required immediately alerts to SMS, PagerDuty, or some high priority service.
- Awareness needed but no immediate action required alerts can go to an internal chat room.
- Record for historical/diagnostics purpose can go to a log file

Write runbooks A good Runbook answers several questions:

- What is this service, and what does it do?
- Who is responsible for it?
- What dependencies does it have?
- What does the infrastructure for it look like?
- What metrics and logs does it emit, and what do they mean?
- What alerts are set up for it and why?

Every alert would have a link to it's Runbook, so that the person responsible for answering to it can understand heat is going on.

Arbitrary static thresholds aren't the only way A static threshold is far too specific and won't tell you about outlier situations or events that you haven't specifically set a rule for. For example, disk usage going from 10% to 80% overnight.

The alternative is: Statistical/percent change analysis, with tools such as Graphite. You can use statistical models like moving averages, confidence bands and standard deviation.

Delete and tune alerts Having too many alerts in the system causes alert fatigue. This, in turn causes you to be desensitized to alerts.

The solution:

- Do all alerts require someone to act?
- Are there alerts that can be deleted, re-structured or made more accurate?
- Can you automate to make the alert obsolete?

Use maintenance periods If you are working on the feature the alert is monitoring and your chances will trigger the alert, disable it, in order to reduce noise and save your peers some time.

Attempt automated self-healing first If the alert response is a well documented set of instructions to follow, automate them and let the computer do the work. If they don't work, then send the alert.

On-call

Is the person expected to take action when a alert is triggered. Be it the time it may.

Below are some ways to remediate on-call frustrations.

Fixing false alarms Strive for 100% alert accuracy.

Cutting down on needless firefighting Monitoring doesn't fix anything. You have to fix things after they break.

Two effective strategies:

1. Make the duty of on-call to work on system resilience and stability during their on-call shift when they aren't fighting fires.
2. Explicitly plan for system stability/resilience work.

Building a better on-call rotation Rotate on-call people instead of having always the same person.

The on-call handle period has to be inter-week because the person handing the on-call will have to tell the person getting the on-call the problems/state of the system and what has to be done.

Put software engineers on-call as well as operation engineers. This makes them empathy with operations engineers and incentives them to write better software.

Incident management

> An unplanned interruption to an IT service or reduction of quality of an IT service. **ITIL**

Establish a consistent method for detecting and responding to incidents. Such an example:

1. Incident identification (Monitoring identifies the problem)

2. Incident logging (Monitoring automatically opens a ticket for the problem)
3. Incident diagnosis, categorization, resolution, and closure
4. Communications throughout the event as necessary
5. After the incident is resolved, come up with remediation plans for building in more resiliency

For more serious incidents, a well-defined set of roles becomes crucial. Each of these rolls has a singular function and they should not be doing double duty.

- **Incident commander (IC)** This person's job is to make decisions. They oversee the outage investigation, and that is it.
- **Scribe** Writes down what is going on. Who is saying what and when. What decisions are being made and follow-up items identified.
- **Communication Liaison** Communicates status to stakeholders. They are the communication point between the people working on the incident and the people demanding to know what is going on.
- **Subject matter experts (SMEs)** Are the people actually working on the incident.

Postmortems

After a incident occurs, have a discussion about it (Who, What, When, Why, What). Do not enroll in blame culture, as people will feel compelled to cover up problem areas.

1.4 Statistics Primer

2. Monitoring Tactics