Nice and naughty systems in production

Jan Macháček, Anirvan Chakraborty, Christian Villoslada

July 8, 2018

Abstract

This paper presents the result and analysis of a survey of developers in a large organisation. The aim of the survey was to find out what aspects of a software system contribute the most to the system's operation.

1 The first survey

The first survey asked *teams* to answer very specific questions about the code, testing, dependencies, deployment, and infrastructure. Across 31 teams, the average number of "fields" in the answers was 150. Listing 1 shows a small portion of the first survey; the answers were merged from pull requests to a single GitHub repository.

```
redundancy:
  is_load_balanced: true
  num_of_azs: 3
  num_of_regions: 3
...
testing:
  unit:
    is_used: true
    is_automated: true
    tracks_coverage: false
  And similar for integration, functional, performance, chaos.
...
```

Listing 1: Survey example

Anecdotally, the review process during the pull requests' merge process was helped the teams clarify their answers and improve the shared understanding of some of the answers: the teams now understand, for example, what constitutes good performance testing (read the value ***true*** in the response).

2 The second survey

The large number of questions, together with evolving understanding of the meaning of (some of) the answers meant that the data collected did not reveal any useful patterns.

Hence a second survey was planned. The survey asked *every engineer* fewer questions, specifically:

- what are 10 most important properties of reliable systems
- how do you rate your system (scale $(1; 10)$) in each of the features
- how easy is it for a new starter to be productive (scale $(1; 10)$)
- how "painful" is it to support the system (scale $(1; 10)$)
- if you could re-implement the system, how much would you change (scale $(1; 10)$)

Finally, the infrastructure team added their own 2 classes: *nice* and *naughty*. The classification selected 5 of the most frequent properties, and added properties from the first survey that were considered important. A parallel coordinates plot in Figure 1 of the collected data shows how the features **A-I** map to the final class in **J**.

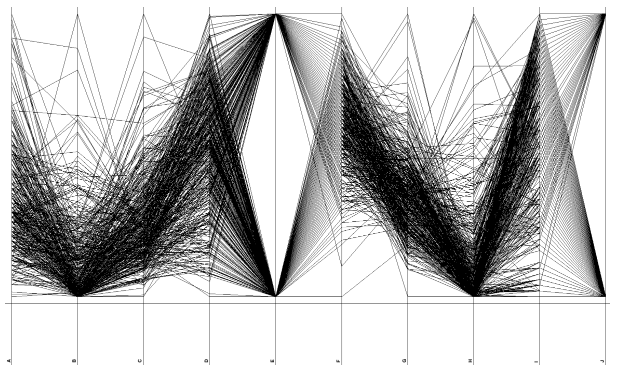

Figure 1: All responses

Colouring the *naughty* class blue and changing the opacity of the parallel coordinate lines to indicate the count of values reveals the properties that appear in the naughty class the most number of times. (See Figure 2 and Figure 3.)

3 The three properties

The top 3 properties of naughty systems are X, Y, and Z; the top 3 properties of nice systems are X', Y', and Z'!

4 Automated classification

The three properties can be used to build automated tools that can predict the type of the system during its continuous integration.

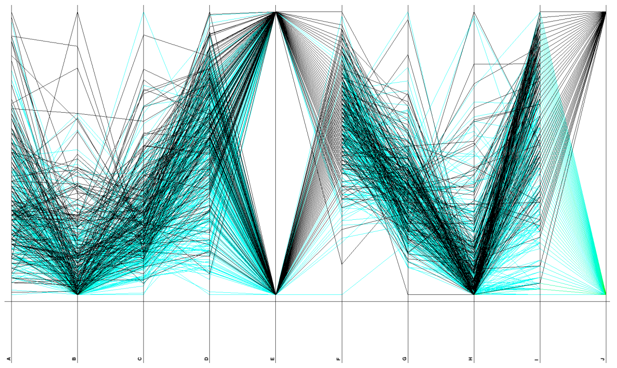

Figure 2: Highlighted naughty class

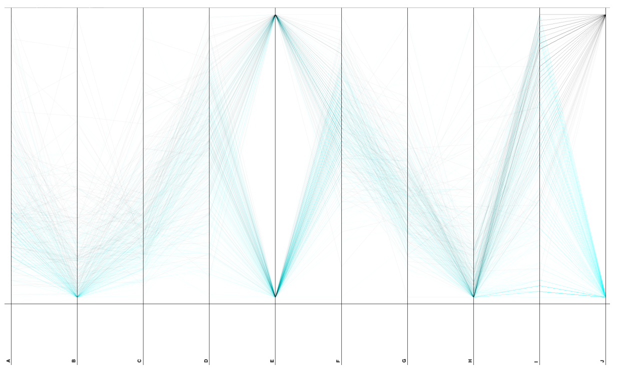

Figure 3: Contributions to the naughty class

5 Perfect vs. pragmatic

The survey analysis results provided 3 areas that need to be "perfect" if the goal is to achieve resilient and painless operation. The results also provide useful approach to the age-old debate of perfection vs. business value. The data from an incident repository show that sloppy code in these aspects result in

- missing early problem indications, resulting in service failures
- long time-to-diagnosis
- long time-to-root-cause
- dismissing some underlying problem as "one-off"

5.1 Back-pressure or circuit-breakers

XXX

5.2 Structured & centralised logging

Logging is one of the key tools that helps to locate the cause of a problem. I don't always do logging, but when I do, it looks like Listing 2.

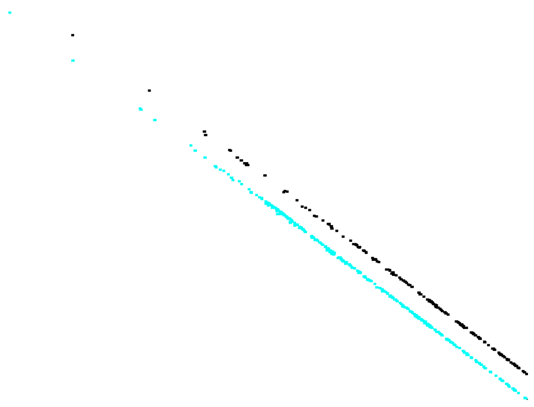

Figure 4: Separation along the first feature

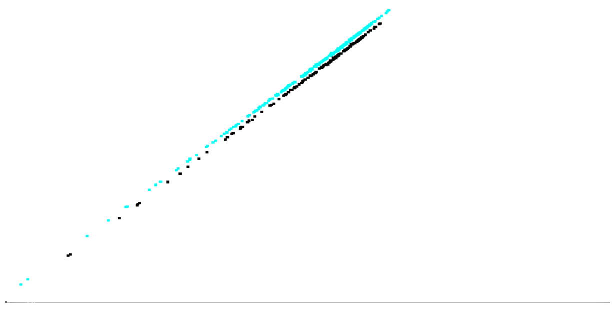

Figure 5: Separation along the second feature

```
def logClientError[A: IWrites](
  e: TranscodeClientError,
  job: A,
  blockedOnCall: Boolean)
  (implicit lc: LoggingContext,
            ls: LoggingScope):
  StateT[IO, AggregatorFailuresState[A], Unit] =
  StateT.liftF(
    IO(
      SLogging.logWarn(
        Seq[KV](
          failedSentsKey → e,
          jobKey → job,
          blockedOnCallKey → blockedOnCall
        )
      )
    )
  )
```

Listing 2: Structured logging

5.3 Performance testing

XXX

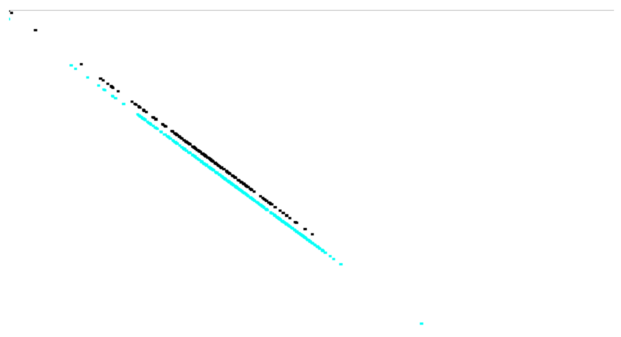

Figure 6: Separation along the third feature

6 Further work

Automation: is it possible to build a tool that automatically detects the proper application of the top 3 items? (Yes; one way is to use “internal OSS” that implements the top 3 and scan for its usage in the code, I’m sure there are other ways.)

Continued learning: once the top 3 have been resolved, other top 3 will emerge. It will be necessary to keep applying the approach in this paper forever.