

Visualization-Focused Continuous Monitoring of Evolving Flask & Python-Based Web Services

Author One, Segundo Autor, Troisième Auteur

Famous Institute for Software Engineering and Computer Science
University of Great City, Some Country
Email: {first.last}@university.to

Abstract—Python is one of the fastest growing programming languages at the moment. Flask, a Python-based web framework is the technology used by tens of thousands of web applications. These applications, from interactive websites to service APIs, lack a technology-specific service monitoring solution.

In this paper, we present Flask Dashboard, a tool that provides insight into the utilization and performance of evolving Flask-based web services. We present the ease with which the tool 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

There is no getting around it: you are building a distributed system argues Mark Cavage in a recent article written for the *Communications of the ACM* [2]. Indeed, even the simplest second-year student project these days 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 Index¹ ranks Python as the 4th most popular programming language as of June 2016. An analysis of StackOverflow from September 2017 argues that “Python has a solid claim to being the fastest-growing major programming language”².

Within the Python community, Flask³ is a very popular web framework⁴. 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 [21].

Despite their popularity however, and 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 [17].

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 [16].

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, i.e. it was being operated without any monitoring solution.

Extending Previous Work

This paper is an extension of [Anonymous Paper by Anonymous Auhthors] which was published as a NIER paper in [Anonymous Conference]. The current paper extends that paper in the following ways:

- Providing more details about the case study in Section II and extending the data collection period by half a year.
- New perspectives and a better categorization of the perspectives are presented in Sections III – VI.
- A detailed discussion on automated outlier detection and monitoring is provided (Section IV).
- An advanced discussion on the integration with continuous integration environments in order to triangulate the live collected data with information from testing the end-points with a constant load and introduce a mechanism for performance prediction is added (Section VII)
- A more extended discussion about lessons learned while developing the tool including measurements on the overhead incurred by the presented tool (Sections VIII and IX)

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

²<https://stackoverflow.blog/2017/09/06/incredible-growth-python/>

³<http://flask.pocoo.org/>

⁴More 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

Zeeguu is a platform and an ecosystem of applications for accelerating vocabulary acquisition in a foreign language [13]. The architecture of the ecosystem has at its core an API and a series of ecosystem applications that together offer to a learner three main inter-dependent features:

- 1) *Reader applications* that provide one-click (or one touch, on touch-enabled devices) translations for the words and expressions in a text that the reader does not understand. Besides facilitating reading, the translations serve as input to a user knowledge model that is used further to generate personalized exercises and further reading recommendations.
- 2) *Interactive exercises* are generated based on the texts that the reader has studied in the past and are scheduled in such a way as to optimize the memory retention and to prioritize the most important unknown vocabulary items.
- 3) *Article recommendations* are also personalized for the interests and language capabilities of each learner. The recommendations come with a difficulty estimation that helps the learner find articles with appropriate difficulty.

Fig. 1 presents a high-level view of the case study ecosystem. The core API is implemented with Flask and Python and provides three types of functionality, respectively: contextual translations, personalized exercise suggestions, and article recommendations. In total, the API provides a bit under 50 endpoints in total, out of which around a dozen are very frequently used. The development of the core API itself is a research project.

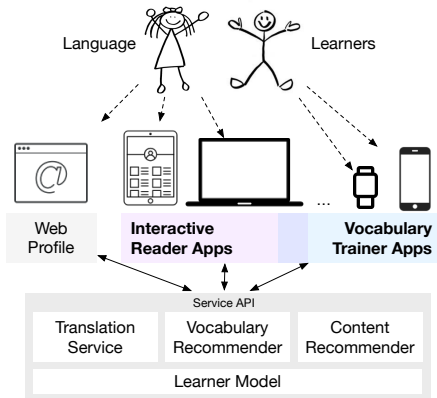


Fig. 1: The architecture of the Zeeguu ecosystem [13]

At the time of writing, the ecosystem consists of a reader web application, a web-based exercises platform, and a smart-watch application, which are used by more than two hundred active users on a regular basis. The users come from several Dutch highschools and a language center associated with a Dutch University. Several users are using it on their own, outside of any formal educational context. The learners use Zeeguu to study a variety of foreign languages including German, Dutch, and French. They have also diverse native languages, including: Chinese, Dutch, and English. The system

can scale to many languages by delegating the machine translation tasks to third party APIs.

A. Study characteristics

We will use this Zeeguu service API as a case study for this paper. It is open source, and has been deployed for several years without any monitoring solution until June 2017, when the Flask Dashboard has been deployed with it. During the ten month deployment (between June 2017 and the time of writing this article, April 2018) more than 112.000 requests to the API have been recorded by the API⁵. The highest load observed until now on the API consisted of 12K requests in a single day. With the help of the applications in the ecosystem, in the aforementioned time period the users have performed about 30K vocabulary exercises and have translated about 28K words and expressions while reading foreign language texts.

All the figures and measurements in this paper are captured from the actual deployment of Flask Dashboard in the context of the Zeeguu API. The figures are interactively offering basic data exploration capabilities: filter, zoom, and details on demand [22]. The Flask Dashboard deployment for the case study can be accessed publicly⁶.

B. The Technology Stack

As discussed in the previous section, Flask is a microframework for Python. The “micro” in microframework means Flask aims to keep the core simple but extensible. The framework does not make many decisions for the user, such as what database to use and those decisions that it does make, such as what templating engine to use, can be changed.

Flask is built on top of WSGI⁷ — The Web Server Gateway Interface — a calling convention for web servers to forward requests to web applications or frameworks written in Python. 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. The following code snippet presents a simple “Hello World” web service written in Python with the help of the Flask framework.

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

@app.route('/')
def index():
    return 'Hello_World!'
```

In the following we discuss the capabilities of the Flask Dashboard in more depth illustrating them for clarity on the case of the Zeeguu API.

⁵Note that the recorded requests are a lower bound on the usage of the API, as only a subset of the API endpoints have been actually instrumented

⁶<https://zeeguu.unibe.ch/api/dashboard; username: guest, password: icsme>

⁷<http://wsgi.org>

III. SERVICE UTILIZATION AND PERFORMANCE

The Flask Dashboard is designed to take advantage of the fact that it runs in the same context as the web application it monitors, and that the monitored API already has a web presence. It thus simply makes available one extra endpoint (i.e. /dashboard) at the same address as the API, which serves the interactive application presented in the remainder of this paper.

A dashboard can be attached to an already existing Flask application with a single line of code [citation withheld due to double blind] as the following snippet shows⁸:

```
import flask_monitoringdashboard as dashboard

# LOC #1: associate the main Flask application
# object with the dashboard
dashboard.bind(app)
```

During binding, the Flask Dashboard will search for all endpoints defined in the target application and make them available in the dashboard in an interactive configuration panel (Fig. 2). The panel presents all the automatically discovered endpoints and lets the user select the ones that should be monitored (the checkboxes in the last column). When the user enables the monitoring of an endpoint a function wrapper is added to the implementation of that endpoint (which is a function in Flask). The information is also saved in a database such that it is persistent across application restarts.



Endpoint	Method	Function	Timestamp	Monitor
/report_exercise_outcome/exercise...	OPTIONS, POST	api.report_exercise_outcome	2018-04-09 10:37:16	<input checked="" type="checkbox"/>
/get_possible_translations/drom_lan...	OPTIONS, POST	api.get_possible_translations	2018-04-09 10:28:11	<input checked="" type="checkbox"/>
/contribute_translation/drom_lang_c...	OPTIONS, POST	api.contribute_translation	2018-04-09 07:44:07	<input checked="" type="checkbox"/>
/translate_and_bookmark/drom_lan...	OPTIONS, POST	api.translate_and_bookmark	2018-04-07 10:33:02	<input checked="" type="checkbox"/>
/user_articles/recommended/kintzou...	GET, OPTIONS, HEAD	api.user_articles_recommended	2018-04-09 10:29:03	<input checked="" type="checkbox"/>
/get_exercise_log_for_bookmark/dro...	GET, OPTIONS, HEAD	api.get_exercise_log_for_bookmark		<input type="checkbox"/>

Fig. 2: Once connected to an API the Dashboard presents the endpoints that are available for monitoring

By default the dashboard takes an *opt-in* approach to monitoring: to prevent the performance penalties incurred by the dashboard⁹ to affect performance-sensitive endpoints, the service developer is asked to manually define the endpoints they want to monitor.

One alternative to configuring the monitored endpoints from the user interface is to allow the service developer to annotate the code. This would however pollute the code, and prevent deploying two versions which would monitor different endpoints. In addition, it would pose problems with endpoint evolution in the future.

The remainder of this section presents several interactive visualizations that are available in the dashboard without any further configuration¹⁰. They are focused on two types of information that is important to the API maintainer:

⁸We do not count the import statement that enables that line.

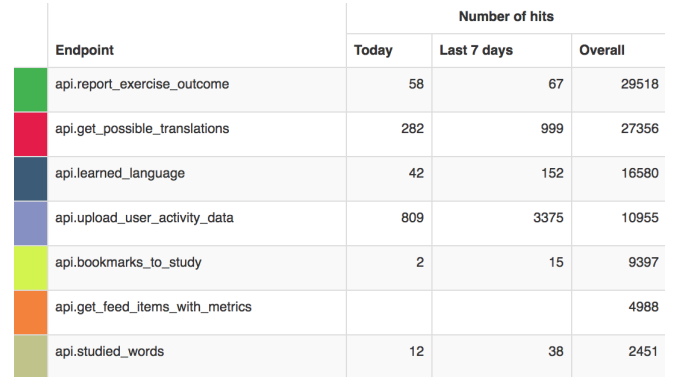
⁹We discuss performance issues later.

¹⁰We recommend obtaining a color version of this paper for better readability

- 1) **Utilization** – refers to how the third parties use the API, which parts are most used, which are less used, etc. This is known to be critical information for upstream developers in general[10]. In the case of source code dependencies one can detect such information by analyzing software repositories. However, in the case of services, there is no other way but monitoring service utilization.
- 2) **Performance** as measured in response times. Knowing the performance of one's API is critical for the quality of a service API, and especially so for APIs that are supposed to be integrated in live systems.

A. Utilization

The default landing page of the dashboard presents an overview of all the endpoints, which presents some basic utilization information as: (1) the number of calls to that endpoint since the beginning of tracking (2) the number of calls in the last seven days, and (3) the number of calls in the last day



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. 3: The API Overview presents synthetic endpoint utilization information

This kind of information is already very useful, and can provide the API maintainer with insight into the evolution of their service API. The seven endpoints present in Fig. 3 illustrate three types of insight that the maintainer can obtain by looking at such a view:

- **Service Specific Usage Patterns.** The two most used endpoints in the figure stand for the two main activities in the system:
 - get_possible_translations (■) is an indicator of the amount of foreign language reading the users are doing.
 - report_exercise_outcome (■) is an indicator of the amount of foreign vocabulary practice the users are doing.
- **Sudden Increase of Endpoint Usage.** One endpoint (upload_user_activity_data (■)) has disproportionately been used in the last day and last week.
- **Possibly Discontinued Endpoint Usage.** One endpoint (get_feed_items_with_metrics (■)) has not been

used in the last 7 days. This might be an endpoint that is deprecated.

For a more detailed view of utilization evolution, Fig. 4 shows the **Daily API Utilization** perspective on API utilization that Flask Dashboard provides: a stacked bar chart of the number of hits to various endpoints grouped by day¹¹. Fig. 4 in particular shows a peak utilization, on June 28, 2017 when the API had more than 12.000 hits¹².

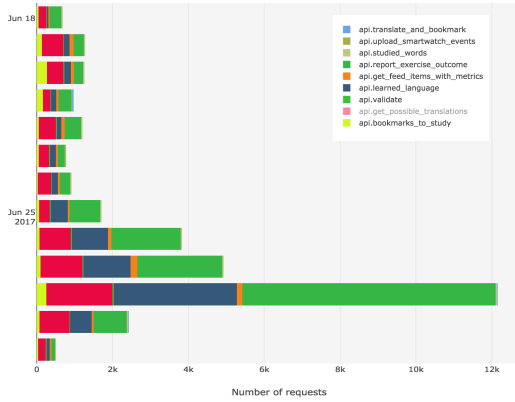


Fig. 4: Usage patterns become easy to spot in the requests per hour heatmap

Another utilization perspective is the **Hourly API Utilization** in which the Flask Dashboard can highlight *cyclic patterns of usage per hour of day* by means of a heatmap, as in Fig. 5.

Figure 5 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 happened in on afternoon/evening.

¹¹Endpoint colors are the same in different views

¹²Turns out that the high school students that were using the system for their French-learning class had a deadline on that date

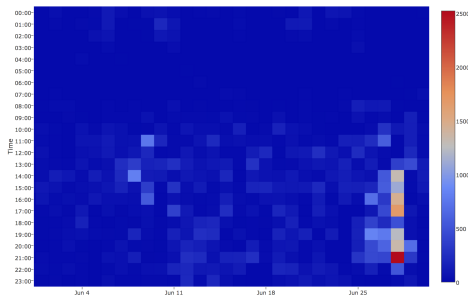


Fig. 5: Usage patterns become easy to spot in the requests per hour heatmap

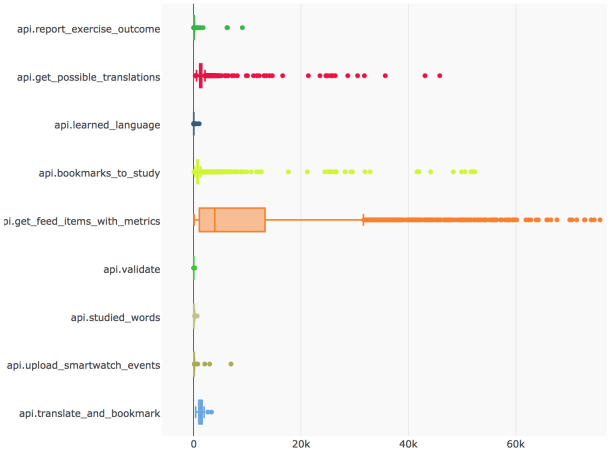


Fig. 6: The response time (in ms) per monitored endpoint view highlights performance variability and balancing issues

B. Performance

The **API Performance** visualization perspective presents an overview of the response times for all the tracked endpoints by using a box-and-whiskers plot.

Fig. 6 presents such a perspective from our case study which was taken in the last quarter of 2017. Several observations are straightforward:

- The slowest endpoint and the one with the highest variability is `get_feed_items_with_metrics` (■) : it retrieves a list of recommended articles for a given user. Since a user can be subscribed to anything from one to three dozen article sources, and since the computation of the difficulty was done in real time, the variability in time among users is likely to be very large.
- `get_possible_translations` (■) and `bookmarks_to_study` (■), two of the most used endpoints (as seen in Fig. 4) are also some of the slowest and with most outliers. The first one is very critical for the usability of the Reader applications since it called when a user taps on a word to receive a translation.

The **API Performance** perspective is important for allocating optimization efforts. In our case study, after seeing this perspective, the API developers decided to rearchitect the system in order to improve the performance of `get_possible_translations` and `get_feed_items_with_metrics`.

IV. AUTOMATED OUTLIER DETECTION AND MONITORING

When an API endpoint is called from within a highly interactive application (as it is the case with `get_possible_translations` (■) in the previous section) of particular interest to the API developers are *performance outliers*. Indeed, a translation request that takes three times more than expected can decrease the perceived quality of the application significantly. Thus, identifying, collecting all appropriate data, and diagnosing the root causes of such outliers is especially critical in improving application quality.

For this purpose the Flask Dashboard tracks for every monitored endpoint a *running average* response time value.¹³ When it detects that a given request is an outlier with respect to this past average running value¹⁴, it triggers the *outlier data collection routine* which stores extra information about the current execution environment. This extra data includes: the current Python stack trace, CPU load, memory consumption, request parameters, etc.



Fig. 7: Automatically collected outlier information for the `get_possible_translations` (■) endpoint

Fig. 7 shows an example of a **Performance Outlier Detail** perspective on one of the outliers in the `get_possible_translations` (■) endpoints. In this particular case, the analyst could corroborate several bits of informations:

- Neither the memory (1) nor the processor (2) were overloaded at the moment of the request. Thus the slow response is not due to the machine being overloaded.
- The stack trace snapshot shows that code was in the `google_translator.py`. Since the system uses as back-end multiple translators it is revealing to see that in this particular case it was waiting for the Google Translator. By manually investigating multiple outliers, the developer discovered that in a large percentage of the cases the Google Translator was indeed the reason for the slow response time.

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.

¹³ 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.

¹⁴ A configurable threshold with a default value of 2.5 times the running average response time is used for this purpose

V. IDENTITY-BASED GROUPING OF REQUESTS

It is sometimes the case that the utilization and performance of an API must be understood not only per endpoint, but also by grouping the requests by another criterion. Two such examples are:

- When a service has different types of users (e.g. the paying vs. the free-for-evaluation ones, or the in-house vs. those in a different organization) the maintainer must understand the performance for the different groups
- When the load mix of the users can vary dramatically and the system response time is a function of the individual user load¹⁵ the maintainer must understand the variation of performance with the users (and implicitly the user load).

To support narrowing down the analysis to groups of users, or even individual users, the Flask Dashboard can log together with every request grouping information for that request. The simplest way of achieving this is to take advantage of the architecture of Flask applications in which a global `flask.request` is used to retrieve session information which can in turn is normally used to identify the user sending the request. From the user one can obtain the group.

The following code snippet shows how we enabled Flask Dashboard to enable user-by-user analysis¹⁶:

```
# LOC #2: group requests by user id
dashboard.config.group_by = 'User',
lambda: Session.find(flask.request).user.id
```

The `dashboard.config.group_by` is assigned a tuple with two elements, in which:

- 'User' stands for the name of this particular grouping strategy
- `lambda: Session.find...` is a callback with no arguments, that makes use of the global `flask.request` and extracts from the request the current session, and in turn the user id¹⁷

In Zeeguu, it was decided to further understand the per-user performance of `get_feed_items_with_metrics` — an endpoint that retrieves a list of recommended articles for a given user. It is the endpoint with the slowest response time and highest variability (see Fig. 6).

After configuring the Flask Dashboard as shown earlier, the **Per-User Endpoint Performance** perspective becomes available to present the different response times for different users. Figure 8 presents a subset of the corresponding view in the Flask Dashboard. The figure shows that the response times for this endpoint varied considerably for different users with some extreme cases where a user has to wait a full minute until their recommended articles were shown.

¹⁵ E.g. in Gmail some users have a few dozen emails while other have tens of thousands: this difference in user loads will eventually induce a difference in the response times for different users

¹⁶ If we wanted to group by the user group for example, the code would change slightly by replacing "user.id" with "user.group.id"

¹⁷ Note that every web service or application must have a way of associating a request with a user. In fact, the `Session.find(request)` was already an existing function in the analyzed system

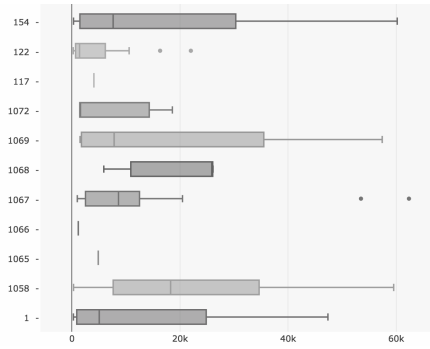


Fig. 8: And endpoint with a high variability across users

The reason for this behavior is that a user can be subscribed to anything from one to three dozen article sources and for each of the sources the system computed the personalized difficulty of each article at every request. After seeing the two perspectives presented in Figures 6 and 8, the Zeeguu maintainer refactored the architecture of the system to move this difficult computation out of the interactive loop.

VI. VERSION-AWARE MONITORING

The Stack Overflow Developer Survey from 2018 revealed the fact that 88.4% of professional developers use Git. Given that versioning using git is the main way in which the source code of services evolves and is deployed, it is natural to monitor their progress also across versions.

Version control in the dashboard can be supported in two ways: (1) the developer explicitly states the current version, or (2) the current version is automatically detected based on some version control system.

For Flask Dashboard to start using Git, an extra configuration line enables it to automatically¹⁸ associate API traffic with the currently deployed version based on information in the .git folder¹⁹:

```
# LOC #3: provide the dashboard with
# information on where to find
# git information
dashboard.config.git = 'path/to/.git'
```

This technique assumes that the web application code which is the target of the monitoring is deployed using git by pulling the latest version of the code from the integration server; this will result in a new commit being pointed at by the HEAD pointer. Thus, when the deployment engineer restarts the new version of the service, 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. 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 would not work in the presence of dynamic updates.

¹⁸Alternatively, the maintainer can add version identifiers manually for the web application through a configuration file if the system does not use Git.

¹⁹<https://git-scm.com/>

Evolving Utilization

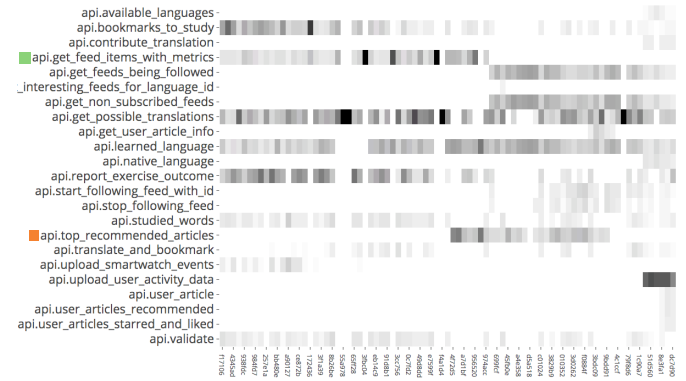


Fig. 9: Evolving Endpoint Utilization Across System Versions

Fig. 9 shows the **Multi-Version API Utilization** perspective which presents the utilization of the tracked endpoints across versions. The view is a matrix, in which, the intensity of the color at the intersection of an endpoint line and a version column is proportional to the percentage of requests served by that endpoint in that particular version²⁰.

Several patterns could be visible on such a graph:

- Endpoints being deprecated. In the case study, `get_feed_items_with_metrics` (■) stops being used somewhere after the half of the presented timeline.
- Endpoints being introduced. In the case study, `top_recommended_articles` (■) appears soon after the half of the presented timeline.
- Endpoint renames. Although not visible in the figure, the candidates for rename would be made easily visible as one would be discontinued at the same time when the other would appear.

Evolving Performance

Fig. 10 is a zoomed-in version of such a view for `get_possible_translations` (■) with versions increasing from top to bottom.

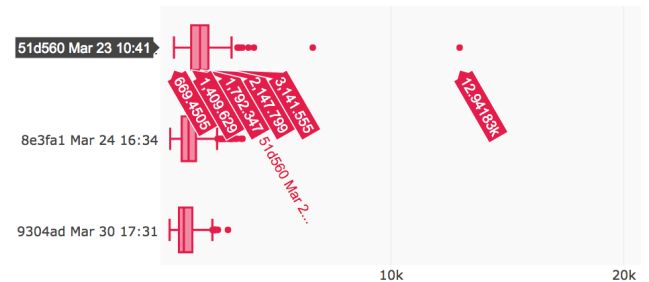


Fig. 10: Endpoint performance across three versions

²⁰The chart does not plot the absolute utilization of that endpoint in that version but rather the percentage of all the API calls that go to that endpoint. Otherwise a version that is deployed for many weeks would make all those deployed for a few days invisible

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. 10) to 0.8 in the latest version (bottom-most box plot).

Evolving Grouped Performance

The limitation of the previous view is that it does not present the information also on a per user basis. To address this, a different visual perspective entitled **User-Focused Multi-Version Endpoint Performance** can be defined. Fig. 11 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.

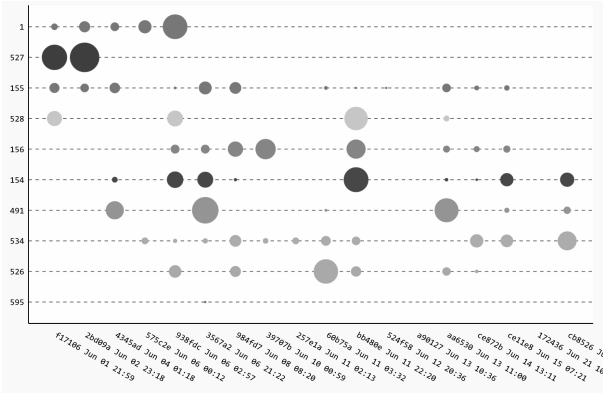


Fig. 11: This perspective shows that the evolution of response times for individual users (horizontal lines) across versions (the x-axis) for a given endpoint

The colors represent users. The figure shows average performance varying across users and versions with no clear trend: 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 with `id=1`, performance degrades over consecutive versions. Given that for other users the performance does not degrade in the same way, it is probable that the problem might lay elsewhere: the server was overloaded or the endpoint is inherently “algorithmically slower”.

VII. PERFORMANCE TRIANGULATION WITH REGRESSION MONITORING

Suppose the developer observes a performance degradation of a given API endpoint in the latest version of the system. How do they know whether the degradation is due to the performance of the code, the server load, or the workload mix of the user?

One way of triangulating such an observation is by monitoring the evolution of the performance of the endpoints when tested with a constant load across versions. Thus, if the performance decreases in a new version, and the endpoint in question is tested with a constant load the problem must be the endpoint implementation.

Integrating with Continuous Integration Servers

To monitor endpoint performance with a constant load Flask Dashboard takes advantage of two best practices in software evolution: (1) an API must have unit tests for its endpoints, and (2) these unit tests will be run within a Continuous Integration (CI) server. By tracking the timing of these unit tests one can obtain a perspective on endpoint performance evolution with a constant load (as long as the unit tests do not change).

Flask Dashboard can be configured to work together with CI frameworks like Travis²¹ that deal with automated integration testing. To do this one needs to simply add one extra line in the script that runs their continuous integration testing. In the case of Travis CI, a developer has to add the following line in the ‘.travis.yml’ configuration file:

```
# LOC #4: is added to the .travis.yml file
python -m flask_monitoringdashboard.collect
--test_folder=./tests_zeeguu_api --times=5
--url=https://zeeguu.unibe.ch/api/dashboard
```

Each time a new build is created through Travis, the Flask Dashboard automatically detects all available unit tests defined by the application developer (in `--test_folder`) and iterates through each one of them a number of times (`--times`) while monitoring the response times for each test. The resulting measurements are uploaded to a specific endpoint that is created by the tool at the API url (`--url`). The data uploaded to this endpoint are persisted in a separate part of the Flask Dashboard database so as not to contaminate the “live” monitoring data.

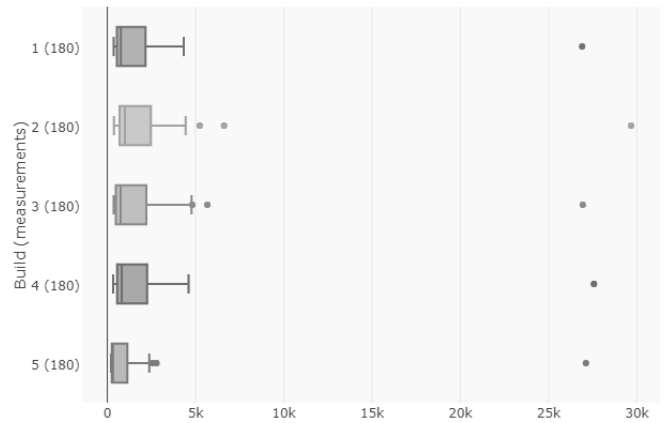


Fig. 12: Response times in 5 consequent Travis builds

Fig. 12 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.

²¹<https://travis-ci.org/>, one of the most popular CI solutions at the moment

Preemptive Monitoring

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 [15], 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 to create an emulated “live” environment for integration testing purposes, and use unit testing as the dedicated test input.

While this integration testing environment is different from the production one, and the load used is purely synthetic, it can serve as an early performance indicator for the developer. Fig. 13 shows the actual endpoint response times across five versions of the system (above) and the corresponding testing times for the synthetic load in the same five versions (below).

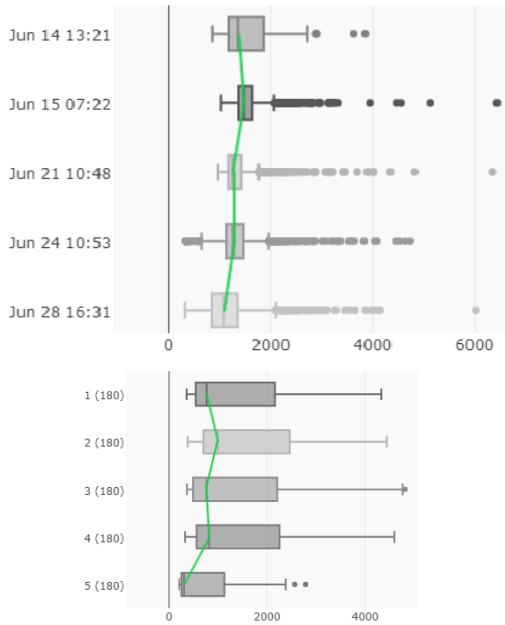


Fig. 13: The reported response time (in ms) per deployed version in the observation period: in the actual deployment (above) and using integration with Travis (below). Green line hints to a correlation between the two sets of measurements.

Computing Pearson correlation ($r(3) = .93, p = .02$) between the median values of the two datasets (cf. Table I) shows a correlation between the two. Further investigation on the characteristics of this observation is currently ongoing and it will be elaborated further in future work.

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

TABLE I: Median response times in Figure 13

VIII. OVERHEAD OF THE FLASK DASHBOARD

To measure the performance overhead of the Flask Dashboard, we have implemented an automated benchmarking system. It is open source and available online and can be tested by the reader²². The benchmark downloads the latest version of the Zeeguu API and installs it in a Docker container. Then it calls several selected endpoints for 500 times each, tracking the response times. The endpoints are called in three different configurations:

- 1) With no dashboard installed.
- 2) With the dashboard enabled but with the outlier detection deactivated.
- 3) With the dashboard enabled and the outlier threshold set to zero, thus effectively treating every request *like it were an outlier*.²³

The last configuration is meant to evaluate the effect of the outlier detection as discussed in Section IV to the overall performance. Fig. 14 and Table II use violin plots and descriptive statistics, respectively, to present the distribution of response times resulting from running the benchmark for three different API endpoints on a quad-core machine, with Intel Core i5-4590 processor @3.30GHz, 8G of RAM and 240GB SSD disk drive.

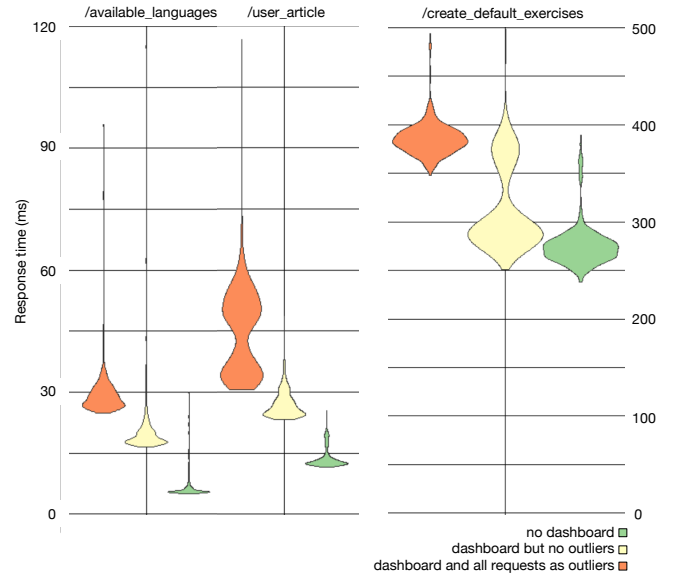


Fig. 14: The distribution of response times when calling the three endpoints for 500 times in three conditions: no dashboard, dashboard but no outliers, dashboard and every request treated as an outlier

The three endpoints that are tested fall in two categories of complexity:

- The two fast ones are very simple. The first returns a list of language codes which are defined in the code, so it

²²[The link will be added after deanonymization]

²³This forces all the requests to be treated as outliers, and thus provides insight into this situation, which otherwise would be hard to generate

Endpoint	Dash.	All Out.	Mean (ms)	SD (ms)
available_languages			6.3	2.3
available_languages	✓		19.9	5.6
available_languages	✓	✓	29.4	5.4
user_article			13.9	2.6
user_article	✓		26.9	3.2
user_article	✓	✓	44.8	10.0
create_default_ex...			276.0	22.4
create_default_ex...	✓		313.6	43.2
create_default_ex...	✓	✓	385.3	16.9

TABLE II: Response times for three endpoints run 500 times in three different conditions each. Dash. = with dashboard. All Out. = every call being considered an outlier

does not touch the Zeeguu database. The second does a simple read from the database.

- The slower endpoint does several complex writes to the database, so this is why it is much slower. However, in our case study, about half of the measured endpoints were at least as slow as this one, so its response time is representative.

The data shows that the dashboard (without outliers) introduces for the faster endpoints an overhead of 14ms and for the slower endpoint an overhead of 40ms, on average. In the case of an outlier, the overhead is doubled, so the design decision of only collecting the extra information only for outliers seems to be validated. Depending on the monitored application these numbers might be acceptable or not. In the case of the Zeeguu case study, the maintainers find this overhead acceptable since they do not affect the overall user experience in any significant manner.

IX. DISCUSSION

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.

The presented approach was centered around an API written in Python and Flask, but, in principle, we can use the same approach for other technologies. 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. However, all the interactive perspectives we developed can be equally applied to other service technologies (e.g. Django) by simply providing a few back-end adapters in the right places (e.g. endpoint discovery, the /dashboard endpoint deployment, etc.) As such, our approach is more generic than the technologies that it relies upon for its implementation.

At the same time, the presented approach also comes with several limitations. Some of these limitations might also represent challenges for the future builders of similar tools.

The Dashboard Overhead

We have provided a mechanism for measuring performance, and showed a glimpse into the overhead imposed by the first version of the Dashboard. For our case study the overhead is

acceptable, but for performance critical applications, it might be too large.

One of the downsides of the current implementation is the fact that the dashboard runs in the same process as the main application and that Python only supports green threads. The performance could be greatly improved if instead the dashboard would run in a different process, and a message queue was used to allow the decoupling of the monitoring itself from the persistence of measurements to the dashboard DB. However, that would complicate the installation and configuration of the Flask Dashboard.

Limitations of the Measurements

One limitation of the benchmarking we used in Section VIII is that it does not use a more realistic mix of workload, but rather, measures individual endpoints. It might be that multiple concurrent and diverse endpoints would result into different performance impact. In fact, based on the data in the dashboard one could actually construct a statistically relevant workload mix. At this point, this constitutes future work.

Lack of Flexibility in Allowing Multiple Groupings

To be agile, a tool has to be easily adaptable to different usage scenarios. However, currently one of the current limitations of Flask Dashboard is that that one can specify only one type of grouping. This is insufficient for cases where multiple groupings are desirable. In the Zeeguu case study this need emerged in the context in which it would be good to grouping the requests also by the client application that sends the request. This feature is currently under development.

Limitations to Evolution Monitoring

The advantage of the presented approach to evolution monitoring based on observing the .git folder is the need for minimal configuration effort, as discussed in the presentation of the Flask Dashboard. 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 to the Flask Dashboard.

Endpoint Provenance

The Flask Dashboard can not detect endpoint renames, and thus the history is normally lost in the case of an API rename. The problem can be approached in a similar way to the *provenance problem* in source code evolution analysis)[5] by inferring that a newly introduced endpoint with similar patterns of request traffic might be the same as one that is not used anymore. Since this would still be a statistical approach, the integration in the tool would have to ask the user for confirmation. However, this would work better in a system where endpoint tracking is activated by default for all the endpoints. Otherwise, it would require the user to remember to always enable the tracking of the new endpoint.

Performance Prediction

In Section VII we provided evidence towards our assumption that integration testing can be used for performance triangulation across service versions. The idea is to use a synthetic load based on unit testing to drive endpoints before they are deployed in production and observe whether performance improves or deteriorates on a version by version basis. The next logical step is to use this information for performance prediction purposes. A full blown performance prediction feature would require advanced statistical work, but it can be used as an advanced warning system for the overall performance trend for a given set of endpoints. As discussed in Section VII however further evaluation of this feature is currently ongoing.

Distributed Deployment

Last but not least, the presented approach and case study assume that the monitored Flask application is deployed as a single node, i.e. without support for horizontal scaling by means of replication [23] — or at least that the application owner is interested in monitoring each application deployment in isolation. This is a clear limitation of this work which we are already working towards addressing by allowing multiple application instances to be registered with the same dashboard, essentially providing a “meta-dashboard” approach through which application endpoint performance and utilization can be monitored and visualized in a unified way. Due to the addition of an orthogonal to the service evolution dimension that the Flask Dashboard currently supports, this requires a significant effort of re-engineering on our part.

Need of Further Proof of Usefulness and Ease of Adoptability

We designed the dashboard to make it easy to adopt and we think this is to a certain degree apparent. However, we have only presented a single use case where the API developers have used the dashboard. It has been successful, since based on the dashboard, the developers have become aware of the limitations of their service APIs and have started improving it. However, we believe that it will be worthwhile to organize a dedicated study to observe the challenges and opportunities of the adoption of such a dashboard across different applications. It will also be important to see whether the perspectives presented here are relevant for other developers, and also which other perspectives are missing.

X. RELATED WORK

A rich body of work exists that studies the evolution and maintenance of APIs from the perspective of source code evolution [7], [11], [12]. However, since services are increasingly being used across a variety of systems in which adaptability, flexibility, and environmental awareness are essential [19] in our work we focus on the analysis of the runtime aspects of service APIs. Our work is thus, more related to the work on dynamic analysis of software systems; however, most of this work has been done in order to support the reverse engineering and enhance the understanding of software systems[4].

In this context, our work falls within the server-side run-time monitoring of services [9]. There is a long tradition of using visualization for gaining insight into software performance. Tools like Jinsight [6] and Web Services Navigator [18] pioneered such an approach for Java and for Web Services that communicate with SOAP messages. More recent tools present live visualizations for monitoring multiple interacting software systems utilizing the city metaphor for each software system [8]. All these approaches 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.

Kieker – has been presented by Rohr et al [20] as a system targeted at monitoring the performance of Java services. The system provides better performance in terms of overhead with respect to ours, but does not support integration with git and multi-version analysis as the system presented here.

Similar to our discussion on pre-emptive monitoring, Cito et al. [3] argue for bringing performance measurements back in the IDE to provide the developers with such information during the development process.

In their work Baresi and Guinea have proposed the Multi-layer Collection and Constraint Language (mlCCL) which allows them to define how to collect, aggregate, and analyze runtime data in a multi-layered system. They also present ECoWare, a framework for event correlation and aggregation that supports mlCCL, and provides a dashboard for on-line and off-line drill-down analyses of collected data [1].

Monitoring can be performed on different types of system components and for several purposes beyond the ones presented here. Service-oriented literature on the subject discussed the multitude of aspects related to monitoring for services; the interested reader is referred to [9], [14], or more recently [19] for an extensive presentation of the subject.

XI. CONCLUSION AND FUTURE WORK

In this paper we discussed our proposal for a monitoring dashboard for Flask-based web applications as the means for providing a low effort analytics solution. The emphasis is in allowing application developers to gain insight into how the performance and utilization of their services evolves together with the application itself. Flask Dashboard allows for integration with both `git` and the Travis CI platform. We also discussed more advanced features of the dashboard and provided evidence towards the use of unit testing-sourced synthetic loads as a potential limited form of performance prediction. Finally we measured the overhead of the proposed solution within acceptable by the case study application owner limits. Section IX which discusses what we perceive as limitations of our proposal and also potential challenges for other similar tools, effectively acts also as a roadmap for further development, with the ultimate goal of increasing the adoption of the Flask Dashboard by as many projects as possible. This will allow us to evaluate our proposal in depth in the future.

REFERENCES

- [1] L. Baresi and S. Guinea. Event-based multi-level service monitoring. In *2013 IEEE 20th International Conference on Web Services*, pages 83–90, June 2013.
- [2] M. Cavege. There is no getting around it: you are building a distributed system. *Communications of the ACM*, 56(6):63–70, 2013.
- [3] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth. Runtime metric meets developer: building better cloud applications using feedback. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 14–27. ACM, 2015.
- [4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.*, 35(5):684–702, Sept. 2009.
- [5] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR’11: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 183–192, 2011.
- [6] 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.
- [7] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software: Evolution and Process*, 18(2):83–107, 2006.
- [8] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.
- [9] C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and analysis of web services*, pages 237–264. Springer, 2007.
- [10] 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)*, pages 1–6, 2014.
- [11] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. How do developers react to api evolution? the pharo ecosystem case. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 251–260. IEEE, 2015.
- [12] A. Hora and M. T. Valente. apiwave: Keeping track of api popularity and migration. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 321–323. IEEE, 2015.
- [13] M. F. Lungu, D. Chirtoaca, M. Avagyan, and L. van den Brand. As We May Study: Towards the Web as a Personalized Language Textbook. In *Proceedings of CHI’18: ACM Conference on Human Factors in Computing Systems*, page (to appear). ACM.
- [14] 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.
- [15] 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.
- [16] O. Nierstrasz and M. Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012.
- [17] M. Papazoglou, V. Andrikopoulos, and S. Benbernou. Managing evolving services. *Software*, 28(3):49–55, 2011.
- [18] 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.
- [19] 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.
- [20] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoecker, S. Giesecke, and W. Hasselbring. Kieker: continuous monitoring and on demand visualization of java software behavior. In *Proceedings of IASTED 2008 – International Conference on Software Engineering*, pages 80–85, 01 2008.
- [21] A. Ronacher. Quickstart. <http://flask.pocoo.org/docs/0.12/quickstart/#quickstart>, 2010. [Online; accessed 25-June-2017].
- [22] 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.
- [23] L. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.