Observable Python Applications

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)
Ancestral homeland of the Ramaytush Ohlone

0.1 Introduction to Observability

0.1.1 What is observability?

Our applications execute a lot of code, in a way that is invisible. Is this code working? Is it working well? Who is using it? How?

Observability is the ability to look at data that tells you what your code is doing. Mostly, in this context, the main problem area is server code in distributed systems.

It is not that observability is not important for clients: just that clients tend not to be written in Python. It's not that observability does not matter for, say, data science, it is that the tooling for observability there (mostly Juptyter and quick feedback) are different.

What is observability

It's 5pm, do you know where your application is?

0.1.2 Why does observability matter?

So why does observability matter? Observability is a key part of software development life cycle.

Shipping an application is not the end, it is the beginning of a new cycle. The first step is to know the new version is running well. Otherwise, a rollback is probably needed.

Then, you need to know what is going on to know what to work on next. Which features are working well? Which ones have subtle bugs?

Things fail in weird ways. Whether it is a natural disaster, a roll-out of underlying infrastructure, or an application getting into a strange state, things can fail at any time, for any reason.

Outside of the normal SDLC, you need to know that everything is still running. If it is not running, it is impportant to be able to know how it is failing.

Why observability

Ship it and forget it?

0.1.3 Feedback

The first part of observability is getting *feedback*. When code gives information about what it is doing, this can help in many ways.

In a staging or testing environment, this helps find problems and, more importantly, triage them in a faster way. This improves the tooling and communication around the validation step.

When doing a canary deployment, or changing a feature flag, feedback is also important. This lets you know whether to continue, wait longer, or roll it back.

Feedback
Is my code doing what I think it does?

0.1.4 Monitor

Sometimes you suspect that something has gone wrong. Maybe a dependent service is having issues, or maybe Twitter is, um, a-Twitter with questions about your site.

Maybe there is a complicated operations in a related system, and you want to make sure your system is handling it well. In those case, you want to aggregate the data from your observability system into *dashboards*.

When writing the application, these dashboards need to be part of the design criteria. The only way they will have data to display is if the application shares it with them.

Monitor
What is going on right now?

0.1.5 Alert

Watching dashboards for more than 15 minutes at a time is like watching paint dry. No human should be subjected to this.

For this, we have alerting systems. Alerting systems compare the observability data to the expected data, and send a notification when it is not.

Fully delving into incident management is beyond the scope. However, observable applications are alert-friendly in two ways:

- They produce enough data, with enough quality, that high quality alerts can be sent.
- The alert either has enough data, or the receiver can easily get the data, to help triage the source.

High quality alerts have three properties:

- Low false alarms: if the alert fires, there is a problem.
- Low missing alarms: the alert fires whenever there is a problem.
- Timeley: The alert is sent quickly to minimize time to recovery.

These three properties are in a three-way conflict. You can reduce false alarms by raising the threshold of detection, at the cost of increasing missing alarms. You can reduce missing alarms by lowering the threshold of detection, at the cost of increasing false alarms. You can reduce both false alarms and missing alarms by collecting more data, at the cost of timeliness.

Improving all three parameters is harder to do. This is where the quality of observability data comes in. Higher quality data can reduce all three.

Alert

Is there a problem?

0.2 Logging

0.2.1 Intro to logging

Some people like to make fun of print-based debugging. But in a world where most software runs on not-your-local-PC, print debugging is all you can do.

Logging is a formalization of print debugging. The Python logging library, for all of its faults, allows standardized logging.

Most importantly, it means you can *log from libraries*. The application is responsible for configuring which logs go where.

Ironically, after many years where applications were literally responsible for that, this is less and less true. Modern applications in a modern container orchestration environment will log to stdandard error and standard output, and trust the orchestration system to properly manage the log.

However, you should not rely on it in libraries – or pretty much anywhere. If you want to let the operator know what is going on, using logging, not print.

logging

A print for the modern world

0.2.2 Logging levels

One of the most important features of logging is *logging levels*. Logging levels allow you to filter and route logs appropriately.

But this can only be done if logging levels are *consistent*. At the very least, you should make them consistent across your applications.

logging levels

What should go where?

Consistent semantics

With a little help, libraries that choose incompatible semantics can be retroactively fixed by appropriate configuration at the application level. This is done by using the most important universal convention in Python: using the getLogger(__name__).

Most reasonable libraries follow this convention. Filters can modify logging objects in place before they are emitted. You can attach a filter to the handler that will modify the messages based on the name to have appropriate levels.

Fixing logging levels

Filter

name

```
import logging

LOGGER=logging.getLogger(__name__)
```

With this in mind, you now how to actually specify semantics for logging levels. There are a lot of options, but the following is my favorite:

- Error: this should send an immediate alert. The application is in a state that requires operator attention. (This means that Critical and Error are folded)
- Warning: "Soft" alert. I like to call those "Business hours alerts". Someone should look at this within one business day.
- Info: This is emitted during normal flow. It is designed to help people understand what the application is doing if they already suspect a problem.
- Debug: This is not emitted in the production environment by default. It might or might not be emitted in development or staging, and can possibly be turned on explicitly in production if more information is needed.

logging level semantics

- Error: Alert now
- Warning: Alert in business hours
- Info: In Prod
- Debug: Staging/Explicit

In no case should you emit PII (personally identifying information) or passwords into logs. This is true regardless of levels.

Levels change, debug levels activated, etc. Logging aggregation systems are rarely PII-safe, especially with evolving PII regulation (HIPAA, GDPR, etc. etc.)

logging privacy

No PII

no passwords

regardless of levels

0.2.3 Logging aggregation

Modern systems are almost always distributed. Redundancy, scaling, and sometimes jurisdictional needs mean horizontal distribution. Microservices mean vertical distribution.

Logging into each machine to check the logs is no longer realistic. It is often a bad idea for proper control reasons: logging into a machine is too much.

All logs should be sent into an aggregator. There are commercial offerings, you can configure an ELK stack, or use any other database (SQL or no-SQL).

As a really low-tech solution, you can write the logs to files and ship them to an object storage. There are too many solutions to explain, but the most important thing is to *choose one* and aggregate everything to it.

logging aggregation
All instances → Centralized server
Query and Alert

0.2.4 Logging queries

After logging everything to one place, there are too many logs. The specific aggregator will define how to write queries, but whether it's grepping in an S3 bucket or writing NoSQL queries, logging queries to match source and details will be useful.

logging queries
Match
Source

0.3 Metric scraping

Metrics scraping is a server pull model. The metrics server connects to the application, periodically, and pulls the metrics. At the very least, this means the server needs connectivity and discovery for all relevant application servers.

Metrics scraping
Server pull model

0.3.1 Prometheus as a standard

The prometheus format as an endpoint is useful if your metrics aggregator *is* prometheus. But it is also useful if it is not! Almost all systems contain a compatiblity shim for prometheus endpoints.

Adding a prometheus shim to your application, using the client Python library, allows it to be scraped by most metrics aggregator.

Prometheus format

All common metrics aggregation systems support it

Prometheus expects to find, once it discovers the server, a metrics endpoint. This is often part of the application routing, often on /metrics. Regardless of the platform of the web application, if you can serve a custom byte stream with a custom content type at a given endpoint, you can be scraped by promtheus.

For the most popular framework, there will also be a middleware plugin or something equivalent which will automatically collects some metrics, like latency and error rates. This is not usually enough. You will want to collect custom application data: for example, cache hit/miss rates per endpoint, database latency, etc. etc.

Web endpoint

Integrate into web framework of choice

Use native library

0.3.2 Using counters

Prometheus supports several data types. One important, and subtle, type is the *counter*. Counters always advance – with one caveat.

When the application resets, the counter goes back to zero. These "epochs" in counters are managed by having the counter "creation time" sent as metadata. Prometheus will know not to compare counters from two different epochs.

Counters

Tick up or die

Hits

Bytes sent

0.3.3 Using gauges

Gauges are much simpler: they measure instantaneous values. Use them for measurements which can go up and down: for example, total allocated memory, size of cache, etc.

Gauges

Point in time measurement

Total allocated memory

0.3.4 Using enums

Enums are useful for states of the application as a whole, though they can be collected on a more granular basis. For example, if you are using a feature gating framework, a feature that can have several "states" ("in use", "disabled", "shadowing") might be useful to have as an enum.

Enums
Different states
0/1 mutually exclusive gauges

0.4 Analytics

Analytics are different from metrics in that they correspond to coherent "events". For example, in network servers, an "event" is one outside request and its resulting work. In particular, the analytics event cannot be sent until the event is finished.

An event contains specific measurements: e.g., latency, number and possibly details of resulting requests to other services, etc.

Analytics
Per-transaction measurements

0.4.1 OpenTelemetry: Strictly in the Future

A popular open source framework for collecting analytics is OpenTelemetry. It is the future: but only the future. At this point, major parts of OTEL are still "experimental" or "to be done". Putting it in production is somewhat fraught.

OpenTelemetry
Looks good...
but not there yet

0.4.2 Structured Logging

One current possible option is structured logging. The "send event" is just sending a log with a properly formatted payload. This data can be queried from the log aggregator, parsed, and ingested into an appropriate system for allowing visibility into it.

Structured Logging
Collect data in per-transaction object
Send it to log

0.5 Error tracking

You can use logs to track errors. You can use analytics to track errors. But a dedicated error system is worthwhile.

A system optimized for errors can afford to send more data, since errors are rare. It can send the right data. It can do smart things with the data.

Error tracking systems in Python usually hook into a generic "exception handler", collect data, and send it to a dedicated error aggregator.

Error Tracking
Detailed data about errors
Usually exceptions

0.5.1 Using Sentry

In many cases, running Sentry yourself is the right thing to do. When an error has occurred, something has gone wrong. Reliably removing sensitive data is not possible, since these are exactly the cases where the sensitive data might have ended up somewhere it shouldn't.

It is also, often, not a big load: exceptions are supposed to be rare. Finally, this is not a system that needs high-quality high reliability backups. Yesterday's errors are already fixed, hopefully, and if they are not – you'll know!

Sentry
For most non-trivial cases, run yourself:
Detailed error data can be sensitive!

0.6 Summary

0.6.1 Fast, Safe, Repeatable: Choose All Three

Observable systems are faster to develop, since they give you feedback. They are safer to run, since when they go wrong, they let you know sooner. Finally, observability lends itself to building repeatable processes around it, since there is a feedback loop.

Observability gives you knowledge about your application. And knowing is half the battle.

Data, Not Speculation
Observability → Knowledge

0.6.2 Upfront Investment Pays Off

Building all the observability layers is hard work. It also often feels like "wasted work", or at least like "nice to have".

Can you build it later? Maybe, but you shouldn't. Building it right will let you speed up the rest of development so much at all stages: testing, monitoring, and even on-boarding new people. In an industry with as much churn as tech, just reducing the overhead of on-boarding a new person is worth it.

Return on Investment

- Testing
- Monitoring
- On-boarding