Agile Monitoring of Evolving Web Services

The Author, The Other Author

Institute
University, Country
Email: {author}@university.domain

***Abstract*— Todo: Update abstract & title Tens of thousands of web applications are written in Flask, a Python-based web framework. Despite a rich ecosystem of extensions, there is none that supports the developer in gaining insight into the evolving performance of their service. In this paper, we introduce Flask Dashboard, a library that addresses this problem. We present the ease with which the library can be integrated in an already existing web application, discuss some of the visualization perspectives that the library provides and point to some future challenges for similar libraries.**

I. Introduction

Todo: Intro to be rewritten; refer to VISSOFT paper as previous work, explain focus is on integration with version control/CI for evolution support, and use of unit tests as early performance indicator

Todo: motivate with: monitoring in a world which is becoming more and more service-oriented is essential because it allows to perform three main types of actions: system adaptations to provide the request service at the desired level of quality, enabling flexibility in dealing with changing requirements and modes of operation, and enabling operational awareness through dashboards organizing the collected data [10]. In this work we focus on the last part and we discuss how a minimal effort probe-based solution for a particular type of web services can be used to facilitate the other two types of actions by involving the developer.

Todo: explain: in [13] we introduced the Flask Dashboard an a high level with a focus on presenting its performance visualization aspects; in this work we only summarize these features by means of introducing some of the provided functionalities of the Flask Dashboard. We focus on discussing how the tool is implemented and operating, provide a deeper look at its capabilities for integration with version control and continuous integration environments, introduce a mechanism for performance prediction, and provide an evaluation of the proposed approach based on a case study.

There is no getting around it: you are building a distributed system argues a recent article [1]. Indeed, even the simplest second-year student project is a web application implemented as two-tier architecture with a Javascript/HTML5 front-end a service backend, usually a REST API.

Python is one of the most popular programming language choices for implementing the back-end of web applications. GitHub contains more than 500K open source Python projects and the Tiobe Index[1] ranks Python as the 4th most popular programming language as of June 2016.

Within the Python community, Flask[2] is a very popular web framework[3]. It provides simplicity and flexibility by implementing a bare-minimum web server, and thus advertises as a micro-framework. The Flask tutorial shows how setting up a simple Flask *"Hello World"* web-service requires no more than 5 lines of Python code [11].

Despite their popularity, to the best of our knowledge, there is no simple solution for monitoring the evolving performance of Flask web applications. Thus, every one of the developers of these projects faces one of the following options when confronted with the need of gathering insight into the runtime behavior of their implemented services:

1) Use a commercial monitoring tool which treats the subject API as a black-box (e.g. Pingdom, Runscope).
2) Implement their own ad-hoc analytics solution, having to reinvent basic visualization and interaction strategies.
3) Live without analytics insight into their services.

For projects on a budget (e.g. research, startups) the first and the second options are often not available due to time and financial constraints. Even when using 3rd-party analytics solutions, a critical insight into the evolution of the exposed services of the web application, is missing because such solutions have no notion of versioning and no integration with the development life cycle. [8]

To avoid projects ending up in the third situation, that of living without analytics, in this paper we present Flask Dashboard — a low-effort service monitoring library for Flask-based Python web services that is easy to integrate and enables the *agile assessment* of service evolution. [7]

As a case study, on which we will illustrate our solution, we are going to use an open source API which, for several years, was in the third of the above-presented situations.

Todo: Add signposting paragraph

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

[2]http://flask.pocoo.org/

[3]More than 25K projects on GitHub (5% of all Python projects) are implemented with Flask (cf. a GitHub search for "language:Python Flask")

II. Case Study: The API (Zeeguu)

Zeeguu[4] is a platform and an ecosystem of applications for accelerating vocabulary acquisition in a foreign language [4]. The architecture of the ecosystem has at its core an API and a series of satellite applications that together offer to a learner three main inter-dependent features:

1) Reader applications that provide effortless translations for those texts which are too difficult for the readers.
2) Interactive exercises personally generated based on the preferences and past likes of the learner.
3) Article recommendations which are personalized for the interests of the learner and come with difficulty estimation that helps the learner find articles with the appropriate difficulty.

The core API implemented with Flask and Python provides correspondingly three types of functionality: contextual translations, article recommendations, and personalized exercise suggestions. In total the API provides a bit less than 50 endpoints, out of which probably a dozen are very frequently used.

At the time of writing this article, the ecosystem consists of a reader web application, a web based exercises platform, and a smartwatch application, which are used at the moment of writing this article by more than two hundred active users. The users come from a highschool, a language school, and some users are using it on their own, without any educational context. The highest load we observed until now on the API consisted of 12K requests in one day.

We will use this Zeeguu API as a case study for this paper. All the figures in this paper are captured from the actual deployment of Flask Dashboard in the context of the Zeeguu API. The figures are interactive offering basic data exploration capabilities: filter, zoom, and details on demand[12]. The Flask Dashboard deployment for the case study can be accessed publicly[5].

[4]https://github.com/zeeguu-ecosystem/

[5]https://zeeguu.unibe.ch/api/dashboard; username: *guest*, password: *soapsac*

III. Case Study: The Technology (Flask)

Flask is a microframework for Python. It IS USED BY THOUSANDS OF PROJECTS. ELABORATE MORE ON THIS...

Flask depends on two external libraries: the Jinja2 template engine and the Werkzeug WSGI toolkit.

Micro does not mean that your whole web application has to fit into a single Python file (although it certainly can), nor does it mean that Flask is lacking in functionality. The micro in microframework means Flask aims to keep the core simple but extensible. Flask wont make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you dont.

Flask has many configuration values, with sensible defaults, and a few conventions when getting started. By convention, templates and static files are stored in subdirectories within the applications Python source tree, with the names templates and static respectively. While this can be changed, you usually dont have to, especially when getting started.

A Python web application based on WSGI has to have one central callable object that implements the actual application. In Flask this is an instance of the Flask class. Each Flask application has to create an instance of this class itself and pass it the name of the module, but why cant Flask do that itself?

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello␣World!'
```

NOTHING PREVENTS GENERALIZATION

In this article we take as an example the Flask case study. But there is nothing that prevents the generalization of our approach.

MAKE SURE THAT THIS IS CLEAR.

IV. LOC #1: Service Utilization and Performance

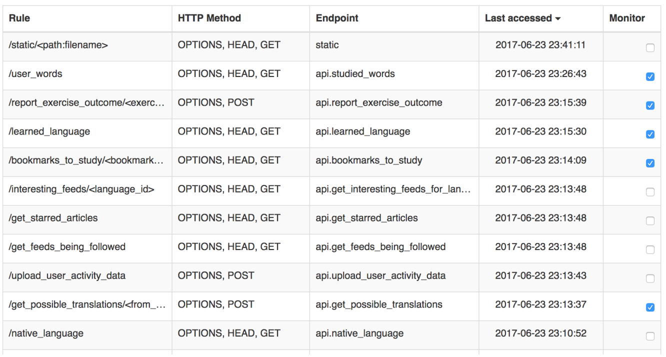
Since it is taylored for Flask applications, to deploy the Flask Dashboard one simply needs to use one line of code[6] to bind the dashboard to their Flask web application[7]:

```
import flask_monitoringdashboard as dashboard

# app is main application object
# every Flask web application has one
dashboard.bind(app)
```

During binding, the Flask Dashboard will search for all endpoints defined in the target application and add function wrappers around all the corresponding endpoint functions

The tool takes advantage of the fact that the monitored API already has a web presence, and makes available one extra endpoint (i.e. `/dashboard`), which serves the interactive visualization perspectives presented in the remainder of this paper.   The first perspective presents all the automatically discovered endpoints and lets the user select the ones that should be monitored. Fig. 13 shows that the last access time of every endpoint is tracked irrespective of whether it is selected by the user to be monitored or not.



Rule	HTTP Method	Endpoint	Last accessed ▾	Monitor
/static/<path:filename>	OPTIONS, HEAD, GET	static	2017-06-23 23:41:11	☐
/user_words	OPTIONS, HEAD, GET	api.studied_words	2017-06-23 23:26:43	☑
/report_exercise_outcome/<exerc…	OPTIONS, POST	api.report_exercise_outcome	2017-06-23 23:15:39	☑
/learned_language	OPTIONS, HEAD, GET	api.learned_language	2017-06-23 23:15:30	☑
/bookmarks_to_study/<bookmark…	OPTIONS, HEAD, GET	api.bookmarks_to_study	2017-06-23 23:14:09	☑
/interesting_feeds/<language_id>	OPTIONS, HEAD, GET	api.get_interesting_feeds_for_lan…	2017-06-23 23:13:48	☐
/get_starred_articles	OPTIONS, HEAD, GET	api.get_starred_articles	2017-06-23 23:13:48	☐
/get_feeds_being_followed	OPTIONS, HEAD, GET	api.get_feeds_being_followed	2017-06-23 23:13:48	☐
/upload_user_activity_data	OPTIONS, POST	api.upload_user_activity_data	2017-06-23 23:13:43	☐
/get_possible_translations/<from_…	OPTIONS, POST	api.get_possible_translations	2017-06-23 23:13:37	☑
/native_language	OPTIONS, HEAD, GET	api.native_language	2017-06-23 23:10:52	☐

Fig. 1: Once connected to an API the Dashboard presents the endpoints that are available for monitoring

We have decided for an opt-in approach to monitoring to avoid any the performance penalties incurred by the dashboard to affect performance sensitive endpoints. We discuss performance issues later. Also, we discuss later one situation in which opt-out might be desirable.

One alternative to allowing the user to use annotations would be to let them to annotate the code. However, this pollutes the code, and prevents deploying two versions which would monitor different endpoints.

The remainder of this section presents several of the interactive visualizations that become available without any further configuration[8].

[6]We don't count the import statement that enables that line...

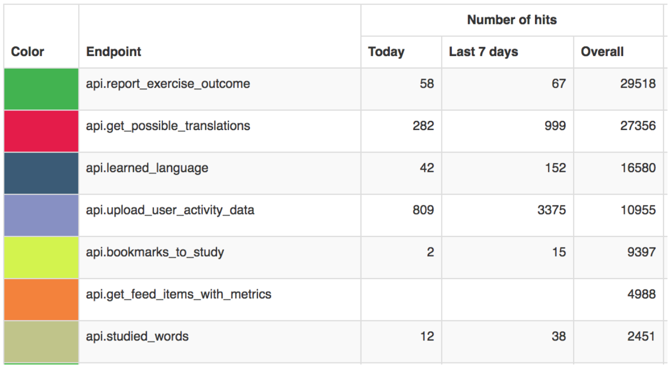
[7] In this paper we present the integration with APIs written in Python and Flask hoping that this will not prevent the reader from seeing the more general idea; all the tools we show here for Flask can be applied to other API technologies (e.g. Django) by simply providing a few back-end adapters in the right places.

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

A. Service Utilization

Knowing how third parties use one's API is difficult even in the case of static dependencies. In the case of services, there is no other way but monitoring service utilization. Flask Dashboard introduces a series of perspectives on utilization.

The most basic possible view shows the cummulative information about all the endpoints and the number of calls to that endpoint over the lifetime of its tracking as well as in the current day and the last seven days.



Color	Endpoint	Number of hits		
		Today	Last 7 days	Overall
🟩	api.report_exercise_outcome	58	67	29518
🟥	api.get_possible_translations	282	999	27356
🟦	api.learned_language	42	152	16580
🟪	api.upload_user_activity_data	809	3375	10955
🟨	api.bookmarks_to_study	2	15	9397
🟧	api.get_feed_items_with_metrics			4988
🟫	api.studied_words	12	38	2451

Fig. 2: ....

This kind of information is already very useful, and can provide the API maintainer with insight into the evolution of their system. By looking at Fig. 2, which presents the top 7 endpoints (for lack of space) one can already see several patterns:

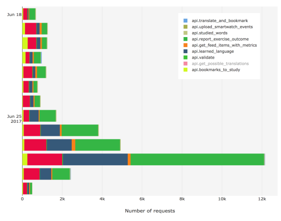
- **Frequent Patterns of System Usage**. The two most used endpoints stand for the two main activities in the system:
  - 🟥`api.get_possible_translations` is an indicator of the amount of foreign language reading the users are doing
    🟩`api.report_exercise_outcome` is an indicator of the amount of foreign vocabulary practice the users are doing.
- **Sudden Increase of Endpoint Usage**. One endpoint (🟪`api.upload_user_activity_data`) has been disproportionately been used in the last day and last week; much more than before.
- **Possibly Discontinued Endpoint Usage**. One endpoint (🟧`api.get_feed_items_with_metrics`) has not been used in the last 7 days

`Lesson:` : Although very useful, it was only after several months of using the system that the client requested the extra columsn for one day and seven days.

The table view is limited, and the one day and one week periods are conveniently but arbitrarily selected. For a more detailed evolution of utilization, Figure 3a shows the **`Daily Utilization`** perspective on endpoint utilization that Flask Dashboard provides: a stacked bar chart of the number of hits to various endpoints grouped by day[9]. Figure 3a in particular

[9]Endpoint colors are the same in different views

shows a peak utilization, a day when the API had more than 12.000 hits.


(a) Daily Utilization

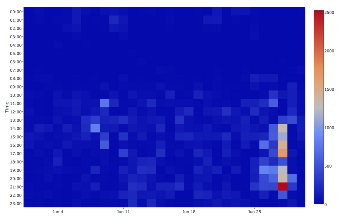

(b) Hourly Utilization

Fig. 3: Some of the available views

Another utilization perspective is the **`Hourly Utilization`** in which the Flask Dashboard can highlight *cyclic patterns of usage per hour of day* by means of a heatmap, as in Fig. 3b.

Figure 3b shows the API not being used during the early morning hours, with most of the activity focused around working hours and some light activity during the evening. This is consistent with the fact that the current users are all in the central European timezone. Also, the figure shows that the spike in utilization that was visible also in the previous graph happended in on afternoon/evening.

B. Service Performance

Knowing the performance of ones API is critical for the quality of a service API. Flask Dashboard introduces a series of perspectives on performance, which focus on the response times of the various endpoints.

HERE? ELSEWHERE? Actually, probably better for performance. The importance of the last day and last seven days became clear only after having used the dashboard for several months.

The Flask Dashboard also collects information regarding endpoint performance. The view in Fig. 4 summarizes the response times for various endpoints by using a box-and-whiskers plot.

In the Zeeguu case study, one of the slowest endpoints, and one with the highest variability as shown in Fig. 4 is 🟧`api.get_feed_items_with_metrics`: 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.

From this view it became clear to the maintainer that four of the endpoints had very large variation in performance. The most critical for the application and consequently the one optimized first was the 🟥`api.get_possible_translations` endpoint which was part of an interactive loop in the reader applications that relied on the Zeeguu API. Moreover, cf. Fig. 3a this endpoint is one of the most used in the system.

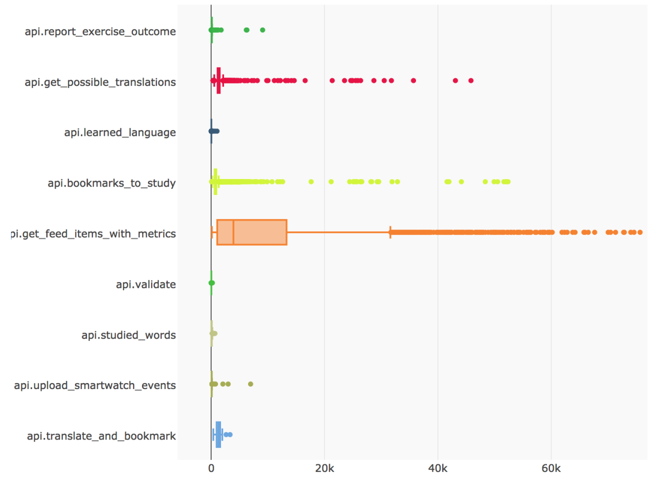


Fig. 4: The response time (in ms) per monitored endpoint view allows for identifying performance variability and balancing issues

C. Automated Outlier Detection and Monitoring

When an API is called from within a highly interactive application (as it is the case with the case study in this paper) of particular interest to the API developers are performance *outliers*.

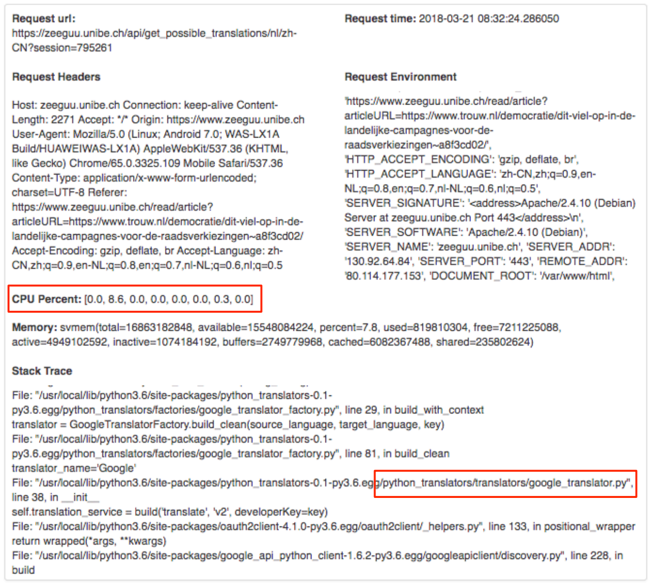
Indeed, a translation request that takes three times more than expected can seriously decrease the perceived quality of the application. Thus, identifying, collecting all appropriate data, and diagnosing the root causes of such outliers is especially critical in improving the quality of an application.

For this purpose the Flask Dashboard tracks for every monitored endpoint a *running average* response time value[10]. 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 extra information about the current execution environment. A configurable threshold with a default value of 2.5 times the running average response time is used for this purpose.

For every detected outlier request, the Flask Dashboard collects information about the current Python stack trace, CPU load, memory consumption, request parameters, etc. in order to allow the maintainer to investigate the causes of these exceptionally slow response times. In this way it is possible to get detailed insight into the operation of the application in the extreme cases without unnecessarily burdening it with logging this information for every request .

The bottom panel shows the stack trace. In this particular case, it is revealing for the developer to learn that at the time of the stack trace snapshot, the code was in the google_translator.py: indeed, the system uses as back-end multiple translators, and it has been observed that many of the outliers happen to be waiting in the google translator.

[10] For performance reasons, we assume that the response times for the endpoints are normally distributed. Otherwise, more general density distribution information must be collected in real time.



```
Request url:                                              Request time: 2018-03-21 08:32:24.286050
https://zeeguu.unibe.ch/api/get_possible_translations/nl/zh-
CN?session=795261

Request Headers                                           Request Environment

Host: zeeguu.unibe.ch Connection: keep-alive Content-     'https://www.zeeguu.unibe.ch/read/article?
Length: 2271 Accept: */* Origin: https://www.zeeguu.unibe.ch   articleURL=https://www.trouw.nl/democratie/dit-viel-op-in-de-
User-Agent: Mozilla/5.0 (Linux; Android 7.0; WAS-LX1A     landelijke-campagnes-voor-de-
Build/HUAWEIWAS-LX1A) AppleWebKit/537.36 (KHTML,          raadsverkiezingen~a8f3cd02/',
like Gecko) Chrome/65.0.3325.109 Mobile Safari/537.36     'HTTP_ACCEPT_ENCODING': 'gzip, deflate, br',
Content-Type: application/x-www-form-urlencoded;          'HTTP_ACCEPT_LANGUAGE': 'zh-CN,zh;q=0.9,en-
charset=UTF-8 Referer:                                    NL;q=0.8,en;q=0.7,nl-NL;q=0.6,nl;q=0.5',
https://www.zeeguu.unibe.ch/read/article?                 'SERVER_SIGNATURE': '<address>Apache/2.4.10 (Debian)
articleURL=https://www.trouw.nl/democratie/dit-viel-op-in-de-   Server at zeeguu.unibe.ch Port 443</address>\n',
landelijke-campagnes-voor-de-raadsverkiezingen~a8f3cd02/  'SERVER_SOFTWARE': 'Apache/2.4.10 (Debian)',
Accept-Encoding: gzip, deflate, br Accept-Language: zh-   'SERVER_NAME': 'zeeguu.unibe.ch', 'SERVER_ADDR':
CN,zh;q=0.9,en-NL;q=0.8,en;q=0.7,nl-NL;q=0.6,nl;q=0.5     '130.92.64.84', 'SERVER_PORT': '443', 'REMOTE_ADDR':
                                                          '80.114.177.153', 'DOCUMENT_ROOT': '/var/www/html',

CPU Percent: [0.0, 8.6, 0.0, 0.0, 0.0, 0.0, 0.3, 0.0]

Memory: svmem(total=16863182848, available=15548084224, percent=7.8, used=819810304, free=7211225088,
active=4949102592, inactive=1074184192, buffers=2749779968, cached=6082367488, shared=235802624)

Stack Trace

File: "/usr/local/lib/python3.6/site-packages/python_translators-0.1-
py3.6.egg/python_translators/factories/google_translator_factory.py", line 29, in build_with_context
translator = GoogleTranslatorFactory.build_clean(source_language, target_language, key)
File: "/usr/local/lib/python3.6/site-packages/python_translators-0.1-
py3.6.egg/python_translators/factories/google_translator_factory.py", line 81, in build_clean
translator_name='Google'
File: "/usr/local/lib/python3.6/site-packages/python_translators-0.1-py3.6.egg/python_translators/translators/google_translator.py",
line 38, in __init__
self.translation_service = build('translate', 'v2', developerKey=key)
File: "/usr/local/lib/python3.6/site-packages/oauth2client-4.1.0-py3.6.egg/oauth2client/_helpers.py", line 133, in positional_wrapper
return wrapped(*args, **kwargs)
File: "/usr/local/lib/python3.6/site-packages/google_api_python_client-1.6.2-py3.6.egg/googleapiclient/discovery.py", line 228, in
build
```

Fig. 5: Automatically collected outlier information

This information has to be corroborated with the observations that neither the memory nor the processor are overloaded at the moment. Thus this functionality in microsoft_translator is really slow in itself, and this is not a result of the machine being overloaded for example.

V. Grouping Requests

For service endpoints which run computations in real time, the maintainer of a system might want to understand the endpoint performance on a per-user basis, especially for situations where the system response time is a function of some individual user load[11]. **Todo: remove the configuration part and rephrase the following, the discussion has been partially subsumed by Section III.**

To enable this, the Flask Dashboard must be configured to associate an API call with a given user. The simplest way is to take advantage of the architecture of Flask applications in which a global `flask.request` object can be used to retrieve the session which can in turn lead to user identification:

```
# attaching a group by function
dashboard.config.group_by = 'User',
          lambda: Session.find(flask.request).userid
```

Todo: consider losing one or both of the figures — or at least updating them

In Zeeguu, 🟧`api.get_feed_items_with_metrics` retrieves a list of recommended articles for a given user. Cf. Fig. 4 it is the endpoint with the slowest response time and highest variability. The reason for this is that a user can be subscribed to anything from one to three dozen article sources and for each of the sources the system must compute the personalized difficulty of each article at every request.

[11]E.g. in GMail some users have two emails while other have twenty thousand and this induces different response times for different users

A **`Per-User Performance`** perspective should show the different response times for different users. Figure 6 presents a subset of the corresponding view in the Flask Dashboard. The figure shows that the response times for this endpoint can vary considerably for different users with some extreme cases where a user has to wait a full minute until their recommended articles are shown[12].

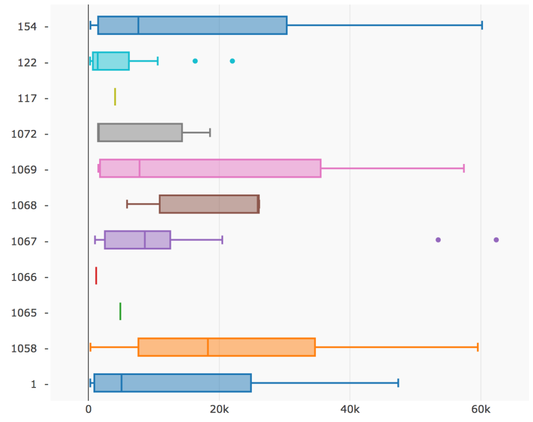


Fig. 6: The 🟧`api.get_feed_items_with_metrics` shows a very high variability across users

VI. Evolution Monitoring

When thinking about performance, one can not avoid thinking about the performance evolution.

Version control in Flask Dashboard can be supported in two ways, the developer explicitly states the current version, or the current version is automatically detected based on some version control system.

However, with the current configuration of the tool, it would be impossible for the maintainer to see the improvements resulting from the optimization.

The latest STack Overflow Developer Survey: 88.4% of **professional** developers use git. If assume that the code that they run is deployed using .git, then with an extra line of configuration they can allow Flask Dashboard to find the git[13] folder of the deployed service and automatically detect the version of the project that is running:

```
dashboard.config.git = 'path/to/.git'
```

If the Flask Dashboard can automatically detect the current version of the project by reading the .git configuration as soon as the API is started and can then group measurements by version[14].

Evolving Utilization: Fig. 7 shows the **`Multi-Version Utilization`** perspective. It shows the utilization of all the tracked endpoints across versions.

[12]After seeing this perspective, the maintainer refactored the architecture of the system to move part the difficulty computation out of the interactive loop

[13]https://git-scm.com/

[14]Alternatively, the maintainer can add version identifiers manually for the web application through a configuration file if the system does not use git.

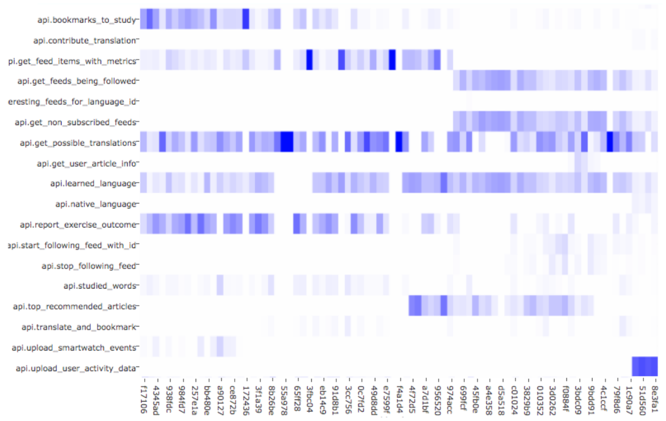


Fig. 7: The Evolution of Endpoint Utilization Across System Versions

Evolving Performance: Fig. 8 is a zoomed-in version of such a view for 🟥`api.get_possible_translations` with versions increasing from top to bottom

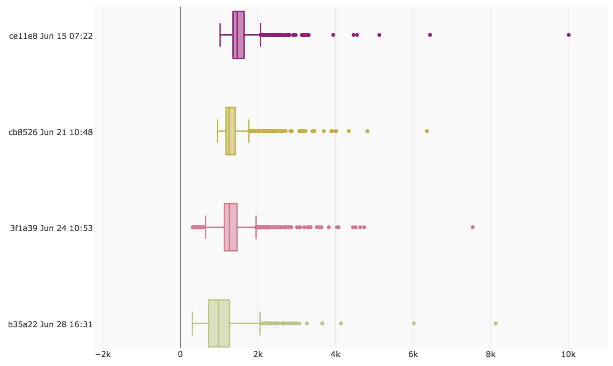


Fig. 8: The Performance Evolution of the 🟥`api.get_possible_translations` endpoint

This view confirms that the performance of the translation endpoint improved in the recent versions: the median of the last three versions is constantly moving towards the left, and progresses from 1.4 seconds (in the top-most box plot in Fig. 8) to 0.8 in the latest version (bottom-most box plot).

CHALLENGE: Supporting other types of version control... not real challenge, as we said 88percent of devs use git. CHALLENGE 2: detecting minor versions which don't need to be tracked, since they didn't touch the performance of the system. E.g. a modification of the README, etc.

Todo: TODO: endpoint evolution

Evolving Groups: The limitation of the previous view is that it does not present the information also on a per version basis. To address this, a different visual perspective entitled **`Multi-Version per-User Performance`** can be defined. Figure 9 presents such a perspective by mapping the average execution time for a given user (lines) and given version (columns) on the area of the corresponding circle.

The colors represent users. The figure shows average performance varying across users and versions with no clear trend:

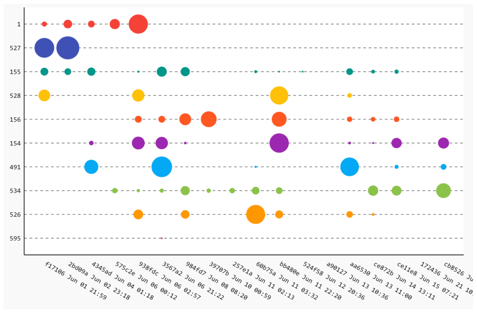


Fig. 9: This perspective shows that the evolution of response times for individual users (horizontal lines) across versions (the x-axis)

this is probably because varying user workload (i.e. number of sources to which the user is registered) is the reason for the variation in response times. one can see that for user 1 performance degrades over the

VII. Preemptive Monitoring

INTRO - LINKING WITH PREVIOUS PARAGRAPH: the previous evolutionary view crucial for understanding the impact of software evolution on performance. However, just by observing this one can't be sure whether performance changes because of the code or because of the payload.

Todo: the following might need a bit of rewrite

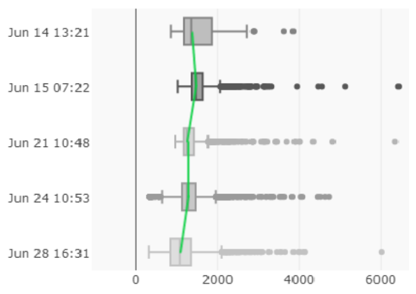
The concept of *preemptive monitoring* of the application performance by means of instrumenting integration unit testing as the synthetic load is similar to the idea of augmenting service monitoring with online testing [6], i.e. testing service-based applications by using dedicated test input in parallel to its normal use and operation. The difference is that we take advantage of the capability of the CI framework (i.e. Travis in this case) to create an emulated "live" environment for integration testing purposes, and use unit testing as the dedicated test input in order to measure performance.

For example, Fig. 11 is a screenshot from the dashboard showing the measured response times for 5 consequent builds, with 180 iterations of the unit tests executed in total across all endpoints. The outliers on the right of the figure are due to initial requests which must wait for a boot-up phase of the API .

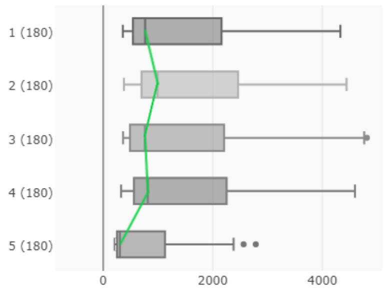
While this integration testing environment is different from the actual production deployment one, and the load used is purely synthetic, as will show in Section X, it can actually be used successfully as an early performance indicator for the application developer. Although predicting the performance of the application would require advanced statistical work, it can be succesfully used as an advanced warning system for the overall performance trend for given endpoints .

a) Integration with Travis: ML: added new section... also, this has to be moved after introducing the pre-emptive monitoring since it's about running tests. switched the order

The pre-emptive monitoring system can be configured to work together with Continuous Integration (CI) frameworks



(a) The reported response time (in ms) per deployed version in the observation period



(b) The measured response time (in ms) using integration with Travis

Fig. 10: Comparison of the response times per endpoint: actual production system versus preemptive monitoring data

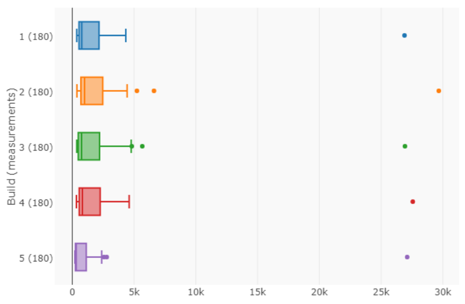


Fig. 11: Response times in 5 consequent Travis builds in the Zeeguu case study

like Travis[15] that deal with automated integration testing. The developer needs to define the unit test folder of the project, the URL that Travis should use for reporting its results, and the number of iterations for each unit test should be executed in order to generate a load for the application.

Each time a new build is created through Travis, the Flask Dashboard automatically detects all available unit tests defined by the application developer and iterates through each one of them the number of times predefined by the developer while monitoring the response times for each request. The resulting measurements are persisted in a separate part of the tool database so as not to contaminate the "live" monitoring data. Beyond seeing the results of each Travis build in the Flask Dashboard, this feature also allows for preemptive monitoring of the application, as we discuss in the following.

Besides functioning as an early warning system for performance, tracking the evolving performance of API tests serves a second purpose. To function as an anchor when the developer analyzes performance degradations. For example, when a developer sees that a given endpoint has become less performant after in a newly deployed version, they are able to investigate whether the performance degradation is visible also with the synthetic load. This would correspond to a performance degradation which is due to "algorithmic performance degradation". If on the other hand, the performance of the tests does not change between versions, the developer might conclude that the performance degradation could be due to the workload on the machines, or maybe to the workload mix of the users

Todo: here goes the discussion about our forecasting capabilities

Iteration	**Live (median)**	**Travis (median)**
1	1349.41	764.87
2	1466.13	992.87
3	1256.65	760.87
4	1266.42	813.89
5	1080.68	303.4

Pearson correlation $r(3) = .93, p = .02$

[15]https://travis-ci.org/

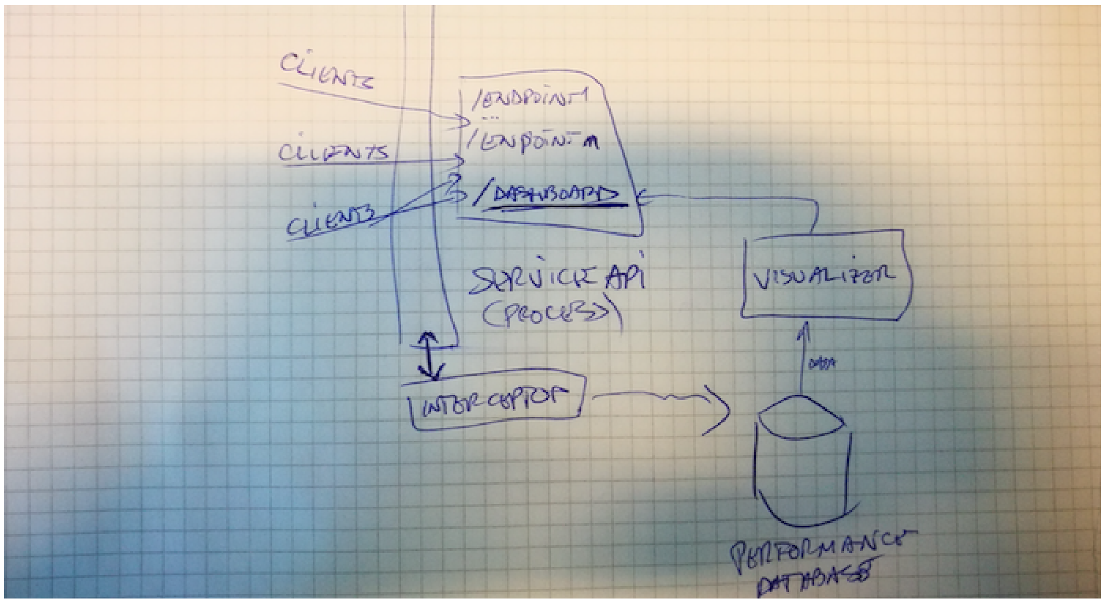


Fig. 12: The first thing that needs to be done, is to decorate the application object with an INTERCEPTOR

VIII. Architecture

REMEMBER THE GOAL: Our main goal with the Flask Dashboard issues to allow the integration of the dashboard with an API with minimal effort.

The first important question that such a service monitoring system should provide is information about what endpoints are being used and by whom .

TODO: Find references, arguments, related work that provide more details about why understanding who uses endpoints, and which are used are important.

This is clearly the case in static analysis (see paper by haenni and lungu...) but should even more be the case in APIs.

Data collected by the wrappers are persisted in a local database. The SQLAlchemy Object Relational Mapper[16] on top of SQLite[17] is used for this purpose.

TODO: TALK ABOUT THE METAMODEL...

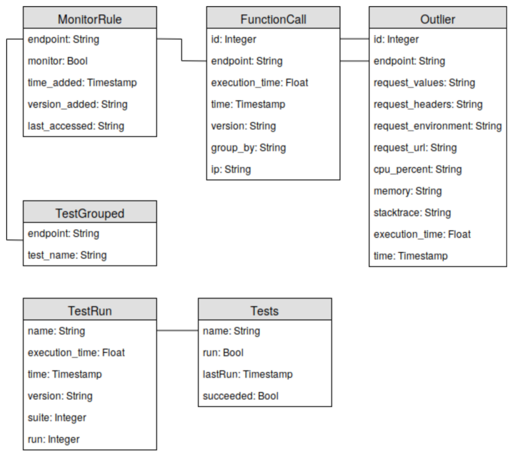


Fig. 13: The model that supports the views presented in this paper

As discussed in the introductory section, in previous work [13] we introduced Flask Dashboard, a drop-in Python library that allows developers to monitor their Flask-based Python web applications with minimal effort. The Flask Dashboard is implemented for Python 3.6 and is available on the Python Package Index repository[18] from where it can be installed by running `pip install`

[16]https://www.sqlalchemy.org/
[17]https://www.sqlite.org/
[18]https://pypi.python.org/pypi/flask-monitoring-dashboard/1.8

`flask_monitoring_dashboard` from the command line. The source code of the Flask Dashboard is published under a permissive MIT license and is available on GitHub[19].

IX. Performance Perspectives

PERFORMANCE DISCUSSION: this introspection, which looks up endpoints, happens only once at the startup of the service, so it is going to affect the API startup performance, but not the actual endpoint performance. In our case, the overhead here was quite small: TO MEASURE .

Todo: evaluate which figures to keep and ideally take new screenshots for the ones we keep; consider adding two more views (like Fig. 3.2 of Patrick's thesis)

X. Evaluation

XI. Discussion

- We can do this minimal configuration for any technology. We showed here how to do it for Python and Flask because they are some of the most popular web development technologies at the moment.

- The fact that the Flask Dashboard itself is being developed in Python using Flask makes binding to the services of a to-be monitored application developed with the same technologies easy and intuitive. However, by writing different backends that do the measurement and monitoring, the frontend can be reused.

- A Special Type of Outlier: Exceptions. TODO: probably something for the journal extension ... statistics about which endpoints fail most often might be useful too. same information as the outlier maybe?

Todo: use this subsection to discuss limitations (from the previous evaluation section) and add the need to be able to handle the horizontal scaling of the application; remove sectioning and summarize briefly limitations that have already been discussed in the VISSOFT paper/move material to other sections; consider renaming as limitations or something similar

A. Distributed System Monitoring

Todo: To think about Supporting situations where the API is deployed across multiple containers for example.

1) Automatically Monitoring System Evolution: The main goal of the Flask Dashboard design was to allow analytics to be collected and insight to be gained by making the smallest possible changes to a running API. This technique assumes that the web application code which is the target of the monitoring is deployed using `git` in the following way:

1) The deployment engineer pulls the latest version of the code from the integration server; this will result in a new commit being pointed at by the HEAD pointer.
2) The deployment engineer restarts the new version of the service. At this point, the Flask Dashboard detects that a new HEAD is present in the local code base and consequently starts associating all the new data points with this new commit[20].

[19]https://github.com/flask-dashboard

The advantage of this approach is the need for minimal configuration effort, as discussed in the presentation of the tool. The disadvantage is that it will consider on equal ground the smallest of commits, even one that modifies a comment, and the shortest lived of commits, e.g. a commit which was active only for a half an hour before a new version with a bug fix was deployed, with major and minor releases of the software. A mechanism to control which versions are important for monitoring purposes is therefore required to be added. A further possible extension point here is supporting other version control systems (e.g. Mercurial). However, this is a straightforward extension.

2) NEW: API Renames: ML: just an idea

The history would normally be lost in the case of an API rename. (also known as the *provenance issue* in source code evolution analysis) In theory we could infer that an API was renamed if it is not to be found anymore, and another one with very similar timing characteristics would be found. Such a detection can never be completely sure, but it would help with meeting the needs of the API maintainer...

3) User-Awareness : For the situations in which the user information is not available, the Flask Dashboard tracks by default information about different IPs and in some cases this might be a sufficiently good approximation of the user diversity and identity.

The visualizations for the user experiene perspectives as presented in Section V have been tested with several hundred users (of which about two hundred were active during the course of the study), but the scalability of the visualizations must be further investigated for web services with tens of thousands of users.

4) Other Possible Groupings: There are other groupings of service utilization and performance that could be important to the maintainer, that we did not explore in this paper. For example, if the service is using OAuth, then together with every request, in the header of the request there is information about the application which is sending a request. Grouping the information by application that sends the request could be important in such a context.

In general, providing a mechanism that would allow very easy specification of groupings (either as code annotations, as normal code, or as configuration options) is an open problem that Flask Dashboard and any other similar library will have to face.

XII. Related Work

Todo: Refer the reader to [3] and [5] for a more extensive discussion

[20]The Flask Dashboard detects the current version of the analyzed system the first time it is attached to the application object, and thus, assumes that the Flask application is restarted when a new version is deployed. This is in tune with the current version of Flask, but if the web server will support dynamic updates in the future, this might have to be taken into account

There is a long tradition of using visualization for gaining insight into software performance. Tools like Jinsight [2] and Web Services Navigator [9] pioneered such an approach for Java and for Web Services that communicate with SOAP messages. Both have an “omniscient” view of the services / objects and their interactions. As opposed to them, in our work we present an analytics platform which focuses on monitoring a single Python web service from its own point of view.

From the perspective of service monitoring, our work falls within the server-side run-time monitoring of services [3]. While we don't implement the more advanced features of related monitoring solutions like QoS policies driving the monitoring, it presents nevertheless an easy to use approach support improving the performance of web applications.

XIII. Conclusion and Future Work

Todo: update as appropriate

In this paper we have shown that it is possible to create a monitoring solution which provides basic insight into web service utilization and performance with very little effort from the developer. The user group that we are aiming for with this work is application developers using Flask and Python to build web applications with limited or no budget for implementing their own monitoring solutions. The emphasis is in allowing such users to gain insight into how the performance of the service evolves together with the application itself. We believe that the same architecture, and lessons can be applied to other frameworks and other languages.

In the future, we plan to perform case studies with other sytstems, with the goal of discover other needs and to wean out the less useful visualizations in the Flask Dashboard. We plan to also extend the tool towards supporting multiple deployments of the same applications across multiple nodes (e.g. for the situations where the application is deployed together with a load balancer). Finally, we plan to integrate Flask Dashboard with unit testing as a complementary source of information about performance evolution.

Appendix

A. *Changing the Default /dashboard Endpoint*

THIS STUFF SHOULD BE MOVED TO THE APPENDIX. And in a footnote we just mention that ... REQUIREMENT : The system must provide an easy way to add more specific configurations if needed. A way of overwriting the common sense defaults.

Further configuration is not needed possible by adding additional statements before the binding definition, for example,

```
...
dashboard.config.link = 'mydashboard'
dashboard.bind(flask_app)
```

allows for a custom route (`/mydashboard`) to the dashboard to be defined by the programmer. An external configuration file can be used instead, or in addition to this:

```
...
dashboard.config.from_file('config.cfg')
...
```

B. *For those who don't use GIT*

Manual version control requires the developer to tag each new version of the application with an appropriate version identifier [8] using the `APP_VERSION` configuration parameter, for example by adding to the configuration file:

```
    [dashboard]
    APP_VERSION=<versionID>
```

References

[1] M. Cavage. There is no getting around it: you are building a distributed system. *Communications of the ACM*, 56(6):63–70, 2013.

[2] 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*, pages 151–162, London, UK, 2002. Springer-Verlag.

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

[4] M. F. Lungu. Bootstrapping an ubiquitous monitoring ecosystem for accelerating vocabulary acquisition. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW 2016, pages 28:1–28:4, New York, NY, USA, 2016. ACM.

[5] A. Metzger, S. Benbernou, M. Carro, M. Driss, G. Kecskemeti, R. Kazhamiakin, K. Krytikos, A. Mocci, E. Di Nitto, B. Wetzstein, and F. Silvestri. Analytical quality assurance. In *Service research challenges and solutions for the future internet*, pages 209–270. Springer Berlin/Heidelberg, 2010.

[6] A. Metzger, O. Sammodi, K. Pohl, and M. Rzepka. Towards proactive adaptation with confidence: Augmenting service monitoring with online testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 20–28, New York, NY, USA, 2010. ACM.

[7] O. Nierstrasz and M. Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012.

[8] M. Papazoglou, V. Andrikopoulos, and S. Benbernou. Managing evolving services. *Software*, 28(3):49–55, 2011.

[9] W. D. Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar. Web services navigator: Visualizing the execution of web services. *IBM Systems Journal*, 44(4):821–845, 2005.

[10] B. Pernici, P. Plebani, and M. Vitali. About monitoring in a service world. In *Smart Cities, Green Technologies, and Intelligent Transport Systems*, pages 3–23. Springer, 2016.

[11] A. Ronacher. Quickstart. http://flask.pocoo.org/docs/0.12/quickstart/#quickstart, 2010. [Online; accessed 25-June-2017].

[12] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, pages 336–343, College Park, Maryland 20742, U.S.A., 1996.

[13] P. Vogel, T. Klooster, V. Andrikopoulos, and M. Lungu. A Low-Effort Analytics Platform for Visualizing Evolving Flask-Based Python Web Services. In *Proceedings of the 5th IEEE Working Conference on Software Visualization (VISSOFT 2017)*, page (to appear). IEEE Computer Society. Preprint available at https://github.com/mircealungu/vissoft17/blob/master/vissoft.pdf.