Visualizing Evolving Service Performance in Python

NAMES ORDER TBA

Johann Bernoulli Institute of Mathematics and Computer Science
University of Groningen
Groningen, the Netherlands
Email: {v.andrikopoulos,m.f.lungu}@rug.nl, {t.klooster.1,p.p.vogel}@student.rug.nl

***Abstract*—The abstract goes here.**

I. Introduction

Every system is a distributed system nowadays []. ML: Vasilios: I can't find anymore the reference to the paper that you sent me a while ago about this topic :(Do you have it's bibtex?)

Python is currently one of the most popular programming languages. At the time of writing this paper[1] Python is the 4th most popular programming language cf. the Tiobe Index[2].

A search on GitHub with the keyword "language:Python" returns more than 500K open source projects written in the language. If we restrict the search by adding the keyword "Flask" we obtain a listing of 25K projects, that is, 5% of all the Python projects. Flask – advertised as a *micro-framework* – is a lightweight alternative to web site and service development.

However, there is no dedicated solution for monitoring the performance of Flask web-applications. Thus, every one of those Flask projects faces one of the following options when confronted with the need of gathering insight into the runtime behavior of their implemented service:

1) Use a heaviweight professional API monitoring setup they require setting up a different server ML: Thijs, Patrick: can we find a few examples of professional but overkill tools? ideally they require setting up a bunch of servers, and writing configs in XML! .
2) Implement their own analytics tool
3) Live wihout analytics insight into their services [3]

ML: For the first point in the list, we can also argue that analytics solutions like Google Analytics can be used, but they have no notion of versioning/integration with the development lifecycle. Feel free to cite [1] for service evolution purposes

For projects which are done on a budget (e.g. research projects) the first and the second options are often not available due to time and financial constraints.

To avoid these projects ending up in the third situation, in this paper we present a low-effort, lightweight service monitoring API for Flask and Python web-services.

[1]June 2017

[2]TIOBE programming community index is a measure of popularity of programming languages, created and maintained by the TIOBE Company based in Eindhoven, the Netherlands

[3]This is very real option: and is exactly what happened to the API that will be presented in this case study for many months.

Listing 1. TBA

```
import dashboard
dashboard.config.from_file('dashboard.cfg')
dashboard.bind(app=flask_app)
```

To start using our Python library for service visualization solution one needs to add exactly three lines of code that connect their Flask application object with the dashboard:

ML: Thijs, Patrick: Can we do with only these lines and w/o a config? Then we'll introduce the extra config later, as it becomes needed

ML: Thijs, Patrick: a small description of how the dashboard automatically intercepts the calls to the various API calls

ML: Thijs, Patrick: metion also that the dashboard automatically is available at /dashboard endpoint

ML: Thijs, Patrick: a small screenshot of how the dashboard allows one to select the interesting

II. Case Study

Zeeguu case study description to be used as running example throughout the rest of the paper [2]

Architecture: series of web and mobile applications built aroud a core web service implemented in Python and Flask which provides:

- contextual translations
- reading recommendations
- exercises

We have this system for helping learners read texts that they like, and enable them to practice with exericses generated on their past readings.

ML: we should consider adding also one section in which the architecture/implementation and main features of the dashboard are presented before going on with discussing them in more depth in the following sections — this should include a rundown on which views are provided from where (overview or per endpoint)

In the remainder of the article:

- all the views are screenshots from the actual tool; in the tool they are interactive with the user being able to zoom in, pan, etc.

III. Overall Endpoint Utilization

The most fundamental insight that a service maintainer needs regards service utilization.

Figure 1 shows a first perspective on endpoint utilization that Dashboard provides: a stacked bar chart of the number of hits to various endpoints grouped by day [4] shows that at its peak the API has about 2500 hits per day. The way users interact with the platform can also be inferred since the endpoints are indicators of different activity types, e.g.:

1) **api.get_possible_translations** is an indicator of the amount of reading the users are doing
2) **api.report_exercise_outcome** is an indicator of the amount of vocabulary practice the users are doing

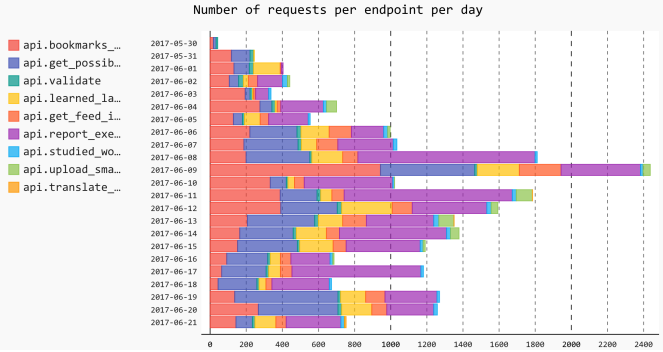

Fig. 1. The number of requests per endpoint per day view shows the overall utilization of the monitored application

This visualization also provides feedback to the maintainer when deciding aboiut endpoint deprecation, the most elementary way of *understanding the needs of the downstream* [3]. In our case study, the maintainer decided to not remove several endpoints once they saw that, contrary to their expectations, they were being used.[5]

A second type of *utilization* question that an API maintainer can answer by using the Dashboard regards cyclic patterns of usage during various times of day.

Figure 2 shows the users of the Zeeguu API not practicing languages at night, but otherwise hitting the API around the clock with several hundred hits per hour.

[4]We recommend obtaining a color version of this paper for better readability

[5]Usage information can also be used to increase the confidence of the maintainer that a given endpoint is not used, although it is not a proof.

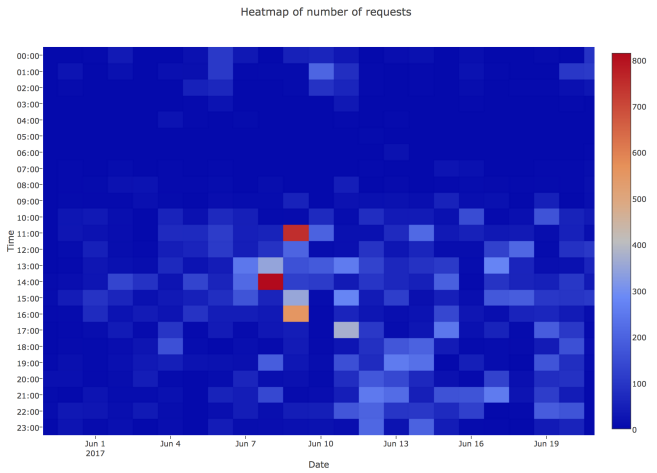

Fig. 2. Usage patterns become easy to spot in the requests per hour heatmap

IV. Visualizing Service Performance

The Dashboard also collects information regarding endpoint performance. The view in Figure 3 summarizes the response times for various endpoints by using boxplots.

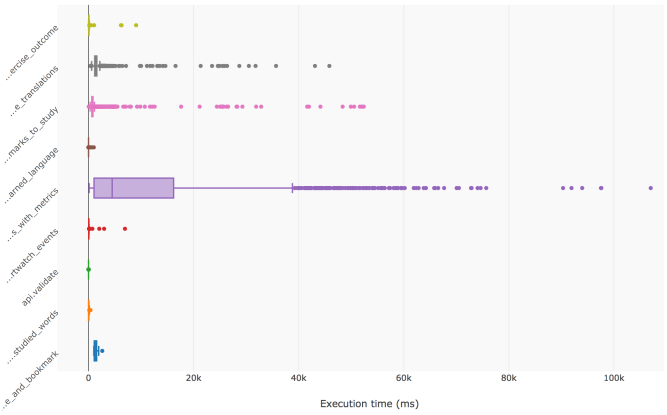

Fig. 3. The response time (in ms) per monitored endpoint view allows for identifying performance variability and balancing issues

After investigating this view it became clear to the maintainer that three of the endpoints had very large variation in performance. One of the three was most critical was optimized first: the **api.get_possible_translations** is part of a live interaction and it having such variable performance was a usability problem for the users of the reader applications.

To be able to see their improvements in action, the maintainer had to add an extra configuration line to the 'dashboard.cfg' file:

```
[dashboard]
GIT_ROOT=<path to .git folder of the app>
```

After redeploying the API, the dashboard can now automatically detect the current version of the project, and can group

measurements by this criterion. Based on this data, Dashboard can generate the view in Figure 4 where the performance of the give endpoint is tallied by version.

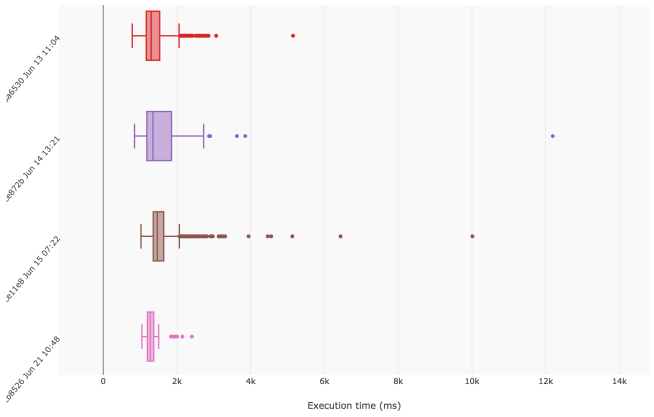

Fig. 4. Visualizing The Performance Evolution of the **api.get_possible_translations** endpoint

This way the maintainer could confirm that the performance of the translation endpoint improved: in the latest version (bottom-most box plot in Figure 4) we see the entire boxplot move to the left and there being less outliers.

The Dashboard also collects **extra informaiton about outliers**: Python stack trace, CPU load, request parameters, etc. in order to allow the maintainer to investigate the causes of these exceptionally slow response times.

In order to address this, but not degrade the usual performance the Dashboard tracks for every endpoint a running average value. When it detects that a given request is an outlier with respect to this past average running value, it triggers the *outlier data collection routine* which stores all the previously listed extra information about the current execution environment.

V. User Centered Visualization

For service endpoints which run computations in real time as they are called, there might be very different timings based on the different loads that are sent to the endpoint.

In our cases study, one of the slowest endpoints, and one with the higest variability is **api.get_article_difficulties**: it retrieves a list of recommended articles for a given user. However, since a user can be subscribed to anything from one to three dozen article sources, and since the computation of the difficulty is personalized and it is slow, the variability in time among users is likely to be very large.

Dashboard provides a way of grouping information on a per user basis. However, to do this, the developer must specify the way in which a given API call can be associated with a given user. There are multiple ways, the simplest takes again advantage of the strengths of the Flask framework which offers a global request object which contains session information:

Listing 2. TBA

```
# app specific way of extracting the user
# from a flask request object
def get_user_id(request):
  sid = int(request.args['session'])
  session = User.find_for_session(sid)
  return user_id

# attaching the get_user_id function
dashboard.config.get_group_by = get_user_id
```

Sometimes, grouping the service calls per endpoint it is not sufficient. Figure 5 shows some of the results of calling the **api.get_article_difficulties** endpoint for various users.

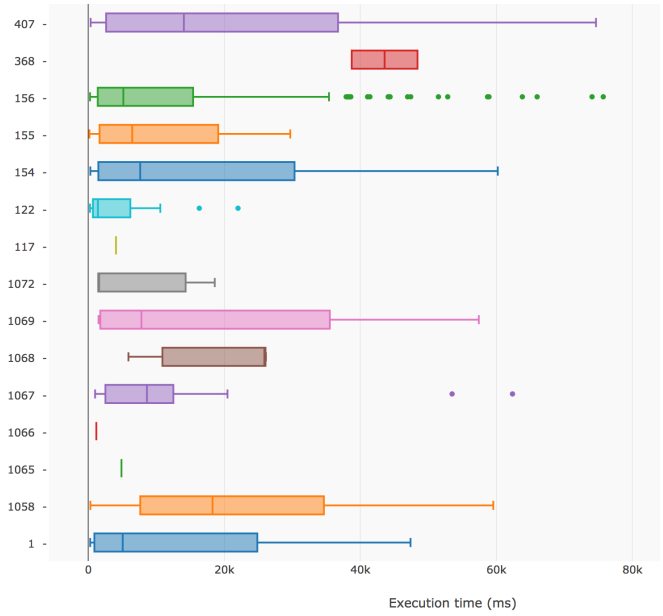

Fig. 5. The **api.get_article_difficulties** shows a very high variability across users

Different users might have different experiences: - a user has 10K emails one has 10 emails - a user is subscribed to 20 feeds one to 2 feeds

The system will have different processing times. It is important for the DevOps-er to be able to understand the difference in performance on a per user basis.

If we try to show also per version:

VI. Tool Availability

The code of Dashboard is available online at ...

The images in this paper are screenshots of the actual deployment of the tool which can be found at https://zeeguu.unibe.ch/api/dashboard. For the readers of this paper to be able to see the tool in action, they can login with the username and password: guest, dashboardguest!. ML: Thijs, Patrick: we should add a guest/guest username password which is allowed to only visualize things, but not modify anything, and not export anything!

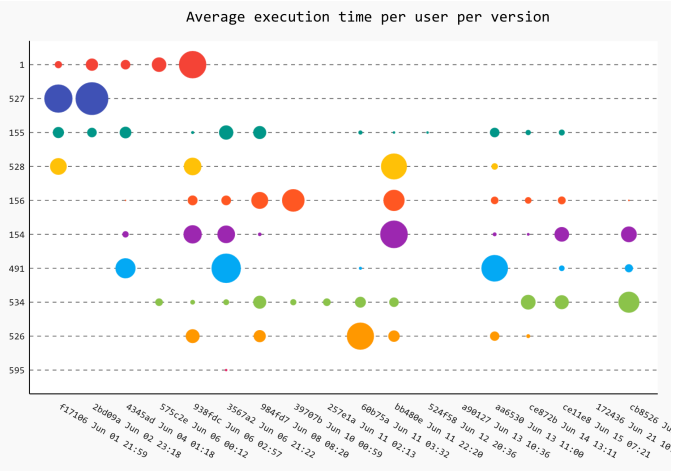

Fig. 6. Caption here

VII. Related Work

Java Visualization [4]

Run-time monitoring of services [5]

VIII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] M. Papazoglou, V. Andrikopoulos, and S. Benbernou, "Managing evolving services," *Software*, vol. 28, no. 3, pp. 49–55, 2011.

[2] M. F. Lungu, "Bootstrapping an ubiquitous monitoring ecosystem for accelerating vocabulary acquisition," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW 2016. New York, NY, USA: ACM, 2016, pp. 28:1–28:4. [Online]. Available: http://doi.acm.org/10.1145/2993412.3003389

[3] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A quantitative analysis of developer information needs in software ecosystems," in *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, 2014, pp. 1–6. [Online]. Available: http://scg.unibe.ch/archive/papers/Haen14a-QuantitativeEcosystemInformationNeeds.pdf

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2002, pp. 151–162.

[5] C. Ghezzi and S. Guinea, "Run-time monitoring in service-oriented architectures," in *Test and analysis of web services*. Springer, 2007, pp. 237–264.