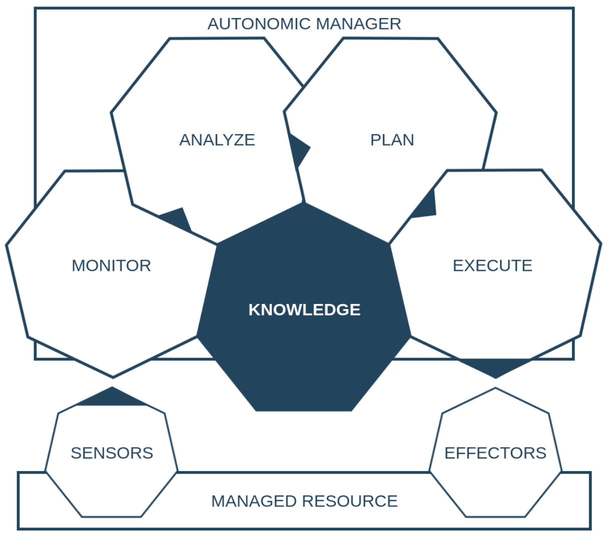
Effects of Openstack Watcher Deployment in a Large Scale Cloud

European Organization for Nuclear Research (CERN)

Corne Kenneth Lukken
Faculty Digital Media & Creative Industry
Amsterdam University of Applied Sciences
Amsterdam, Netherlands
corne.lukken@hva.nl



Internship Supervisor: Marten Teitsma
Site Supervisors: Jan van Eldik & Jose Castro Leon

Amsterdam University of Applied Sciences

PREFACE

Several years ago I became highly interested in the organization which had driven the discovery of the Higgs boson, namely CERN. The Amsterdam University of Applied Sciences (AUAS) had managed to enroll students for internships at CERN in the past and I was determined to figure out how to become one of them. Eventually, I came into contact with Marten Teitsma who, unfortunately, told me that the only students entering were almost always from the Technical Informatics (TI) course. Together we arranged that I could follow additional courses from TI and set out to maximize my change of entry. Years went by and eventually I set out to follow the minor software for science which led to a project around the A Large Ion Collider Experiment (ALICE) detector. Additionally, this minor also included a presentation from Nikhef about possible places for internships at CERN. I prepared as best as I could for the application which underwent several revision and in the end I was still not fully satisfied with my application. Nevertheless, around the end of November in 2018 I got the exciting news that my application was accepted. I could not be happier and I would not be here without my friends and family who throughout all my frustration, stress and absence during events kept supporting me.

Corne Lukken
March 2019

Arigatou Gozaimasu

CONTENTS

***Abstract*— Many organizations providing computer infrastructure on demand face similar problems. This kind of on demand infrastructure is known as Infrastructure as a Service (IaaS). The problems with IaaS typically come from users not being able to correctly estimate the amount of resources they need. This results in poor utilizations or over utilizations of the available infrastructure. As a result large parts of the available hardware end up being idle and unused while other parts might see bad performance because of over utilizations.**

At the European Organization for Nuclear Research (CERN) they face similar problems. Their cloud is based on OpenStack, an open source framework built from many different components that facilitate IaaS. One of these components is called Watcher and it is made to resolve under and over utilization, among other things. However, Watcher is relatively new and not used as extensively as other components. Earlier evaluations by CERN showed that Watcher could not be deployed without changes. The question remained: what changes are required to deploy Watcher?

To determine required changes both Watcher and it's community are evaluated. The results from these evaluations include both experimental and literature based analysis. The community is evaluated based on the platforms it uses to collaborate. Watcher is evaluated using both source code and operational analysis. Each identified possible improvement is described in its own section were methods and results are individually addressed.

The evaluations lead to the rescheduling of bi-weekly meetings. These meetings are of importance for the community to discuss ongoing matters. Even though the evaluations identified some possible improvements the community is surprisingly mature, indicated by the correct and extensive use of many collaborative tools.

The evaluation of Watcher itself identified many changes, some of which were determined by previous evaluations done by CERN. Primarily, the development of a new datasource is the most crucial improvement to be developed. This new datasource is required because all the other datasources are not used at CERN. A series of other changes significantly improves the performance by reducing the time to build so called data models.

The implemented datasource allows Grafana to be used with its subsequent databases. The solution is extensible so that any future Grafana database can be easily integrated into the developed Grafana datasource.

The build time of the data model is reduced in three individual changes. The first change provides a scope to limit the amount of infrastructure that is taken into account. Because of the size of CERN's OpenStack infrastructure this change is essential. CERN typically wants to limit this scope to a single so called cell. The scope reduces the estimated build time from 26471 to 427 seconds for a single cell on average. The second change improves the API calls that are made during the construction of the data model resulting in a further 32% performance improvement for single and multiple cells. Finally, The last change introduces parallelism to the API calls further reducing the time to build the data model to 38 seconds for single and multiple cells. However, the parallelism is still a proof of concept and can not be implemented into Watcher in its current state.

Further work should evaluate the current strategies as it is likely that they do not perform optimally or behave in unexpected ways. By improving these strategies Watcher can be used instead of just being deployed.

Index Terms—**OpenStack, Watcher, MAPE-K, Grafana**

1. Prospects of CERN's Watcher Enabled OpenStack Cloud

The European Organization for Nuclear Research better known as CERN was founded in 1954 and has become one of Europe's largest research organization. It aims to provide fundamental research in the field of particle physics thereby helping to discover what the universe is made of. To accomplish this mission CERN is home to the worlds largest particle accelerator known as the Large Hadron Collider (LHC). With over 30.000 people working with or for CERN every year its main site located on the border between France and Switzerland has become a large hub for science [1].

1.1. Resource Provisioning Services

The organizational structure of CERN is sectioned into departments which are split into groups and those groups are further divided into sections. One of these sections is the RPS section. They are tasked with giving the right computer resources to the right users in a fast, flexible, secure and scalable manner. Many pieces of literature have been written on different approaches to reach these goals and large industries have entire research & development (R&D) teams devoted to improve these approaches and their implementations. Currently, a common approach to realize these high user demands is infrastructure as a service (IaaS). This approach allows users to interface with high-level Application Programming Interfaces (API) to perform underlying tasks such as managing computing resources, data partitioning and scaling. Typically, the computing resources are managed through a hypervisor, but in the case of OpenStack [2], a popular open source IaaS framework, this can also be realized with containers and even bare-metal.

Hypervisors and containers are methods[1] to run applications and even entire operating systems in an isolated manner such that the underlying host processes and memory can not be accessed from within the environment. Furthermore, hypervisors and containers allow to limit the amount of resources their guests can use and also allow to control networking and other hardware interactions. These technologies are a fundamental part of the IaaS paradigm.

The RPS team manages to leverage this IaaS paradigm well with OpenStack this shows as many of its member contribute heavily to OpenStack and have gained important privileges within the community. The team is able to give users access to resources quickly and easily which allows them to work more effectively but it remains a problem for users to correctly estimate the amount of resources they need. Underutilized computing resources or resource contention due to over utilization are the most common problems of any RPS team. CERN's RPS team sees the same problems in their large scale OpenStack cloud. Which methods could reduce under utilization and resource contention in an OpenStack based cloud?

[1]Throuhgout this work, method will be short for methodology and constitute an approach or set of behavior. It will not be used as terminology for functions in programming languages.

1.2. *The OpenStack Cloud at CERN*

The scale of CERN's OpenStack cloud is very large with over 15000 physical servers placed inside their data center they can consume up to five megawatts of power [3]. The servers consist of varieties of different hardware and many different OpenStack components are deployed on top to cater a large variety of user requests. Some of these servers are placed in so called critical regions with their own separate power and cooling systems. Typically servers are placed in to so called cells which contain between 50 and 200 servers. these cells allow OpenStack clouds to scale to larger sizes. Additionally, these cells allow for flexibility to better meet user demand. Because of the scale of this OpenStack cloud many components are run in parallel with load balancers in front so that the large amount of requests can be handled accordingly.

1.3. *OpenStack*

CERN has chosen for OpenStack as IaaS framework since it has many desired features, one of which is a component architecture. It consists of many smaller components, of which many are optional but some are mandatory. The architecture of OpenStack is similar to a micro service architecture where many small components fulfill individual functions but the overall system could continue to function even with many of the individual components not functioning. The micro service architecture has gained popularity ever since initial appearances around 2012 although the original author is unknown. Typically individual OpenStack components consist of several processes but in most components it will be three processes 1) API, to handle requests from external components. 2) Analyze, gathering information about states of other services or systems. 3) Execute, applying the changes based on the gathered information. During the time of writing OpenStack consists of roughly 40 different components of which eight are considered core components [4]. The eight core components realize networking, virtual machines, orchestration, storage for objects, blocks and images and identification.

1.4. *OpenStack Watcher*

One of these OpenStack components is Watcher. It is a relatively new project and "provides a flexible and scalable resource optimization service for multi-tenant OpenStack-based clouds" [2]. Watcher uses the MAPE-K feedback loop invented by IBM in 2005 [5] which is a four step loop developed for autonomous and distributed systems and aims to provide the common desired attributes of such systems, namely self-healing & self-optimization. To perform this loop MAPE-K relies on two external systems being the sensors & effectors were these two systems are respectively the source of data & where the decisions are applied to. With the data from the sensors the feedback loop consisting of 1) Monitor 2) Analyze 3) Plan 4) Execute. It is executed using the Knowledge the system should have, hence the K in MAPE-K.

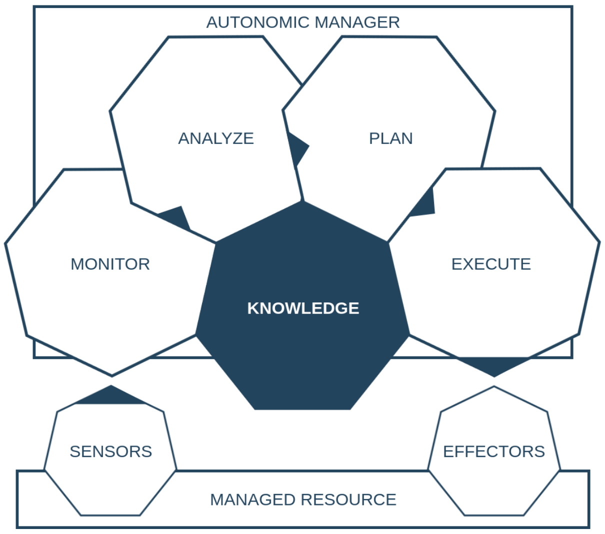


Fig. 1: Diagram of MAPE-K feedback loop CC-BY 4.0[0]

Currently OpenStack Watcher has multiple datasources for its sensory input being 1) Monasca [6] 2) Gnocchi [7] 3) Ceilometer [8]. All three of which are other OpenStack components. However, at this time Ceilometer is being deprecated and the use of it should be avoided. For its effectors Watcher relies heavily on interactions with Nova [9] which is the OpenStack compute component but interactions with other components such as Neutron [10] for networking or Ironic [11] for bare-metal may also occur.

Watcher is split into three different small executables each with their own tasks. 1) The API, which listens for incoming messages to perform operations. 2) The decision-engine, which executes strategies in order to create an action plan. 3) And the applier which executes the actions in an action plan. In Watcher these strategies are made to achieve different goals but these goals are primarily used for creating groups and providing descriptions while they contain no actual logic.

[1]All figures and other creative works will be licensed under CC-BY 4.0 under the attribution of Corne Lukken unless otherwise specified.

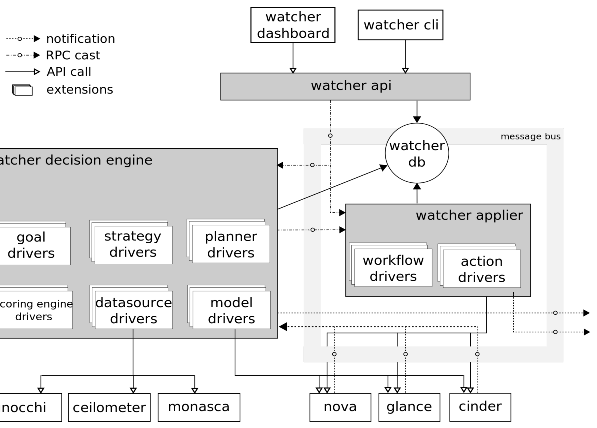


Fig. 2: Overview of Watcher's architecture[2]
CC-BY 3.0 OpenStack foundation

The architecture of Watcher follows a very typical pattern that is similar to many other OpenStack components. In this architecture functionality of the component is split across three different executables that communicate with each other using a message bus. In the case of Watcher this message bus is based on the Advanced Message Queuing Protocol (AQMP) [12] similar to many other OpenStack components. Similarly, the three executables share one central database.

These strategies use one of the available datasources to retrieve metrics such as cpu and memory utilization of compute nodes and instances. These metrics are used by the strategies to make decisions which are translated into actions put into an action plan. At this point the action plans are stored in the database and can be executed later. In addition to metrics strategies also rely on a data model that is build at runtime. that model contains all the information about compute nodes and instances which exist in the OpenStack infrastructure. This stores how many virtual cpus (vcpus) an instances has or the amount of memory a compute_node has. Additionally, information about storage clusters and the amount of available disc space is stored.

Information about compute nodes or instances is collected in data models using data model cluster collectors. Currently, Watcher has three different data models and cluster collectors being 1) ModelRoot, this model contains information about nodes for computing such as compute nodes and instances. 2) StorageModelRoot, that model contains information about pools and volumes for storage. 3) BareMetalModelRoot, this model contains information about bare_metal nodes. Baremetal is when a physical machine is provided as a resource without any virtualization layers. The term could be used interchangeable with *dedicated hosting* although that it is more commonly used when renting or leasing hardware and not applied to cloud infrastructures.

[2]This overview is not up-to-date and some elements are missing. These missing elements will be described later

When strategies are used to develop a new action plan this is done by launching an *audit*. OpenStack operators can either manually launch an audit or it can be scheduled based on a timer running with a configured interval. Within Watcher these types of audits are known as *oneshot* and *continuous*. These audits are run entirely in the decision-engine executable of Watcher and they are responsible for the first three steps of MAPE-K. In the same manner the applier executable is responsible for the last step in MAPE-K. When analyzing other code constructs within Watcher it becomes clear that these can also be mapped into MAPE-K.

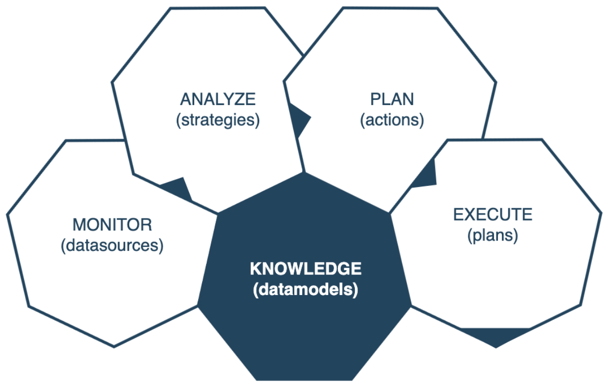


Fig. 3: Code constructs within Watcher can be mapped unto MAPE-K

The monitoring in MAPE-K is achieved with the datasources while the strategies provide the analysis. This in turn generate actions to go into a plan so that, finally, the plan can be executed. However, this leaves knowledge from MAPE-K as these four parts only cover MAPE. This knowledge is needed throughout the progression of the entire feedback loop and is provided by the various data models and their cluster collectors. Since new instances can be scheduled or deleted during the operation of Watcher these models are required to support being updated as soon as changes occur. To achieve this these models are updated at specific intervals which can be configured by users. More important, Watcher will update by listening for update notifications from other OpenStack components such as Nova.

1.5. Resource Optimization for CERN's Cloud

Such a large cloud as the one at CERN sees problems around resource optimizations in order to resolve this CERN wants to deploy Watcher. Unfortunately Watcher is not mature enough to be deployed and has many problems which must be resolved before deploying it. Once Watcher is be deployed it can be used in several ways that would reduce under utilization and resource contention.

The *oneshot* audit modes allows to perform a one time analysis using different strategies and results in an action plan which contains individual actions to be executed to achieve the optimizations. When Watcher is initially deployed the *oneshot* mode will be the most valuable as the individual actions can also have a negative impact on end users. The oneshot mode allows manual intervention which is needed to assure action plans contain logical and safe decisions. When Watcher has proven the ability to create safe and reliable action plans the *continuous* audit mode can be considered. The *continuous* audit mode allows to periodically perform audits which can also be automatically executed if desired.

2. Assignment

As described CERN aims to provide resource optimizations for their private cloud by deploying Watcher. Before this can be done problems with Watcher have to be identified, resolved and the solutions evaluated. The goal of this internship is to work with the upstream OpenStack community to develop Watcher so it can be used in CERN's private cloud.

For contributions to be valuable to the upstream community they need to be developed in generalized way so they can be used by other users. Proposed features and bug-fixes will have to be discussed within the community and the final result might differ from CERN's ideal solution. This way of developing adds to the overall complexity but makes it so that the developed software can benefit many users instead of being limited to internally at CERN.

Some initial tasks have already been established beforehand but many more are likely to be identified. Some of these tasks include developing a Grafana-proxy datasource and reducing the scope of audits. Additionally some initial tasks might require changes and be adapted after more careful analysis. Not all tasks are an absolute requirement to be able to deploy Watcher. This follows a typical software requirement paradigm of *must have*, *should have* and *nice to have* requirements.

2.1. *Research*

Parts of the assignment is broken down into a main research question and subquestions which are going to be covered in detail later. Throughout this work the main research question will be:

> How can Watcher be improved to be successfully deployed in CERN's OpenStack Cloud?

In addition the subquestions are as follows:

- How can be collaborated with the upstream community to develop Watcher?
- What prevents the deployment of Watcher in CERN's cloud?
- What possible solutions can be identified to be able to deploy Watcher?
- What can be evaluated from the final implementations developed to deploy Watcher?

The research methodology will be a combination of a literature and experimental study but will primarily be experimental as most of the research questions require performance evaluations to be answered. Some improvements might regard software patterns or design methodologies while others may regard resolving compatibility problems. The relevant domain knowledge to propose a solution will depend on the types of improvements that are identified.

2.2. *Structure*

This work is divided into sections one for each possible improvement of Watcher and each of these subquestions will be structured similar to a small report. These sections will describe the methodology used, results, evaluations and briefly discuss them. Following the sections the combined results will be used to answer the main question and to draw a conclusion to finalize with a discussion. However, the first two sections will be different and are used as a method to introduce the operation of the community and the methodology used to identify Watcher's potential improvements.

3. Collaborating with the Upstream Community to Develop Watcher

All OpenStack components are community driven projects that require contributors to discuss and collaborate using a large variety of platforms in order to progress the development of the components. To be able to join a community their methods have to be understood and evaluated as the methods of every community are different. In addition some communities might have difficulties collaborating that could be caused by a variety of different reasons. These difficulties should be addressed and resolved in order to improve the way the community is able to make progress.

3.1. *Method*

Having no prior knowledge to the governance and collaboration methodologies surrounding OpenStack projects the first step will be to gather an overview of all platforms containing information or providing services to OpenStack. Some of these are likely written specifically to Watcher while others are more generalized. The information gathered needs to be sufficient to answer a set of questions which are essential for collaborating on an open source project.

- Where is the source code hosted?
- Where are bugs tracked and reported?
- How are patched submitted and reviewed?
- How is a development environment configured?
- How are discussions held and is an agenda managed?
- How are development priorities maintained?

These questions were determined by past experience and intuition from work on previous open source projects or from other large open source projects such as Linux [13]. The information the questions are set to answer cover the most important aspects of a healthy open source project.

With all the relevant information gathered the current procedures of the Watcher project can be analyzed. The analysis should be thorough enough to identify any problems that might put the operation of a healthy open source project at risk. Typical indications of these problems could be. 1) Disputes between contributors not being resolved. 2) Inconsistencies in agreements such as meeting schedules and contributor guidelines. 3) Project deviations from central governance.

The analysis will result in a list of improvement points that will be attempted to be implemented to benefit the overall maturity of the Watcher project. The possible solutions to implement the improvements as well as their potential benefits and drawbacks and finally the implemented solution are described in the results.

3.2. Results

All the information was collected from a large set of online resources which has been made available in the appendix VI. During the period between February and August of 2019 some of the platforms used by OpenStack were migrated. These platforms include where source code was hosted and where patched are submitted and reviewed. Another migration took place during this period but it was started before February and not finished by the end of August. While one migration happened eventually overnight the other is taking more than six months to complete. The reasoning for this is arguably that one requires intervention from the contributors while the other is performed by a body from the OpenStack foundation. To simplify the results only the current platforms will be used but an overview of all migrations and their status is provided so the original platforms can be determined:

TABLE I: Platform migrations

Original platform	New platform	Status
git.openstack.org	opendev.org	complete
review.openstack.org	review.opendev.org	complete
launchpad.net	storyboard.openstack.org	ongoing

List of platform migrations and their status.

The source code for watcher is hosted on opendev.org and this repository is mirrored to github.com. Mirroring is a very typical practice for healthy open source projects. In addition review.opendev.org is used as patch & review system and also contains a mirror of the repository. The review.opendev.org patch & review system is based on gerrit [14] which is a relatively unknown but is based on several well known platform [15]. In addition to having platforms for source code and patches. Watcher tracks bugs on Launchpad [16] which is a platform maintained by Canonical primarily known for the Linux distribution Ubuntu [17]. The documentation for Watcher is hosted on MediaWiki hosted by OpenStack and contains the information on how to setup a development environment. Similarly, the meeting agenda is hosted on MediaWiki [18] and describes the used platforms with the relevant setup procedure. Overall the available platforms and how they are configured is proper and there are no concrete identifiable problems with their configuration.

When analyzing how these platforms are used by contributors of Watcher several problems become clear. The most prominent problem is that although a platform to organize

TABLE II: Platforms used by Watcher

Platform	Purpose
opendev.org	Hosting of source code
launchpad.org/Watcher	Bug tracker
review.opendev.org	Patch and review system
wiki.openstack.org/wiki/Watcher	Watcher specific documentation
docs.openstack.org/watcher/latest/	Watcher specific documentation
wiki.openstack.org/wiki/Watcher_Meeting_Agenda	Watcher meeting agenda
docs.openstack.org/releasenotes/watcher/	Release notes
specs.openstack.org/openstack/watcher-specs/	Feature specifications & priorities

List of platform and their purpose as used by Watcher.

meetings and maintain an agenda is available for all OpenStack components. Watcher has not organized meetings since December 2018 and several meetings before December seem to have very limited content [19], [20]. Other problems include that the members of the launchpad core drivers team [21] do not reflect the current developers and that the current set of core reviewers does not reflect active contributors.

3.2.1. Meetings: Resolving the collaborative problems around meetings was the most important issue to be addressed. To be in a position to suggest the community to organize meetings the community had to be familiar with the person making the suggestion. A typical method to become familiar with the community is by solving bugs or discovering them. To do this the bug tracker was used to identify open issues that could easily be solved by a new community member that had no prior experience in both Python or Watcher.

The first issue was resolved by solving a division by zero error in a method that computes the standard deviation of cpu usage from a list of nodes [3]. The original method did not account for the list being empty while it the standard deviation should always be zero if there are no elements to compute it over.

```
def get_sd(self, hosts, meter_name):
    """Get standard deviation among hosts by \
        specified meter"""
    mean = 0
    variation = 0
    num_hosts = len(hosts)
    if num_hosts == 0:
        return 0
    for host_id in hosts:
        mean += hosts[host_id][meter_name]
    mean /= num_hosts
    for host_id in hosts:
        variation += (hosts[host_id][meter_name] - \
            mean) ** 2
    variation /= num_hosts
    sd = math.sqrt(variation)
    return sd
```

The platforms for patching & reviewing can be used to determine the email addresses of contributors. These addresses are used to declare the intend for new features to be devel-

[3]nodes are also called compute nodes or hosts and these names are used to identify hypervisors containing virtual machines

oped for Watcher during the internship. These initial contacts provides insight in the contributors that are still helping developing Watcher and allowed to identify contributors that are now invested in other projects.

The insights in current contributors was used to suggest a new bi-weekly meeting format using similar channels and times as the previous meetings. This suggestion was sent on the 22nd of March 2019 for which the content is available in the appendix 12.

The reception of this suggestion was overly positive and the first new meeting was scheduled for the 10th of April 2019. These changes in meeting schedule were reflected on the documentation and communicated to the mailing lists of the OpenStack community [22]–[24].

During the first meeting several important problems that prevented Watcher from being deployed at CERN were discussed [25], [26]. Addressing these issues allowed for the development of the solutions some of which were done by other contributor. When developing open source projects it is important to understand that some features or improvements set out to be developed at CERN are developed by other contributors of Watcher. This is due to them observing the same issues and being similarly inclined in solving these. It would be rude or impolite to ask to not work on this features as a means to allow for better demonstration of skills and methodologies in a thesis. These issues and their appropriate details will be discussed in later sections.

3.2.2. *Core Reviewer:*

To organize the review process among other things most projects have a set of reviewers called core reviewers [27]. These core reviewers are given additional privileges to better orchestrate the collaborative progress contributors are able to make. The amount of core reviewers a project has varies but typically new reviewers are appointed by the project team lead(PTL). The primary role of core reviewers is to analyze incoming patches and provide feedback. This feedback can range in topic being related to grammar and formatting but also regards design methodologies and project scope.

The review process is done by up or down voting a patch. Both the up an down vote should come with a general message about why this decision was made. In the case of down voting the lines of code that led to the decision will typically be marked with an additional message. The difference between a normal review and those of core reviewers is that they have the ability to vote with -2 or +2 instead of -1 or +1. Depending on the project this might have different effects but in general a patch with -2 can never merge. Most projects require two +2 votes in order to be merged. However, the Watcher project only requires one +2 vote for the patch to merge.

Maintenance of the set of core reviewers on a project is a process that must be executed manually and is typically done by the PTL. The Watcher project has a large set of core reviewers for the given amount of contributors. Many of the core reviewers are inactive and have not reviewed patches in a long time. Eventually inactive core reviewers should be removed to better reflect the current set of contributors but it was more important to elect current contributors as core reviewers.

During the first Watcher meeting using the new schedule both myself and Chenker [28] were nominated as core reviewers by the current PTL Licanwei [29]. This privilege allows for better steering of the development efforts but comes with responsibilities and must not be taken lightly. In another meeting the large amount of inactive core reviewers was addressed. As a result the new policy would be to remove any core reviewer which is inactive for a one year period. This policy is best fitted because any attempts to contact inactive core reviewers was often met with no response.

3.2.3. *Watcher Drivers Team:*

In Launchpad projects are setup in such a way that they can be managed by individuals or teams. These teams can have members with different roles to orchestrate working together on a project. Members that manage projects can edit the status of blueprints or update issues for example. Since OpenStack components in some cases use Launchpad it is important that the team contains currently active developers.

As stated before, Launchpad is a bug tracker developed by Canonical as primary functionality but it has other capabilities as well. For instance, Launchpad can manage blueprints which track new features that are being developed for projects. In addition, Launchpad can be used as a repository for Debian based packages [30].

The Watcher project contains 36 members of which one is an active contributor, and furthermore, out of the three administrators none are active contributors. This is a potential problem since it limits the amount of contributors that can manage the project and could potentially lead to having no active contributors able to manage the project at all.

The current members of the Watcher Drivers team were contacted to resolve the issue on the 13th of June 2019 and it was resolved on the same day. Jean-Émile Dartois who is a previous contributor to Watcher responded to the correspondence and made the current PTL (Licanwei) an administrator. The correspondence is available in the appendix 12.

It was decided to now remove any inactive members from the Watcher Drivers team, however, the regain of control allowed to add new members to the team. As a result the new members could now manage blueprints and bug reports.

3.3. *Discussion*

The Watcher community embraced the suggestions to improve communication and collaboration efforts. Overall the community collaborates together in a mature and professional fashion despite the small size of the team. Furthermore, the community has shown to be welcoming to new contributors and provides good feedback about issues and patches. Of course in any community there are always ways to improve but especially considering the size Watcher's contributors should be proud of how they are operating.

It might be challenging to continue this momentum after the finalization of this internship and thesis so the community should invest in attempting to maintain it[4]. To achieve this it could make attempts to attract new contributors by creating demo's for example. Finally, this thesis itself could be shared with the community using the mailing list and this could potentially attract new contributors.

4. What Prevents the Deployment of Watcher in CERN's Cloud?

The first step in implementing improvements required to deploy Watcher is to identify them. This sections describes the method used to identify these improvements including other discoveries about Watcher.

4.1. *Method*

Initially there was very limited domain knowledge about the operation of Watcher and the structure of the source code. To identify problems this domain knowledge would need to be improved as well as knowledge on how Watcher is written. In addition, the available experience with the Python programming language and the tools available to it was also very limited and would need to be improved.

Since OpenStack is a large scale collection of open source projects it is very likely that a large amount of documentation is available for new contributors. This documentation will be analyzed to better understand the use of Python programming tools used by OpenStack and how components are structured. Documentation that needs to be found includes how to debug the programs and setup a development environment but also how to write and submit patches.

The available documentation should allow to setup a development environment which will be used to explore the way Watcher operates. The exploration of how Watcher operates could already help identifying problems but the main purpose is understanding how Watcher works internally.

With a working development environment available the source code will be analyzed. This analysis will be used with the predetermined tasks in mind to get initial ideas on how to perform these tasks. In addition, the analysis is used to further identify any additional improvements as well as gain further knowledge on how Watcher works internally.

In short the methodology to identify the improvements that are required before Watcher can be deployed.

- Gather existing documentation
- Configure test environment
- Explore Watcher's operation
- Analyze source code

4.2. *Results*

To gather documentation many of the sources that were identified during the work on collaborating with the upstream community could be used. In reality these tasks were executed largely in parallel of each other. However, the relevance of certain platforms differs significantly between these tasks. The most valuable sources of documentation are highlighted in table III.

[4]I do intend to keep collaborating on Watcher but my future courses can seriously limit the amount of time available.

TABLE III: Valuable OpenStack documentation

Source	Content
docs.openstack.org/contributors/	Contributor guide
docs.openstack.org/watcher/latest/contributor/devstack.html	Test environment
docs.openstack.org/watcher/latest/architecture.html	Watcher architecture

List of documentation sources relevant for identifying improvements.

The analysis of the documentation did not help identify any possible improvements, however, some necessary improvements were already described in the initial assignment. These improvements will be detailed in later sections.

4.2.1. Configure Test Environment: Setting up a test environment was a significant hurdle but one that is well known within the community as being challenging. In the documentation OpenStack describes different type of test environments and calls these environments devstack [31]. The most important difference between these types of environments is single-node and multi-node. In single-node devstack environments all configured components run on a single host. Naturally, in a multi-node devstack environment multi hosts are used to run all configured components and the hosts typically communicate over the network. Multi-node devstack environments can have many possible configurations in terms of which components are installed and the total amount of hosts. For clearness any multi-node devstack environments will be called teststack throughout this document.f

The flexible configuration devstack is able to provide is achieved using a configuration file known as *local.conf*. It is capable of making a selection of components to install which are known within the devstack configuration as plugins [32]. Furthermore, it allows to modify configuration for every component. Finally, devstack allows to select specific branches, commits or releases of components from any arbitrary git url in order for them to be installed.

The *local.conf* configuration used throughout this work is shown below.

```
[[local|localrc]]
enable_plugin watcher https://opendev.org/openstack\
    /watcher
enable_plugin watcher—dashboard https://opendev.org\
    /openstack/watcher—dashboard

enable_plugin ceilometer https://opendev.org/\
    openstack/ceilometer.git
CEILOMETER_BACKEND=gnocchi

enable_plugin aodh https://opendev.org/openstack/\
    aodh
enable_plugin panko https://opendev.org/openstack/\
    panko

[[post—config|$NOVA_CONF]]
[DEFAULT]
compute_monitors=cpu.virt_driver
```

The configuration installs a total of five plugins in addition to the default plugins. These default plugins are installed by devstack unless otherwise specified. The most important plugin is the *watcher* plugin which will install the Watcher component and is followed by the *watcher-dashboard*. Dashboard plugins are additions to the web interface OpenStack can offer which is called Horizon [33]. By installing a dashboard plugin the functionality offered by that component can then be used from the Horizon web interface. The next plugin is called Ceilometer and it is one of the three datasources Watcher supported before any development effort mentioned in this work. As mentioned before, Ceilometer is being deprecated as datasource. This is because Ceilometer used to be a complete solution for measuring, aggregating and storing metrics but the storage backend has since changed. The new storage backend is called Gnocchi and this is how the datasource is internally named in Watcher even thought the actual metering of metrics is still done by Ceilometer. By setting `CEILOMETER_BACKEND\ =gnocchi` the use of gnocchi as backend is ensured although this will likely become the default in a future release. The two final plugins *aodh* and *panko* are less relevant but their installation ensures the availability of certain metrics when using the Gnocchi datasource, however, it is likely that the availability of these metrics can be achieved as well by setting certain parameters in Ceilometer configuration files. Such a configuration change to enable additional metrics is also used in Nova's configuration `compute_monitors=cpu.virt_driver`. This configuration parameter enables the collecting of metrics from compute nodes (hypervisors) since by default only instances(virtual machines) are available.

The complete test environment running both Watcher and a supported datasource allows to test the operation of Watcher. The running test environment is used to explore how Watcher operates. This exploration step is done largely in parallel with the source code analysis. Since performing operations while analyzing how these operations are achieved in the source code is a very powerful method. This method will allow to quickly understand a program such as Watcher.This methodology has led to the discovery of several problems which must be resolved before Watcher can be deployed. These problems can be categorized into issues that must be resolved and issues that have benefits if resolved. Typically within the software development industries non essential features are called *nice to have*. This terminology of *must have* and *nice to have* is used throughout this document. Issues in both groups are ordered according to the time they were discovered starting with the problems that were discovered first.

4.3. *Discussion*

The operational and source code analyses were used to detail Watcher's architecture as shown in section 1.4. The architecture is a crucial section of the results but is only included in the introduction. This is done to have a better flow and improve readability throughout this work.

5. Grafana Datasource

As described before Watcher supports multiple datasources to retrieve metrics. While all current datasources are based on other OpenStack components at CERN none of these are actually used. This introduces the largest problem that prevents Watcher to be deployed in CERN's cloud since Watcher can not operate without a datasource to provide metrics.

At CERN multiple methods are used to gather metrics for monitoring and alarms. The most common method is to use the Collectd [34] service with various plugins on the compute nodes and have them stored in InfluxDB [35] or ElasticSearch [36]. These databases they can be accessed by a variety of tools but Grafana is used the most commonly used.

This additional datasource for Watcher was part of the original assignment and the necessity for this was known in advance of the operational & source code analysis. Because of this an alternative was not really discussed within CERN but some alternatives will be detailed as well as their benefits and drawbacks for completeness. This alternatives will primarily be based around different architectural solutions to realize this datasource.

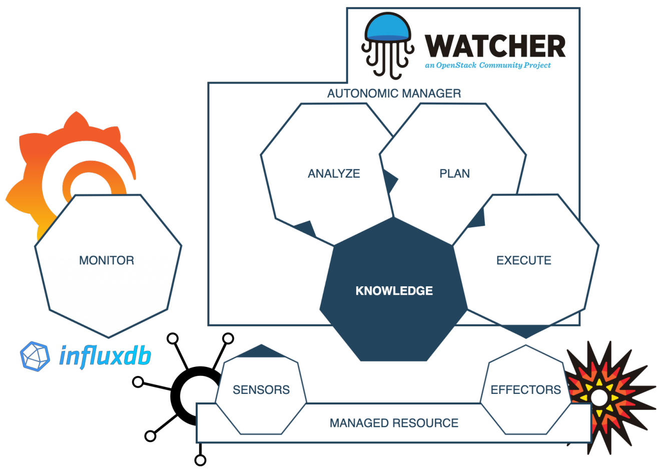


Fig. 4: Overview of services involved in Grafana datasource mapped unto MAPE-K.

When introducing these new services into the overview of MAPE-K their purpose is further clarified. Collectd is responsible for operating as sensors and will send this data to InfluxDB which operates as persistent time-series database. This database is read by Grafana which will operate as monitoring solution as a result Watcher can analyze, plan and execute. This execution is performed by communicating with Nova which is the effector.

Before the Grafana datasource could be designed several changes needed to be made to the interface between datasources and strategies. These changes are described in detail individually in section 6. The proposed solutions are based on how they have to be implemented after the required changes to the interface.

5.1. Configuration Flexibility

The solution for the Grafana datasource will have to allow for a large amount of flexibility in the configuration. This flexibility is necessary because their is no intermediate language or interface between Grafana and the databases it supports. As a result the function[5] required to retrieve metrics will depend on the database that is used. Furthermore, the type of database could be different depending on the metric or even use entirely different attributes to identify compute nodes.

Take the following database structures to emphasize this configuration complexity:

```
[
  {
    "type": "cpu_count",
    "host": "hostname.cern.ch",
    "value": "13"
  }
]
...
[
  {
    "type": "cpu_cycles",
    "uuid": "796eef99-65dd-4622-acca-fd2a1143faa6",
    "value": "24"
  }
]
```

This example shows two possible structural differences in the same type of database being InfluxDB. Nevertheless, both databases identify compute nodes using different attributes were one is using hostname and the other Universally Unique IDentifier (UUID). In addition the type identifier used is different while these flexibility requirements only become larger when different databases are used.

The desire to support this large amount flexibility might not be immediately apparent since it would be easier to fit it for the precise databases of CERN. However, the Watcher project is an open source collaboration so this flexibility is required to make it usable for other organizations as well. As an alternative the datasource could be developed in so called downstream / out-of-tree. This would mean that the code for this functionality is not submitted to the repository of the community. The downside of developing downstream functionality is that the community will not take it into account when creating new changes. As a result the communities development efforts might break the downstream functionality unexpectedly. Because of this downstream functionality is generally harder to maintain when working on open source projects.

[5]Throughout this work, function will be used to refer to a defined function / method in a programming language. When addressing a collection of functions generally method or methodology will be used.

CERN already contributes to many open source projects so it would better fit their nature to do so for the Grafana datasource as well. Combined with that the new datasource could benefit other OpenStack users and it would be easier to maintain. It is clear that the only logical conclusion is to develop a flexible upstream Grafana datasource.

To develop a flexible datasource a total of five configuration parameters need to be definable per metric. These are in addition to three parameters that only require one definition for all metrics. The following list is an overview of these five per metric parameters:

- project
- database
- translator
- attribute
- query

Because of the terminology used by Grafana and Watcher a lot of these terms conflict between one and the other. Up until now the definition used to describe databases was for the types of databases engines themselves such as InfluxDB. However, when using the Grafana API to talk to these engines this is done using a project id. For this reason the project parameter is used to refer to a specific instance of an database engine. These database engines support storing multiple databases at once. The database parameter is used to specify which database to select from within the database engine instance (project). Interfacing with Grafana is done through a REpresentational State Transfer (REST) API as a result many of the configuration parameters are put into the url:

```
.../{PROJECT}/query?db={DATABASE}&q={QUERY}
```

The two parameters that are not part of the url will be explained later. First, the configuration parameters that only have to be defined once are explained. These include 1) base url 2) authentication token 3) datasources 4) metric file. The base url is used to define the location of the endpoint to which requests should be made. This includes aspects such as the scheme and the path, although insecure scheme requests will be blocked. The terminology regarding url's is based on the Request For Comments (RFC) defined by Berners-Lee, Masinter and McCahill [37].

```
https://{BASE URL}/api/datasources/query/...
```

Contrary to many other parameters the authentication token is placed in the header of an HyperText Transfer Protocol (HTTP) request. This type of authentication token is known as a bearer token [38]. Although officially part of OAUTH2 [39] it is used in many other systems as a method for authentication. As an example, the entire header will look similar to the following:

```
Authorization: Bearer deadbeef=
accept-encoding: gzip, deflate
Connection: keep-alive
Accept: application/json
```

The other parameters are part of other developments that were made in parallel with the Grafana datasource. The datasources parameter allows to specify which datasources can be used and the preferred order. Additionally, the metric file parameter allows to specify the path to a YAML Ain't Markup Language [40] (YAML) file. This special type of configuration file is used to configure the available metrics per datasource. The benefits from such a YAML file are primarily improved readability.

The two remaining query and translator parameters are used to allow the Grafana datasource to work with different types of database engines (projects). Specific types of translators will be made one for each type of database. These translators will handle how to extract data from the result, how to construct requests and unit conversion. Finally, the query parameter is used to make the request with a syntax the database understands. With MySQL such a query could look like:

```
SELECT cpu_usage FROM cpu_metrics WHERE host LIKE \
    example AND time > now()—300s;
```

With these parameters it becomes clearer how exactly the Grafana datasource will be implemented. However, before going in to details alternatives to this implementation will be discussed. This is done because the solutions differ primarily on configuration parameters.

5.2. *Alternative Solutions*

In terms of alternative solutions for the Grafana datasource only one is clearly viable. This solution resolves around the underlying databases Grafana interfaces with. Since Grafana only offers a proxy to interface with these database it might be viable to interface with them directly. This would result in not a single datasource being developed but one for each type of database. The advantage would be that Grafana is no longer required for any infrastructure to be able to use these types of datasources. However, authentication, endpoints, queries and more will all have to be managed on a per database basis.

Ideally the solution would be able to use both Grafana and individual databases, however, due to time constraints this solution was not considered. The additional complexity would require a larger amount of effort but might be feasible in the future.

5.3. *Implemented Solution*

The implemented solution used the configuration parameters as described before, as a result the flexibility allows it to be used in any deployment. Because of this high flexibility it has since been merged into the upstream repository [41]. The implemented Grafana works by combining all the parameters together and make a request. The general process is enhanced by performing several validations as well as providing meaningful error messages.

The validation is done by iterating over all the configuration parameters and attempting to build a metric map. This map contains the names of all metrics and the values of parameters that go with them. Any not fully configured metric due to missing or invalid parameters will not be added. This validation of parameters is complicated by the YAML configuration source that was developed in parallel with Grafana. The Grafana datasource would have to be able to validate both sources while not raising warnings if only one is configured. In addition, a decision should be made if the YAML file overrides the normal configuration file or the other way around. Since the default configuration is contained in the traditional file it would require significant user intent to configure the YAML file. Because of this the YAML will override settings from the default configuration. The function to construct the Grafana metric map at runtime is shown in figure 5. This function only covers the handling of per metric parameters but this should be sufficient in demonstrating. Since the other validation is fairly basic such as verifying the parts of an url for instance.

```
def _build_metric_map(self):
    """Builds the metric map by reading config"""

    for key, value in CONF.graf.databases.items():
        try:
            project = CONF.graf.projects[key]
            attribute = CONF.graf.attributes[key]
            translator = CONF.graf.translators[key]
            query = CONF.graf.queries[key]
            if project is not None and \
               value is not None and \
               translator in TRANSLATOR_LIST and \
               query is not None:
                self.METRIC_MAP[key] = {
                    'db': value,
                    'project': project,
                    'attribute': attribute,
                    'translator': translator,
                    'query': query
                }
        except KeyError as e:
            LOG.error(e)
```

Fig. 5: Runtime construction of the Grafana metric map

Traditionally the metrics available for any given datasource are known before runtime of the program. With Grafana, however, this will no longer be true. Due to the availability of metrics being different among infrastructures the configuration will now determine which metrics are available. Clearly, this metric map can only be constructed at runtime. In Watcher the *DataSourceManager* ensures the construction of these metric maps, furthermore, it ensures the best fitting datasource is chosen given the current audit. This decision is based on the metrics that are required for any given strategy to run. Every strategy defines a set of metrics it will need to operate, so that upon the run of an audit the *DataSourceManager* can select a datasource that supports them. Several changes were needed to allow the *DataSourceManager* to work correctly with Grafana. In the following example code this behavior of dynamically constructing the map is shown:

```
 metric_map = OrderedDict([
    (Gnocchi.NAME, Gnocchi.METRIC_MAP),
    (Ceilometer.NAME, Ceilometer.METRIC_MAP),
    (Monasca.NAME, Monasca.METRIC_MAP),
    (Grafana.NAME, Grafana.METRIC_MAP),
])

def __init__(self, config=None, osc=None):
    ...
    self.metric_map[Grafana.NAME] = self.grafana.\
        METRIC_MAP

    map_path = CONF.watcher_decision_engine.\
        map_path
    metrics_file = self.load_metric_map(map_path)
    for ds, mp in self.metric_map.items():
        try:
            self.metric_map[ds].update(metrics_file\
                .get(ds, {}))
        except KeyError:
            msgargs = (ds, self.metric_map.keys())
            LOG.warning('Invalid Datasource: %s. \
                Allowed: %s ', *msgargs)

    self.datasources = self.config.datasources
```

Fig. 6: DataSourceManager constructor to dynamically create Grafana metric map while allowing YAML configuration to override it.

In figure 6 the construction of an OrderedDict can be observed. This special dictionary type ensures a specific order of datasources can be assumed as the default preferential order. This is due to how the standard dictionary in Python works. In these dictionaries no order can be assumed as a result any inserted element could end up at any location in the dictionary. Additionally, the copying of the runtime Grafana metric map into the *DataSourceManager* as well the processing of YAML configuration are shown.

Since the majority of CERN's metrics are stored in InfluxDB databases, naturally , the first translator to be developed for the Grafana datasource is for InfluxDB. The method for combining the parameters, creating requests and extracting results will be detailed using InfluxDB as example. When the function to retrieve a metric is called firstly the existence of this metric in the map is checked. This is done because a strategy can declare the necessity for a set of metrics but nothing limits it from requesting a not declared metric. After this check all data from the parameters is copied into a predefined datastructure for the translators to use. This data is passed along while constructing the translator, additionally, the base class of the translators performs basic validation. The translator is called to construct the parameters for the HTTP request using a function inherited from the base class. The actual request, the error handling and recovery around it remains responsibility of the Grafana datasource itself. Finally, the translator uses the result from the request to extract and possibly convert it so that it can be returned to the strategy. The majority of this behavior is demonstrated in figure 7.

```
...
data = self._build_translator_schema(
    meter_name, db, attribute, query,
    resource, resource_type, period,
    aggregate, granularity)
translator = self._get_translator(trans_name, data)
params = translator.build_params()

r_args = dict(
    params=params,
    project_id=project,
)
args = {k: v for k, v in r_args.items() if k and v}

resp = self.query_retry(self._request, **args)
result = translator.extract_result(resp.content)
return result
```

Fig. 7: Demonstration of how database specific operations are abstracted into translators.

With InfluxDB some aspects of the schema have to be converted to make a valid query. This is due to how InfluxDB optimizes the time series database. In these databases the retention period for retrieved data can be defined. As a result a query with a long period might not be able to retrieve all historical data if the retention period is not adjusted accordingly [42]. The available retention periods depend on the InfluxDB instance and therefor, they are configurable parameters in Watcher. These retention periods are then exposed to the query so that they can be used. Other parts of the schema need to have data extracted instead of converted. This is necessary for the resource object that is part of the data schema as shown in figure 7.

These resource objects are part of the data model that is build when an audit is launched. To demonstrate how the attribute parameters works an example of this resource object is shown in figure 8. In reality there are several types of resource objects and the types can depend on the strategy or the data model cluster collector. Furthermore, many of these resource object types share some attributes.

```
service = node.service
node_attributes = {
    "id": node.id,
    "uuid": service["host"],
    "hostname": node.hypervisor_hostname,
    "memory": node.memory_mb,
    "disk": node.free_disk_gb,
    "disk_capacity": node.local_gb,
    "vcpus": node.vcpus,
    "state": node.state,
    "status": node.status,
    "dis_reason": service["disabled_reason"]}
```

Fig. 8: How the resource object for compute nodes are constructed.

Using the value from the attribute configuration parameter one of these fields from the resource object can be extracted. This is done using a special function in Python called:

`getattr(object, attribute_name)`.

With all these attributes ready they can now be exposed to the query. The exposing of these attributes is done in a comparable manner to how Structured Query Language (SQL) prepared statements [43] work. The configuration allows to define a string with special markers while each of these markers will be replaced with an attribute. This functionality is achieved using Python's builtin string function `format()`. Take the following unformatted query as shown in figure 9. Any of the numbered brackets such as `{0}` and `{1}`. Which of these numbered brackets will be replaced with what attribute is thoroughly documented [44].

```
SELECT 100—{0}("{0}_value") FROM {3}.cpu_percent
WHERE ("host" =~ /^{1}$/ AND "type_instance"
=~/^idle$/ AND time > now()—{2}s)
```

Fig. 9: Unformatted InfluxDB query example.

Currently, there are five exposed attributes for each query. These attributes are named from zero to four. Each of these represents • {0}, represents the aggregate such as *mean* or *min*, • {1}, is the value as extracted from the resource object based on the specified attribute. • {2}, is the period in seconds over which data should be aggregated. • {3}, is the granularity between data points over which should be aggregated. • {4}, is translator specific which in the case of InfluxDB will be the retention periods.

The following will demonstrate how the final query for InfluxDB would look taking the example raw query from figure 9. The placement of all these exposed attributes can now be observed in figure 10. Looking at the structure of these formatted queries it might seem intimidating to create these. However, tools such as postman [45] exist to develop interactions with REST API's. The recommendation to use such tools is also emphasized by the online documentation [44].

```
SELECT 100—mean("mean_value") FROM one_week.
cpu_percent WHERE ("host" =~ /^examplehost\
.cern\.ch$/ AND "type_instance"=~/^idle$/ AND
time > (now()—120s))
```

Fig. 10: Formatted InfluxDB query example.

With the complexity of all these attributes being combined into queries for use on different types of databases, naturally, a thorough guide is necessary to help users configure the Grafana datasource. Similarly to this work explaining the available parameters, attributes and how they are converted in the query. The online documentation covers all these aspects with similar examples to demonstrate how it is put all together.

With all the configuration in place, the query ready and the request made the final part of the operation is to extract the result. Once more, the precise method of extraction is based on the type of database in Grafana. In the case of InfluxDB a JSON structure will be returned. However, since JSON structures are not strictly defined expected structures are part of the documentation as well. The result shown in figure 11 is an response from the query as shown in figure 10.

```
{"results": [{
  "statement_id": 0,
  "series": [{
      "name": "cpu_percent",
      "columns": [
        "time",
        "mean"
      ],
      "values": [[
        1563791841338,
        6.188323840776562
      ]]
    }]
}]}
```

Fig. 11: InfluxDB response when querying with aggregate.

In the case of these aggregated InfluxDB queries it is clear that the value from the column with the name of the type of aggregate should be selected. In the response from figure 11 this would be *6.18832...*. The documentation of the formats expected from a response are critical as without an aggregate they would look similar to figure 12.

```
{"results": [{
  "statement_id": 0,
  "series": [{
    "name": "cpu_percent",
    "tags": {
      "host": "examplehost.cern.ch"
    },
    "columns": [
      "time",
      "mean_value"
    ],
    "values": [
      [
        1563781649000,
        2
      ],
      [
        1563781769000,
        1
      ],
      [
        1563781829000,
        1
      ],
      ...
```

Fig. 12: InfluxDB response when querying without aggregate.

The extraction of the results becomes significantly more complicated when the aggregation of data needs to be handled by the Grafana datasource. It should be noted that for other types of databases this might be required if the database is not capable of aggregating data itself. With InfluxDB the aggregation requires fairly minimal additions to the query, additionally, since InfluxDB is a time series database it should be efficient at performing mathematic operations over periods of time.

The queries for InfluxDB need to be aggregated to have their results extracted since this a minimal change to the query. The actual extraction itself is done with minimal Python code as shown in figure 13. This example only highlights the necessary for extraction. Other parts such as error handling or documentation are omitted[6].

[6]The same applies to any other code example but in this case the amount of omitted code is very substantial.

```
def extract_result(self, raw_results):
    try:
        rslt = jsonutils.loads(raw_results)
        rslt = rslt['results'][0]['series'][0]
        aggregate = rslt['columns'].index(
            _data['aggregate'])
        return rslt['values'][0][aggregate]
    except KeyError:
        ...
```

Fig. 13: Extracting result from InfluxDB response.

After the extraction by the translator the result is returned to Grafana which in turns returns it to the running strategy. The overall operation of the Grafana datasource might seem complex but this additionally complexity allows for extensive flexibility. In addition, this complexity will reduce the development effort required for any additional database type supported by Grafana. Considering both the upstream community and CERN's desires for such a datasource this given architecture will provide a solid foundation for any further improvements.

5.4. Discussion

While overall the Grafana datasource architecture should be a solid foundation. There were several issues which were not addressed mainly due to time constraints. As part of the discussion all these possible improvements will be briefly detailed.

As mentioned before ideally translators should be datasources by themselves. This could be achieved by inheriting both base classes from the translators and the datasource. Python supports the concept of multiple inheritance similar to how C++ supports it. Multiple base classes can be inherited by specifying them in comma separated fashion:

```
class Example(BaseClassOne, BaseClassTwo):
```

Alternatively, both could be developed separately but it is likely that if they would be developed into one class a lot of code could be shared across functions.

Even though effort was put into correctly managing both sources of configuration there still remains a small issue when configuring using YAML. Due to the logging functionality when first constructing the metric map an error will be logged that no metrics are available. However, it does not affect the functionality of the Grafana datasource. A bug report was opened on Launchpad to ensure this issue will not be forgotten [46]. Potentially, future CERN personal to continue the work on Watcher could take this as an issue to get familiar with OpenStack and Watcher.

As final improvements the exposed query parameters could use names instead of numbers which make the raw queries more readable. To achieve this a small wrapper around the `format()` call could be made that would replace all occurrences of any names back into their corresponding numbers. Using builtin Python string functions to achieve this should allow to implement this with relatively little effort. As an example figure 14 shows how this would improve the readability of raw queries.

```
SELECT 100—{aggregate}("{aggregate}_value") FROM
{retention_period}.cpu_percent WHERE ("host"
=~ /^{attribute}$/ AND "type_instance" =~/^idle$/
AND time > now()—{period}s)
```

Fig. 14: Example of how improved queries could look like.

```
class MonascaHelper(base.DataSourceBase):

    def statistic_aggregation(self, id, dimension):
        """
        :param id: unused
        :param dimension: dimension(dict)
        ...
        """
        ...

class GnocchiHelper(base.DataSourceBase):

    def statistic_aggregation(self, id, dimension):
        """
        :param id: id of resource
        :param dimension: unused
        ...
        """
        ...
```

Fig. 15: Highlight of one of the interface problems introduced due to supporting multiple methodologies in the datasources.

6. Standardized Datasource Interface

As mentioned Watcher supports multiple datasources, however, support for these datasources was added one after the other. Initially there was only a single datasource. Overtime with the introduction of more datasource the need for a base class became clear. Such a base class will allow to standardize behavior and interface across datasources. This base class was already implement in Watcher before the analysis but during the source code analysis problems with the implementation of this base class became apparent.

There were multiple datasources when the base class was developed but these datasources both were developed in different ways. To make the base class compatible with both datasources the base class was designed to support both designs of the different datasources. Unfortunately this leads to the strategies being able to work with both methodologies of the datasources as a result strategies still only work with some types of datasources even though the datasources are developed using a base class. An example of these multiple methodologies is shown in figure 15. Imagine two different strategies one calling the function without the *dimensions* parameter supplied while the other does not supply the *id* parameter. Depending on the current datasource being used this might raise an error terminating the execution of the strategy. This situation occurs in multiple strategies because they were created before the datasource base class.

To resolve this issues the base class for the datasources needs to be redesigned and all strategies need to be updated to interface with the datasources using the new method. This would be a significant change but a very important one. It would allow all the strategies to work with each current and any future datasource. The need for this new interface was discovered during the development of the Grafana datasource. Since a datasource that could only be used on very specific strategies would have limited value. The need for this improvement was discussed during the bi-weekly meeting were other contributors also acknowledged the need for this improvement.

Like many other large features a specification was written [47] to define the changes to be made. Naturally, the defining of these changes was an iterative process. In addition, due to the size of this change it took significant effort. To demonstrate this effort the most important changes will be detailed as well as their benefits.

6.1. Required Changes

In datasources metrics are identified using so called metric maps. These dictionaries contain key value pairs to identify what type of metrics are supported by any given datasource. The key value identifies the type of metric while the value identifies the internal name as recognized by the datasource. Strategies would request metrics by specifying the values of the metric they want, naturally these internal names could differ per type of datasource. This would cause the strategy to not work with certain datasources even though they support the same type of metric. An example for these unnecessary forms of incompatibility is shown in figure 16.

```
class GnocchiHelper(base.DataSourceBase):

    NAME = 'gnocchi'
    METRIC_MAP = dict(
        host_cpu_usage='compute.node.cpu.percent',
        host_ram_usage='hardware.memory.used',
        ...

class MonascaHelper(base.DataSourceBase):

    NAME = 'monasca'
    METRIC_MAP = dict(
        host_cpu_usage='cpu.percent',
        host_ram_usage='vm.memory.utilization',
        ...
```

Fig. 16: Metric map key value discrepancy

To resolve these incompatibilities the keys of these dictionaries are now used as identifiers instead. The *METRIC_MAP* is only used in one function but this function is responsible for retrieving results. In the base class this function is known as `statistic_aggregation()` which needed several changes during the development of this new interface. The other changes to this function were the removal of two parameters *group_by* and *dimensions*. Any current datasource that used one or both of these parameters was rewritten to no longer need them. The removal of these parameters was possible due to necessary information already being available in different parameters that were unused by these specific datasources. Another change to the parameters of this function was increasing the amount of data passed about the resource to retrieve metrics from. Instead of passing just the UUID an entire object with many attributes of relevant information is passed. Typically, datasources would use the UUID to make additional API calls to Nova to retrieve attributes like the hostname. These attributes were previously collected during the building of the data model so these calls could easily be prevented. In addition, an *resource_type* parameter is added to indicate the type of resource object being passed. This is because the data model knows many different types of objects for instances, compute nodes or volume pools, for instance. The resulting signature of `statistic_aggregation()` is shown in figure 17.

```
def statistic_aggregation(
    self, resource=None, resource_type=None,
    meter_name=None, period=300, aggregate='mean',
    granularity=300):
```

Fig. 17: Datasource signature of statistic_aggregation

The final change to parameters of `statistic_aggregation()` limited the amount of possible some parameters are allowed to have. Both the *resource_type* and *aggregate* parameters were limited in this way. Before these changes the value *mean* and *avg* where used arbitrarily by strategies for the aggregate parameter. Because of this many datasources needed to check the value of the aggregate parameter. Possibly replacing *avg* with *mean* or the other way around. The new interface specification defines a set of allowed values that can be extended if necessary. Now only a limited number of datasources needs to replace the value of the aggregate parameter in some cases. The set of allowed values is detailed below:

```
AGGREGATES = ['mean', 'min', 'max', 'count']
```

All expected data types for parameters and return types have been documented extensively in the base class. This should prevent a similar situation from occurring again, as well as, reduce the complexity of developing new datasources. In this documentation the data types and values for return types are especially important. If these values were to be of the incorrect unit when returned it would affect the operation of the strategies. An example of these forms of documentation is shown in figure 18.

```
def get_host_cpu_usage(
    self, resource, period, aggregate,
    granularity=None):
    """Get the cpu usage for a host

    :return: cpu usage as float ranging between
             0 and 100 representing the total
             cpu usage as percentage
    """
    pass
```

Fig. 18: Defining of value and type for return type.

With the new interfaces completed some small additional improvements were discovered. Since this improvements is still related to the datasource interface it is included in this section. As some of the datasources used a special function to reattempt any requests that had failed. Especially in larger infrastructures this is a very useful feature as failed requests become more likely. Any consecutive requests after one fails could still succeed depending on the error that caused it. When the request fails for a configurable amount of times consecutively an error is raised. This functionality was abstracted from the datasources that support it into the base class. Finally, the function was generalized while exposing an abstract function to allow for recovery operations. The operation of this functionality is demonstrated in figure 19. A powerful feature in Python is the ability to pass a function and subsequent arguments to this function as parameters. The functionality uses the feature so that arbitrary functions can be wrapped for reattempting upon failure. Additionally, the *query_retry_reset()* function allows to perform datasource specific recovery operations upon failures.

```
def query_retry(self, f, *args, **kwargs):
    ...
    for i in range(num_retries):
        try:
            return f(*args, **kwargs)
        except Exception as e:
            LOG.exception(e)
            self.query_retry_reset(e)
            ...
            time.sleep(timeout)
    raise exception.DataSourceNotAvailable(
        datasource=self.NAME)
```

Fig. 19: Function to recover after failure from external request.

6.2. Discussion

The completion of the improved datasource interface allowed to continue the development of the Grafana datasource. Possible one of the reasons the Grafana datasource was only merged on the 12th of July was due complications like this. The new interface allows all datasources including Grafana to be used with any strategy given it provides the necessary metrics. The correct operation of different datasource was subsequently tested in both test and production environments.

Nevertheless, there are still possible improvements to be made to the datasource interface. Such as the method used to document return types and values. The Sphinx [48] tool used to generate documentation using so called *docstrings* offers extensive functionality. Currently a lot of these features are not used because of the way the docstrings are written. It would improve the generated documentation significantly if the functionality of Sphinx was used properly. The resulting documentation would more clearly indicate types or even possible exceptions.

7. Data Model Scope

As mentioned previously Watcher was designed around the MAPE-K feedback loop which operates with a central body of *knowledge*. In Watcher this knowledge is built into a data model to store a representation of the infrastructure in memory. However, while analyzing the source code it became apparent that the way this data model was constructed needed to be improved.

Watcher constructs its data model by retrieving all compute nodes and instances in the entire infrastructure[7] then it applies a scope afterwards by removing any elements that do not match specific filters. This method for constructing the data model is very problematic for large scale clouds such as that of CERN with over 9000 compute nodes and 33000 instances. In these clouds the total time required to gather all information will be extremely long. Furthermore, the load produced by all these API requests will have significant impact on the operation of the services. The requests might even completely stop some services from functioning due to Out Of Memory (OOM) exceptions for instance. An OOM exceptions occurs when the program uses so much RAM that the host has insufficient memory to continue execution of the program. Typically the operating system will kill the program in order to prevent the entire host and all its programs from freezing.

compute nodes and instances are typically divided into groups called cells this allows to more easily manage the infrastructure. Scheduled maintenance or decommissioning can be organized on a per cell basis which makes it much easier for everyone involved to identify the machines on which actions are to be performed. Internally in OpenStack these cells are known as aggregates while the OpenStack users are provided availability zones. These availability zones map to one or more aggregates. Consequently, these aggregates and availability zones are the filters applied by the scope to remove non matching elements from the data model.

These cells are an important construct since it is desirable to limit the scope of audits to a single cell. This allows to prevent moving instances between different availability zones as users have likely configured such isolations for redundancy. Limiting the scope to a single cell also prevents performing operations on parts of the infrastructure that might be scheduled for decommissioning. Finally, using single cells provides easier insight into why certain strategies have led to a set of recommended actions.

In Watcher audits can be created in advance for later use using so called *audittemplates*. These templates allow to store meta information such as the strategy to use and its associated goal. More important, however, these audittemplates support the scopes. These scopes will allow to specify which aggregates and availability zones to use when executing the audit. Moreover, these scopes allow to exclude specific compute nodes and instances as well. Finally, these scopes can be specified per type of data model such as storage or compute. Finally, the scopes are defined using YAML or JavaScript Object Notation [49] (JSON) files.

Since the concept of scopes is already well in place using the audittemplate feature the most obvious solution would be to apply the scope before gathering data about compute nodes and instances. This would limit the calls made to external APIs to only make requests for compute nodes and instances as defined. Of course some additional API calls are needed to gather information about which compute nodes belong to certain aggregates and availability zones. This should greatly outweigh the impact of gathering all available information. The data model scopes(audit scope) are already readily available to audittemplates. Consequently, it should be relatively simple to change the order in which this scope is applied to the model.

[7]The retrieval of this information is limited to so called regions which are the strongest form of isolation in OpenStack clouds.

An alternative would be to have a continuously updated data model that is constructed on a piece by piece basis by retrieving information from a single aggregate and adding it to the overall data model. The execution of audits would then have to be blocked until all information that the audit needs is contained in the data model. This approach could require significantly more memory as it would always have a data model with the complete infrastructure. Although it would more easily allow multiple audits to run in parallel. When multiple audits are being run in parallel using the same data model instead of having a data model per audit it could eventually outperform the current solution. This applies only if it was common to run multiple audits in parallel. This solution would require significant changes to Watcher not only in the way the data model is build but also in improving the threadpool to run multiple audits in parallel.

The scalability issues introduced due to the way the data model was build was discussed with the community via email and Licanwei proposed to implement this by changing the order in which the data model scope is applied to the model. As typical for the methods used to collaborate in Watcher a spec was written [50] to describe how the feature would be implemented followed by the implementation itself [51].

7.1. Performance Measurements

The entire CERN OpenStack cloud consists of roughly 70 cells but each cell has significant differences in the amount of compute nodes and instances in it. It is not possible to do measurements with all 70 cells of the cloud due to the impact that would have on its operation. After careful discussions it was decided to take measurements up to five cells consecutively. Because of the limited amount of cells the choice of cells could significantly affect the estimate for all 70 cells. The cells were chosen based on the amount of compute nodes and instances so that the five cells would accurately represent all 70 cells. All relevant information about the chosen cells is shown in table IV.

TABLE IV: Selected Nova cells to use in performance measurements

Name	ID	compute nodes	Instances
gva_shared_002	60	136	546 - 670
gva_shared_003	61	164	1785
gva_shared_009	65	123	1640
gva_shared_020	99	44	351 - 354
gva_project_o48	102	56	11 - 157

Overview of Nova cells used in performance measurements.

The measurements from these five cells will be used to estimate the build time for all 70 cells. This estimate allows to evaluate the achieved performance improvement from implementing the data model scope. Since the scope will allow to only gather data from a single cell instead of the entire infrastructure. Coincidentally, CERN aims to start using Watcher with limited scopes most likely single cells. To achieve these estimates a thorough well defined methodology is needed to achieve a high accuracy. Firstly, certain functions of Watcher will be adapted to perform the performance measurements. As it is important to ensure that only the build time for the data model is measured. With the measurements Python will be used to estimate the total build time for all 70 cells. However, it might be difficult to make reasonable estimates for such a much larger infrastructure. To maximize the accuracy of the estimate many measurements for the same cells will be taken. Still, the operation of infrastructure must be taken into account. To attempt to proof the accuracy of the predictions statistical analysis will be used. In this analysis the null hypothesis will be that there is no correlation between the measured and estimate data. Linear regression will be used to test this hypothesis. Finally, the results from both the measurements and the predictions will be shown.

To perform these measurements the *execution* function of the Nova cluster collector was modified. This allowed to only measure the time taken to build the data model without taking other operations into the measurement. In addition the modifications are used to output important information that allows to validate the measurements. Afterwards the further execution of the process is stopped by letting the python debugger set the trace since only the measurements are of interest. The implementation of this measurement technique is shown in figure 20.

```python
import time
LOG.warning("Measurements started...")
start = time.time()
# Data model is build here...
end = time.time()
LOG.warning("Total time: {0}".format(end - start))
LOG.warning("Total compute nodes: {0}".format(len(\
    self.model.get_all_compute_nodes())))
LOG.warning("Total instances: {0}".format(len(self.\
    model.get_all_instances())))
import pdb; pdb.set_trace()
```

Fig. 20: Implemented performance measurements and logging in Nova cluster collector execute function.

As mentioned the amount of measurements taken will be limited due to the impact it might have on the infrastructure. Still, to improve the estimates many different combinations of cells were measured as well as single cells. Single cells were measured five times as the are typically smaller resulting in a smaller impact. While combinations of cells were measured twice. Despite that the extend of the measurements might seem limited this still results in over 60 individual measurements. All of these measurements are available in the appendix 12, 12. Additionally, git can be used to checkout this specific version of Watcher by using the commit hash *36c2095254a24cbcfa7bc11b8bd453de0f659c5a* using the command:[8]

```
git checkout 36c2095254a24cbcfa7bc11b8bd453de0f659c5a
```

To make an estimate the Numpy library was used with the *polyfit* function [54]. This function computes a polynomial limited to a specifiable degrees of freedom. Furthermore, the computed polynomial will have a minimized squared error to best fit the data. Internally the function shown below is used to minimize the squared error:

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2$$

To evaluate the generated polynomial another part of the Numpy library will be used. This function called `linregress\()` allows to compute linear regression so that the correlation between the measured and polynomial data can be proven. Since the polynomial is computed by minimizing the squared error this should always be true. Nonetheless, this step is performed for completeness and as validation.

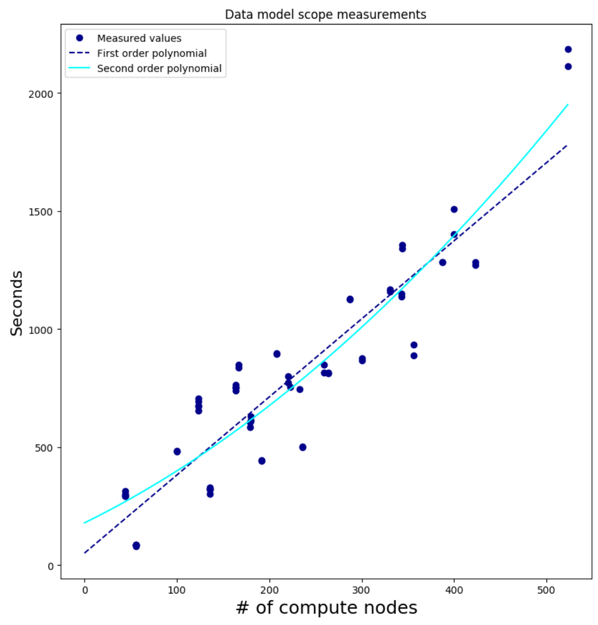


Fig. 21: Measured time to build data model per number of compute nodes.

With the methodology described the measured results can now be detailed. These result will not yet include any estimates for higher number of cell counts. In figure 21 the X axis shows the amount of compute nodes used in the measurement and the Y axis shows the required time to build the data model in seconds. Compute nodes are used for the X axis as to better scale the graph, additionally, allowing to easily distinguish between different cell combinations. The two curves drawn into the diagram depict the start of the estimates. In dark blue on a dotted line the 1^{st} order polynomial is shown while the 2^{nd} order polynomial is shown in cyan on a solid line. The difference between the first and second order polynomials is their degrees of freedom. Effectively, the degrees of freedom limit the curvature of line in between points. A first order polynomial will be a straight linear line while a second order polynomial can be quadratic. Nevertheless, this freedom is only applied while minimizing the squared error so it does not necessary result in a sharper curve. However, higher order polynomials should be used carefully as they have a high change of overfitting. As a result only the first and second order polynomials are used to make estimates in this case.

Some important aspects of the measurements can not easily be shown in figures such as 21. For instance, the mean time to build the data model for a single cell is 427 seconds. When normalized per compute node this is 4.16 seconds. Taking these means to estimate the build time for all 70 cells could have very poor results. This is because having multiple cells in the scope has an unknown effect on overall build time. As a result collections of cells could have lower or higher means normalized per compute node. Finally, The original data for all measurements can be found in the appendix 12.

Before making any estimates the polynomials need to be validated. The validation is performed by using linear regression on the measured values and the estimated values. A scatter plot visualizing these used data is shown in figure 22. In the figure it appears that there is a strong correlation between the measured and estimate data. As mentioned linear regression will be used to test the null hypothesis that there is no correlation. In addition a confidence interval of 99% will be used since the estimate will be used to predict for a much larger infrastructure. The strict confidence interval should help ensure the estimate is highly accurate, while still estimating for a roughly 14 times larger infrastructure than measured.

[8]The repository can be found online [52] similarly to a guide on using git [53].

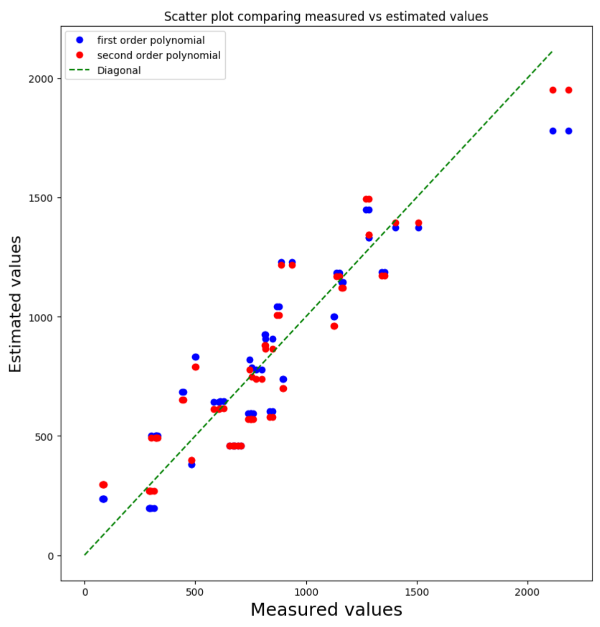


Fig. 22: Scatterplot comparing measured and estimated values.

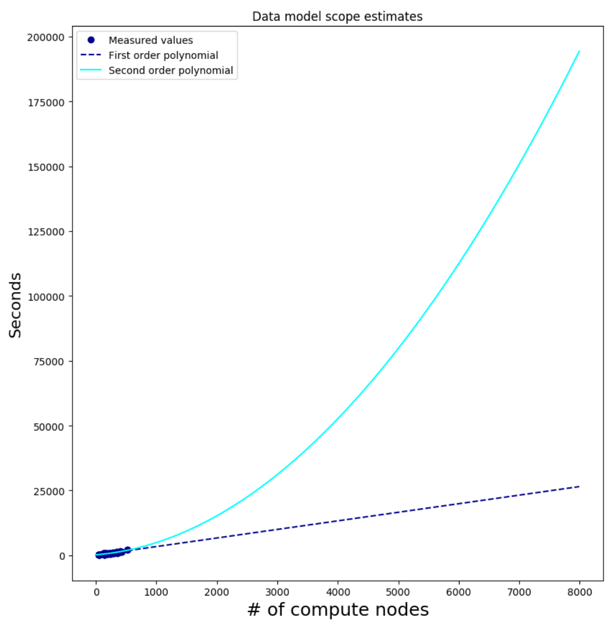


Fig. 23: Estimated data model build times for all 7991 compute nodes.

The P-value for 1st and 2nd order polynomials are $8.94e-28$ and $7.34e-29$ respectively. Given the 99% confidence interval a P-value of 0.01 or lower is required. Since $8.94e-28$ and $7.34e-29$ are well below 0.01 the null hypothesis is rejected in both cases. To conclude, a correlation between the measured and estimated data can be assumed with 99% confidence. Which of the two polynomials is a better estimate can not be concluded. This is a common misconception in statistics as a lower P-value does not indicate a better estimate. A lower P-value can only be used to infer the hypothesis with a higher degree of confidence while the confidence interval should always be chosen before testing a hypothesis not after. Nevertheless, these polynomials are statistically significant and can now be used to estimate the data model build times for all 70 cells.

Before the values for all 70 cells can be estimated the amount of compute nodes needs to be known. The amount of compute nodes depends on the measurement system queried to determine this value, moreover, this value tends to fluctuate slightly over time. Any value between 7500 and 9500 will be a reasonable estimate. however, to more precisely determine the true value the Grafana monitoring solution was used on the 26th of July 2019. In total there were 7991 compute nodes actively shown by Grafana on that day. The results of the estimates for this number of compute nodes is shown in figure 23.

These estimates results in a value between 26471.63 and 194303.99. Although the variance is quite high both should be reasonable predictions, however, since these values will be used to determine the performance improvement the lowest estimate is used for fairness. From the five measured cells the mean time to build the data model was 427 seconds, as a result the data model scope provides a 62 times better estimated performance. Overall the data model scope works as expected and is likely to provide a very substantial performance improvement. The data model build time of 427 seconds is low and enough that Watcher can be deployed in CERN's infrastructure with this performance.

7.2. Discussion

Unlike previous section of this work not many examples of the code were detailed. Neither were many architecture decisions around the reimplementation of the data model scope discussed. This is due to the changes being made by another contributor [29] while the source code analysis was still being performed. Nevertheless was this change reviewed by other contributors and has it since been tested in production successfully as demonstrated by the performance measurements. Before fully usable, however, some small fixes were necessary [55].

Although the functionality of the new data model scope works very well the architecture could be improved. Because of the existing method for applying scopes together with time constraints to implement this feature. The resulting implementation has many pieces of functionality copied over from the scope classes into the cluster model collector. Ideally, the classes used to apply the scope should be used by the cluster data model collectors to facilitate the scope. This is a good improvement to investigate implementing in the future as it would improve the maintainability of Watcher.

Another possible future improvement would be implementing the new methodology to apply the scope to other cluster model collectors. Currently, this new method to apply the scope is only used when the model is build for Compute. With the Storage and BareMetal models the old method is currently left unchanged. This improvement can be made in parallel with the improvements to the scope classes their responsibilities.

As mentioned using the mean to estimate the build time could have poor results. To demonstrate the same scatter plot as shown in figure 22 is made. In this figure the measured values will be compared with the estimate from the mean based on $4.16 * x$. The results are shown in figure 24. They show that there still is a clear linear relation between the estimated and measured values, however, the error between the estimated and measured value is now orders of magnitude larger than with the polynomials. As a result these estimates would be far less suitable for predicting the total build time for all 7991 compute nodes.

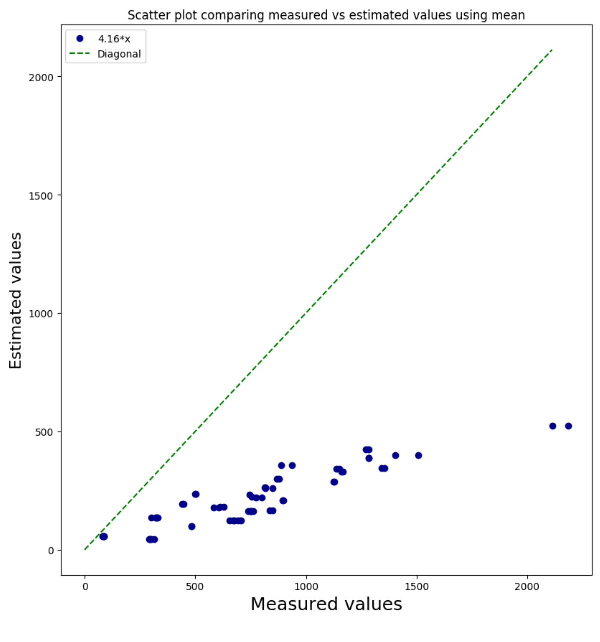


Fig. 24: Scatterplot comparing measured and estimated values using a simple mean estimate.

8. Nova API Call Optimizations

The OpenStack compute component Nova is Watcher's primary effector. It is used to move instances across compute nodes but also supplies the information to construct Watcher's data models. However, Nova is a core component of OpenStack and the API is frequently used by many different components not just Watcher. In CERN's cloud many instances of the Nova API are run in parallel with a load-balancer. This is necessary to be able to handle the load of all users and components. It is desirable to keep this number of parallel instances as low as possible while maintaining good performance since it would allow a larger part of the infrastructure to be used for its main purpose. To maintain a low number of Nova API instances Watcher should use the Nova API effectively. In addition, the efficient use of Nova API is also very likely to have performance benefits.

The ineffective use of the Nova API was discovered by another contributor named Matt [56] who became an active member in Watcher's community in May. This period in May marked the next release of Watcher which is known as Stein, similarly, this name is the same for other OpenStack components that follow the same six month release cycle. Even though the discovery and implementation of these optimizations were performed by another contributor it remains important to include it in this work. For these optimizations CERN's cloud infrastructure will be used as a platform to evaluate and measure these improvements as well as possibly suggest additional improvements. The same methodology will be used as previously detailed for the data model scope as well as the same cells in the infrastructure. However, the performance measurements will not be used to predict higher cell count data models. The same methodology can be used since changes to API calls will be made in functions related to building the data model.

The building of data model for Compute can be grouped into three steps. The first is determining the compute nodes that should be included based on the data model scope. This is done with two API calls to Nova. One call is for the availability zones and the other is for the aggregates. After this step the necessary information needs to be collected for the computes nodes included in the scope. Currently, this is done with two API calls per compute node in order to get all the details. In the last step the server attribute of each compute node is used to retrieve information about all the instances. The server attribute only contains shallow instance objects so the API calls are needed to retrieve all information. To retrieve this information a for loop is used to iterate over the list of instances, subsequently, making an API call to retrieve all information for the given instance.

The implemented changed will be described for these three steps individually as well as why they were made. With the changes the same method for taking measurement as in section 7 will be used as this will allow to compare the results in fair manner. During initial measurements a performance degradation was encounter. This degradation will be explained in detail including how it was resolved. Afterwards the results and what can be concluded from them will be detailed. Finally, the results will be used to conclude a rough performance improvement estimate.

8.1. Implemented Changes

It can be assumed that any API call will introduce significant delay so, naturally, the amount of API calls to Nova should be minimized. In two out of the three previously detailed steps this could be achieved. Firstly, the amount of API calls per compute node could be reduced from two to just one. This required changes to the Nova API [57] so that the details could be retrieved immediately. In figure 25 the difference between the original and improvement implementations are shown.

```
def old_get_compute_node:
    for node_name in compute_nodes:
        cnode = nova.compute_node_by_name(
            node_name, servers=True)
        if cnode:
            node_info = nova.compute_node_by_id(
                cnode[0].id)
    ...

def new_get_compute_node:
    for node_name in compute_nodes:
        cnode = nova.compute_node_by_name(
            node_name, servers=True,
            detailed=True)
        if cnode:
    ...
```

Fig. 25: Comparison of old and methods to get compute node information.

The third step also underwent changes but did not require any changes to the Nova API. Before, an API call was necessary for every single instance. Now a single API call is used to retrieve all instances for any given compute node at once. The differences are illustrated in figure 26. All these changes should together provide a performance improvement, however, this should still be quantifiable measured. Many actors were excited for these measurements as it took effort from many different contributors to develop these changes.

```
def old_add_instance_node(self, node):
    instances = []
    for uuid in node.service['servers']:
        instance = nova.find_instance(uuid)
        instances.add(instance)
    ...

def new_add_instance_node(self, node):
    host = node.service["host"]
    filters = {'host': host}
    instances = nova.get_instance_list(
        filters)
    ...
```

Fig. 26: Comparison of old and methods to get instance information.

8.2. Performance Degradations

During the initial measurements the performance results were much worse than before. This was very unexpected as it took between 26 and 124 percent longer to build the same data models. The expected result were for the new methodology to be faster than the previous. These measurements were thoroughly discussed with the community. Many contributors expected something to be wrong to cause this regression. It took several days of investigating to determine the exact cause of this regression during this period the community collaborated extensively.

TABLE V: Degraded performance measurements

Cell ID(s)	Compute nodes	Time before	Time after	slower %
60	136	319	536	68%
61	164	751	953	26%
65	123	680	917	34%
99	44	299	671	124%
102	56	80	172	115%

Overview performance degradation after initial changes to the Nova API calls.

Eventually the cause was discovered to be due to the implementation of `get_instance_list()`. This function will perform two API calls to get all the instances from the host. This is done to ensure all instances are included as Nova can be configured to only return a limited amount of instances per request. The default limit is 1000 instances per request which a single compute node should never realistically exceed. Luckily, the `get_instance_list()` function provides a limit parameter to manually set the amount of instances that are to be retrieved. If this parameter is set no additional API calls will be made after the set number of instances is retrieved. Coincidentally, the amount of instances on any given compute node is already retrieved as an attribute from the compute node itself. As a result this value can easily be passed to the `get_instance_list()` function. Preliminary testing showed that correctly passing the limit parameter resolved all performance degradations. With the performance degradations resolved the measurements could be redone to, finally evaluate the results.

8.3. *Results*

As mentioned before the same methodology as in section 7 is used, as a result the same amount of measurements are taken with the same combinations of single and multiple cells. A direct comparison of the measurements from before and after the improvements are shown in figure 27. The same order polynomials as used previously are computed for both sets of data. Overall there is a notable performance improvement over the entire range of compute nodes. Furthermore, the performance improvements seems to become more significant with larger collections of compute nodes. The performance improvements are evaluated at three measurements points being for the lowest amount of compute nodes, the mean of all measurements and the highest amount of compute nodes. The smallest measurements with just 44 compute nodes measured a performance improvement of 27% percent. While the largest measurements with 523 compute nodes measured a performance improvement of 39% percent. Finally, from all the combined measurements the improvement was 32% on average.

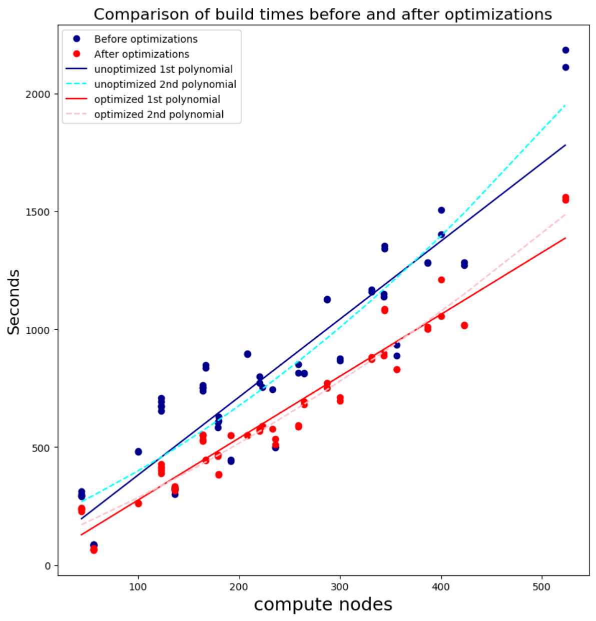


Fig. 27: Performance comparison from before and after optimizations to Nova API calls.

The results clearly emphasize the importance of using and understanding API's correctly. As it can not only affect the performance of the application itself but also impact the services it is using or the underlying network.

8.4. *Discussion*

These improvements to the performance while building the data model could perhaps be applied to other parts of Watcher as well. This would require further analysis to determine what parts could benefit, nonetheless, being resourceful is an important aspect of a mature application.

With more and more computing power becoming available worldwide to both businesses and consumers there is a subsequent increase in energy consumption worldwide [58], [59]. Program methodologies seem to have shifted from being resourceful to using more resources as development efforts seems to outweigh hardware in terms of cost. The field of green computing is an important scientific field that focuses on the effective and power efficient use of computing resources [60]. However, more effort should be spent on not only making computing hardware more efficient but more important, making applications efficient as well.

9. Parallelism Proof of Concepts (PoC)

Modern hardware typically has processing units that can execute many tasks in parallel. Typically, this is achieved using multiple so called *cores* of which all are capable of executing instructions at any given time. In more specialized hardware such as graphics cards these are typically called compute units (CU). With all these different types of hardware being able to perform many operations in parallel it has become important for applications to be able to leverage this degree of parallelism.

The analysis of Watcher indicated that only very limited patterns to enable parallelism were being used. As indicated the Application was separated into three individual executables as well as workers that should allow to run multiple audits in parallel. In practice the data model prevents running multiple audits in parallel because the data model itself is a singleton. When the singleton pattern is applied to a class only one instance of that class can exist at any given time. Since audits are very likely to have different scopes with different data models this results in effectively one audit being able to run at any given time. Any attempt to get read or write access to the data model will be blocked until the execution of the already running audit is finished. These limitations are limited to the decision engine executable although, similar limitations exist in the other executables as well.

While the implementation around the data model for audits could be improved there are likely better candidates to improve parallelism first. Even after the Nova API improvements the majority of execution time is still spent on building the data model. To improve the parallelism in Watcher the building of the data model is the current best candidate. In the rest of this section the approach to implement the parallelism improvements and the reasoning will be described. In addition the solution will be evaluated using performance measurement similar to other improvements. Important to note is that at the time of writing (30th of July 2019) these improvements are still Proof of Concepts (PoC) and none have been implemented in Watcher.

The potential of applying parallelism was discussed several times with the upstream community. These discussions led to different methodologies being proposed. As a result two PoC's have been created to test different methodologies. One of the methodology uses a threadpool [61] to which functions can be submitted for execution. The other methodologies uses a special library called Taskflow [62]. Taskflow is an extensive library supporting many different types of parallelism including distributed onces, however, the PoC implementation will focus on using a unordered flow. Some aspects of the implementations are the same in both, notably the different parts that will be parallelized. As described in other sections the operation of building the data model consists of three steps. First, the compute nodes are determined based on the aggregates and availability zones. Second, the relevant information is retrieved for each compute node and finally, the relevant information is retrieved for all the instances per given compute node. Each of these steps can be parallelized and will be in both implementations. In figure 28 the psuedo code demonstrates how these three steps are executed in parallel one after the other.

```
def add_physical_layer():
    compute_nodes = set()
    waits = []

    waits.add(parallelget(aggregates))
    waits.add(parallelget(availability_zones))
    wait_until(waits)

    for node in compute_nodes:
        waits.add(parallelget(node))
    wait_until(waits)

    for node in compute_nodes:
        waits.add(parallelget(node.servers))
    wait_until(waits)
```

Fig. 28: Pseudo code demonstrating parallel operation.

The threadpool uses a singular pattern so the same pool can be accessed from anywhere within Watcher's code. Furthermore, the amount of workers or *threads* to be supported can be configured to best suit different infrastructures. Effectively, the implementation allows to queue the execution of arbitrary functions so that they can be executed in parallel. Upon submitting a function for execution the futurist library returns a so called future. This object allows to monitor the execution process, therefor allowing to retrieve the returned values when finished. Furthermore the futurist library exposes easy methodologies for performing additional operations when a future finishes execution. If desired an additional function could be submitted to the threadpool when the execution of another finishes. This pattern is implemented in the PoC threadpool as well. The implementation offers one special function for effectively using the threadpool named *do_while_futures*. This function allows to perform a do while loop iterating over a list of futures until all of have completed execution. Natively this do while loop does not exist in Python, nevertheless, in this case it effectively allows to submit functions of step three as soon as a function from step two finishes execution. This special do while loop for futures is shown in figure 29.

```
def do_while_futures(futures, fn, *args, **kwargs):
    waits = wait_for_any(futures)
    while len(waits[0]) > 0 or len(waits[1]) > 0:
        for future in wait_for_any(futures)[0]:
            fn(future, *args, **kwargs)
            futures.remove(future)
        waits = wait_for_any(futures)
```

Fig. 29: Do while loop for list of asynchronous futures.

The taskflow implementation has many similarities with the threadpool implementation but does not use the singular pattern. Instead the implementation works by defining so called tasks which are executed by placing them into flows. Each of these tasks extends a class from the taskflow library to implement a specific interface. These tasks do not work with the concept of futures, as a result parts of step three can not immediately be scheduled when a part of step two finishes. Since the threadpool implementation does support this pattern it is likely that the taskflow implementation will perform worse. For each distinct parallel operation one task was defined which resulted in a total of four tasks. These tasks are for 1) collecting compute nodes from aggregates. 2) Collecting compute nodes from availability zones 3) collecting detailed compute node information. 4) Collecting instance information. First the tasks for aggregates and availability zones are added to one flow so they can be executed in parallel. When these finish execution the resulting set of compute nodes is used for another flow. This second flow gathers all detailed compute node information, subsequently, when all tasks of the second flow finish the resulting compute node objects are used for the last flow. This last flow gathers all instances per compute node. Finally, when the last flow finishes execution the data model is build completely. One of these tasks is shown in figure 30, notably these tasks return data by updating a reference similar to many paradigms in C++. For clarity parameters used for returning data start with an underscore such as *_instances* for instance.

```
class GetInstancesTask(task.Task):
    def execute(self, node_name, length,
                nova_ref, _instances):
        filters = {'host': node_name}
        limit = length if length <= 1000 else -1
        _instances.extend(
            nova_ref.get_instance_list(
                filters=filters,
                limit=limit))
```

Fig. 30

The managing of parameters for tasks is significantly more complex than with the threadpool implementation. Taskflow manages parameters by specifying binds when creating tasks, furthermore, these binds are named key value pairs which enables sharing parameters among different tasks. Although perhaps cumbersome for simple forms of parallelism as required in this case, it provides the powerful feature were one task is able to provide the parameters for another.

Both PoC's were not merged upstream, however, it is still important to make the source code for these implementations available. To achieve this both patches have been submitted to the review system Gerrit**??** with a so called do not merge tag. These patches can be downloaded by using git in combination with a command line interpreter such as Bash. The following two commands can be used to retrieve the patches for the taskflow and threadpool PoC's respectively as is shown in the figures 31 32. Finally, it is important to note that these PoC's are build in addition to the improvements to the Nova API calls.

```
git fetch https://review.opendev.org/openstack/\
    watcher refs/changes/64/671264/8
&& git checkout FETCH_HEAD
```

Fig. 31: Command to retrieve patch for taskflow PoC.

```
git fetch https://review.opendev.org/openstack/\
    watcher refs/changes/56/671556/3
&& git checkout FETCH_HEAD
```

Fig. 32: Command to retrieve patch for threadpool PoC.

Before comparing the results with previous implementations the two different PoC's are first evaluated and compared. Subsequently, only the best performing PoC will be compared against previous implementations. The same comparison between implementations is used to compare the threadpool and taskflow as is shown in figure 33. The results clearly indicate that the taskflow implementation is slightly slower than the threadpool implementation. Not only overall but more important, for almost every single measurement with one or two exceptions. Similarly the two PoC's are compared using a scatterplot as is shown in figure 34. Again, the scatterplot shows that the taskflow PoC has slightly worse performance, moreover, it shows that the taskflow implementation has a larger variance than the threadpool. In addition the measurements indicated significantly higher memory usage with the taskflow PoC. The memory usage was in the order of around 400 to 500 megabytes of memory compared to around 140 megabytes for the threadpool implementation. The larger memory consumption is likely due to the large set of advanced features the taskflow library supports, however, almost all of these features are unnecessary for the required form of parallelism. Typically the taskflow library is used to perform important tasks in distributed systems that need a particular order, moreover, typically tasks need to handle special revert or fallbacks methods to design behavior in the case of failure. Because of the worse performance and higher memory consumption it is clear that the threadpool implementation should be used.

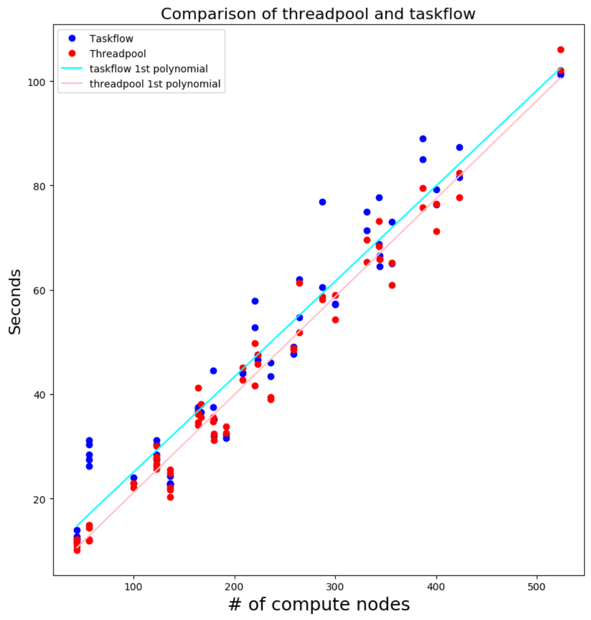


Fig. 33: Comparison of threadpool and taskflow implementation.

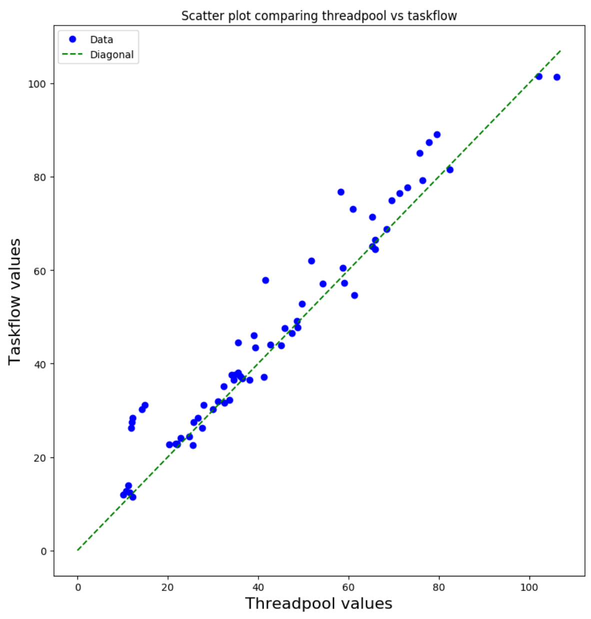


Fig. 34: Scatterplot with the results from the threadpool on the x axis and the results from the taskflow implementation on the y axis.

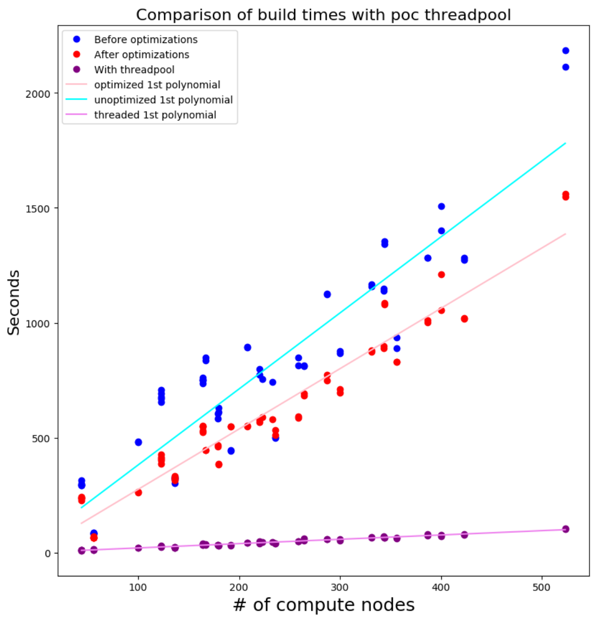


Fig. 35: Performance comparison of all three implementations to build the data model.

The same measurements as in the previous sections were taken to compare all implementations consecutively. The results of this comparison are shown in figure 35. Immediately the very large performance difference between the two previous implementations and the threadpool are clear. On average the threadpool takes 38 seconds to build the data model for both single and multiple cells. Compared to the results from the Nova API call optimizations having 527 seconds on average. This results in a build time that is roughly 14 times faster than before. Instead of denoting the improvement in percentage now the amount of times it is faster than the previous solution is denoted. This is done because when performance improvements become large it becomes hard to interpret the improvement with a percentage. For example the performance improvement of 14 times faster would be a 93% performance improvement. For completeness the minimum and maximum number of compute nodes are also compared with the same denotation. The minimum number of compute nodes has a 21 times faster data model build time while with the maximum number of compute nodes the build time is 15 times faster. Overall the parallelism using both the threadpool or taskflow implementation have very large performance benefits. Finally, because of these large improvements it is important to invest the necessary development efforts to fully implement the threadpool PoC.

9.1. Discussion

The currently developed PoC's are not written in mature way that would be sufficient for production code. Still the performance improvements of especially the threadpool PoC are so large that it is important to invest the necessary efforts to develop a mature threadpool implementation. Nevertheless, the most desired implementation must be discussed with the community following a thorough review process.

The resulting threadpool implementation can be reused in other parts of the Watcher decision engine, predominantly when collecting metrics from datasources. A very rudimentary test showed that similar speed improvements can be achieved during the collection of metrics as the building of the data model.

The amount of parallelism used during these measurements was using 16 threads, however, higher or lower values can have better results depending on the infrastructure available. The 16 thread configuration was used because it had the best results in CERN's infrastructure but the value should be user configurable to best fit different infrastructures. Moreover, high values could negatively impact other services their operation. Therefor, the importance of Watcher's performance should evaluated against the operation of the rest of the infrastructure.

10. General Improvements

This last section regarding changes to Watcher details improvements that are either smaller or otherwise less significant. Nevertheless, these changes are still important and improve the maturity of Watcher overall. These changes will be described in lesser detail and typically do not include any performance evaluations. In addition, some of these improvements are addressed in individual sub sections such as problems with Python version 3.7. The short general improvements without sub sections are detailed first.

The Grafana datasource required several additional changes before it could be fully developed. Now Watcher supports a configuration parameters which forces datasources to be used in a specific order. Effectively, this allows to now use the Grafana datasource without attempting to use Gnocchi or others. This prevents unnecessary errors from being generated. Another improvement to the datasources was the removal of several pieces of functionality from the datasource base class.

Before the datasource base class contained the metric mappings discussed before for every individual datasource. In object oriented programming when a base class knowns information about it's subclasses this is known as an anti-pattern. These anti-patterns are paradigms that can be used in software but should always be avoided, however, typically anti-patterns do list one or two exceptions. In addition to patterns and anti patterns object oriented programming also defines principles. These principles known as SOLID [63] can be used to identify how good a pattern is or if it an anti-pattern. Other principles such as GRASP [64] exist while it should be noted that they are unrelated. When taking SOLID into account it becomes clear that two principles are violated with this anti-pattern. These are the open close [65] and Liskov substitution [66] principles. These problems were resolved by making the metric map a attribute of the base class, subsequently every sub class would have to define the correct values for this metric map. The result is that the base class does not known about the existence of sub classes.

The final change that was made to the datasource was the renaming of functions to make them consistent. This helps other developers understand the operation of Watcher more easily while, additionally, it improves the maintainability. The changes were simple such as functions being named *get_instance_memory* and *get_host_ram*, naturally such functions should be renamed so they both use the same terms.

10.1. *Python 3.7 Deadlock*

Python currently still supports two main versions which are typically individually explicitly installed on systems. These version are Python 2.7 and 3.x although Python 2.7 is scheduled to no longer be maintained after the first of January 2020 [67]. This is done to provide a more maintainable and healthier programming language. Additionally, When a program or tool is scheduled to no longer be maintained it is called End Of Life (EOL). Because of the EOL for Python 2.7 it is important to ensure Watcher can fully function with Python version 3.x. However, even though most functionality of Watcher works identical on both versions the unit tests encounter significant problems.

Unit tests in most OpenStack components are run using a tool called stestr [68] which is managed by another higher level tool called Tox [69]. Both stestr and Tox support parallelism but in different ways. This parallelism is important because the unit tests in Watcher when using Python 3.x encounter deadlocks. A deadlock is a phenomenon in parallel programming when multiple threads are all waiting for the release of the same resource but none of them are able to free any resource without continuing to operate. This causes all threads to wait indefinitely and the execution of the program halts completely.

This problem was well known due to a bug report [70] but since Watcher knows many levels of parallelism it was challenging to determine were the deadlock was caused. Luckily the configuration used for Tox did not result in any form of parallelism being used. The stestr still used parallelism to speed up the execution of unit tests. Moreover, configuring stestr to not use any form of parallelism resolved the deadlocks. This is however, not the preferred solution as it indicates another threading problem that lies elsewhere in Watcher's code. Eventually the deadlocks were determined to be caused by the eventlet [71] library. This library no longer needs to be used within OpenStack projects because the community actively develops the futurist [61] library instead. The use of eventlet was removed from the decision engine being replaced with futurists green threads instead.

10.2. *OpenStack Placement Support*

One of the major advantages of virtual machines (instances) is that multiple virtual processor cores can be allocated to the same physical core. This principle is called over-commit and is not only possible with processors but also with random access memory (RAM) and disk space. However, Over-commitment of resources needs to done carefully as it can cause contention which will cause performance degradation. One of the methods available to prevent contention is the allocation-rate as this controls the allowable percentage of over-commitment. When the allocation ratio is two only two vcpus will be allocated to the same physical core. In OpenStack these concepts of over-commit and allocation ratio are also supported as a result it is heavily used in CERN's cloud infrastructure.

Before the recent release cycle of Stein these allocation ratios were managed by the Nova component. Currently, these ratios are part of a new component called Placement. Watcher never supported these allocation ratios but should since strategies could make much more informed decision if it was available. The introduction of Placement as a new OpenStack component is an excellent time to invest in the development of this feature.

The support of Placement was suggested by Licanwei [29] who also developed support for this Component. Currently, much of the initial work to support Placement is complete but it is not integrated into strategies yet. To support Placement functions to retrieve specific types of data have been placed in a helper class. This is similar to how interaction with other OpenStack components is achieved. Subsequently, this helper class is used to retrieve the necessary information during the building of the data model. An example of how these attributes are added to elements in the data model is shown in figure 36.

```
inv = placement.get_inventories(node.id)
if inv and orc.VCPU in inv:
    vcpus = inv[orc.VCPU]['total']
    vcpu_reserved = inv[orc.VCPU]['reserved']
    vcpu_ratio = inv[orc.VCPU]['allocation_ratio']
else:
    vcpus = node.vcpus
    vcpu_reserved = 0
    vcpu_ratio = 1.0
```

Fig. 36: Demonstration of added attributes with compute node inventories including fallback.

The inventory data from Placement allows to calculate how much of a given resource on the compute node is free. Each compute node element in the data model has been given a function to determine this based on current attribute values to help simplify this calculation. The formula for this calculation is shown below.

$$(vcpu - resevered) * ratio - used$$

Naturally, making retrieving all this information from Placement requires additional API calls. The current implementation introduces two additional API calls per compute node. The same performance measurements done in sections 7 8 9 should be redone with these new API calls. Finally, to further improve Placement support new strategies need to be developed to use the information from Placement effectively.

11. Conclusion

The previous sections describe a methodology to engage with a community while improving the communities maturity and methodologies for working together. Watcher's community shows to be accepting to these proposed changes resulting in bi-weekly meetings which greatly improved communication. Although the Watcher community is small compared to other OpenStack component communities they collaborate in a mature and extensive manner. This maturity will help resolve any potential future issues but for now efforts are best focused on developing Watcher instead.

Analyzing the source code and operation of Watcher helps identify many of the important development efforts in this work. The section describes a thorough methodology to perform such an analysis which could be applied to other open source projects. It should be noted that the analysis results are largely described in the introduction to improve the structure of this work.

This work includes five sections that each describe an improvement that is necessary to deploy Watcher. Developing the Grafana datasource is part of the original assignment the other four sections are determined from the analysis.

The Grafana datasource requires extensive configuration which leads to significant development efforts. This configuration flexibility is necessary to support many different infrastructures which other users might have. The Grafana datasource is improved by supporting a YAML file for configuration. Any further improvements to the Grafana datasource are simplified because of the development of Grafana translators which handle specific database conversions. Finally, How the datasource is configured is documented to improve the ease of use for other users and as future reference.

The Grafana datasource requires the refactoring of the interface between datasources and strategies. The base class uses multiple methodologies because multiple datasources were developed before the base class. The refactoring reduced the methodologies to only one so every datasource can be used for every strategy.

The data model scope is the first of three changes to improve the performance of Watcher. This change allows selection of specific aggregates and / or availability zones, in such a way that parts of the infrastructure can be excluded from the audit. This significantly reduces the amount of time required to build the data model. Statistical methods are used to estimate the achieved performance improvement for which the significance is evaluated using linear regression. The result showed that the build time is estimated to be reduced from between 26471 and 194304 seconds to just 427 seconds for a single aggregate. Conservatively the time to build the data model is now 62 times shorter.

The second changes are made to the API calls used to build the data model. These API calls retrieve important information about compute nodes and instances from the Nova component. Typically, API calls introduce significant delays in the operation of applications such as the decision engine of Watcher. Theoretically reducing the amount of API calls will reduce the time to build the data model. The amount of API calls is reduced from two to one for compute nodes while it is reduced from n to 1 for the instances for a given compute node. Due to an issue in the *get_instance_list* function the performance is much worse, however, this issue can be solved by correctly setting the limit parameter for this function. The performance is evaluated with the limit parameter correctly set resulting in a performance improvement. The reduced API calls introduce a performance improvement of 32% on average.

The third changes are made by implementing parallelism when building the data model. Two different methodologies are evaluated each using the same three step process. The primary differences between the methodologies is the underlying Python library used to introduce parallelism, however, both of these libraries are developed by the OpenStack community. Both implementations called threadpool and taskflow respectively are compared against each other. The threadpool implementation has the best performances used to compare against the API call changes. The threadpool implementation offers a large performance improvement that reduces the data model build time from 527 to 38 seconds on average. However, before the threadpool can be implemented in Watcher it needs to be reviewed with subsequent changes as the current implementation is a proof of concept.

Several smaller improvements are made that are to small to be described in an entire section. These improvements include 1) improved Python 3.7 support 2) improvements to the datasource base class 3) support for OpenStack Placement among others. Other changes also alter functionality of the datasources. The support for Python 3.7 is important because older versions of Python are losing support in the near future. While OpenStack Placement allows strategies to make more insightful decision. This is because Placement provides information about hardware allocation ratios configured by end users. Finally, the changes to the datasources are required to integrate the Grafana datasource or simplify the integration.

This work introduces many changes to Watcher to either increase the maturity or performance. The changes made to improve the build time of the data model manage to reduce it from an estimated 26471 to 38 seconds on average for single cells. Many other changes allow Grafana to be used as a datasource with the flexibility to be used in many different configurations and infrastructures. However, many other further improvements remain possible such as the refactoring of the data model scope architecture for example.

12. Discussion

Although many changes are made to Watcher there are still other areas which can be evaluated and possibly improved. Future work could evaluate these areas such as improving the time to perform an audit for example. Current evaluations only measured the time to build the data model, consequently, not taking other factors into account. Factors such as the gathering of metrics or even differences between strategies could have an impact on the time required to execute an audit.

This work introduced many changes to either improve performance or maturity. However, the purpose of Watcher is to improve OpenStack cloud infrastructure to reach a certain goal. In future work the effective of strategies to reach these goals should be thoroughly evaluated. Likely many strategies can be improved in different ways such as the algorithm used for example.

Abbreviations

1) AUAS - Amsterdam University of Applied Sciences
2) TI - Technical Informatics
3) ALICE - A Large Ion Collider Experiment
4) RPS - Resource Provisioning Services
5) API - Application Programming Interface
6) AQMP - Advanced Message Queuing Protocol
7) UUID - Universally unique identifier
8) R&D - Research and Development
9) IaaS - Infrastructure as a Service
10) CERN - European Organization for Nuclear Research
11) LHC - Large Hadron Collider
12) PTL - Project Team Lead(er)
13) REST - REpresentational State Transfer
14) RFC - Request For Comments
15) HTTP - HyperText Transfer Protocol
16) SQL - Structured Query Language
17) YAML - YAML Ain't Markup Language
18) EOL - End Of Life
19) OOM - Out Of Memory
20) JSON - JavaScript Object Notation
21) PoC - Proof of Concept
22) CU - Compute Unit
23) RAM - Random Access Memory

References

[1] European Organization for Nuclear Research, "Cern," accessed: 2019-04-23. [Online]. Available: https://home.cern/
[2] OpenStack Foundation, "Openstack watcher," accessed: 2019-02-21. [Online]. Available: https://wiki.openstack.org/wiki/Watcher
[3] European Organization for Nuclear Research, "Cern data center," accessed: 2019-04-24. [Online]. Available: https://home.cern/science/computing/data-centre
[4] OpenStack Foundation, "Openstack service map," accessed: 2019-04-16. [Online]. Available: https://www.openstack.org/software/
[5] IBM, "An architectural blueprint for autonomic computing," Tech. Rep., June 2005. [Online]. Available: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf
[6] OpenStack Foundation, "Openstack monasca," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Monasca
[7] ——, "Openstack gnocchi," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Gnocchi
[8] ——, "Openstack ceilometer," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Ceilometer
[9] ——, "Openstack nova," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Nova
[10] ——, "Openstack neutron," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Neutron
[11] ——, "Openstack ironic," accessed: 2019-02-22. [Online]. Available: https://wiki.openstack.org/wiki/Ironic
[12] OASIS, "AQMP," accessed: 2019-06-22. [Online]. Available: http://www.amqp.org/
[13] The Linux Foundation, "The linux kernel," accessed: 2019-05-17. [Online]. Available: https://www.kernel.org/
[14] OpenStack Foundation, "Gerrit," accessed: 2019-03-04. [Online]. Available: https://review.openstack.org/
[15] Google, "Rietveld," accessed: 2019-05-23. [Online]. Available: https://github.com/rietveld-codereview/rietveld
[16] Canonical, "Watcher launchpad," accessed: 2019-08-12. [Online]. Available: launchpad.net/Watcher
[17] ——, "Canonical," accessed: 2019-05-17. [Online]. Available: https://www.canonical.com/
[18] OpenStack foundation, "Watcher meeting agenda," accessed: 2019-08-12. [Online]. Available: wiki.openstack.org/wiki/Watcher_Meeting_Agenda
[19] OpenStack eavesdrop, "Watcher meeting logs 7 november," accessed: 2019-05-23. [Online]. Available: eavesdrop.openstack.org/meetings/watcher/2018/watcher.2018-11-07-08.00.log.txt
[20] ——, "Watcher meeting logs 7 november," accessed: 2019-05-23. [Online]. Available: http://eavesdrop.openstack.org/meetings/watcher/2018/watcher.2018-10-10-08.00.log.txt
[21] Launchpad, "Watcher drivers team," accessed: 2019-05-23. [Online]. Available: https://launchpad.net/~watcher-drivers
[22] OpenStack Foundation, "Watcher wiki meetings," accessed: 2019-05-28. [Online]. Available: https://wiki.openstack.org/wiki/Watcher#Meetings
[23] ——, "Watcher documentation meetings," accessed: 2019-05-28. [Online]. Available: https://docs.openstack.org/watcher/latest/contributor/contributing.html
[24] ——, "Eavesdrop meetings overview," accessed: 2019-05-28. [Online]. Available: http://eavesdrop.openstack.org/#Watcher_Team_Meeting
[25] ——, "Watcher first meeting agenda," accessed: 2019-06-10. [Online]. Available: https://wiki.openstack.org/wiki/Watcher_Meeting_Agenda#04.2F10.2F2019
[26] ——, "Watcher first meeting log," accessed: 2019-06-10. [Online]. Available: http://eavesdrop.openstack.org/meetings/watcher/2019/watcher.2019-04-10-08.01.log.html
[27] ——, "Core reviewer guidelines," accessed: 2019-06-10. [Online]. Available: https://docs.openstack.org/infra/manual/core.html

[28] Canonical, “Chenker,” accessed: 2019-06-13. [Online]. Available: https://launchpad.net/~chenker
[29] ——, “Lican wei,” accessed: 2019-06-13. [Online]. Available: https://launchpad.net/~li-canwei2
[30] Debian, “Deiban binary package,” accessed: 2019-06-13. [Online]. Available: https://manpages.debian.org/unstable/dpkg-dev/deb.5.en.html
[31] OpenStack Foundation, “Openstack devstack,” accessed: 2019-03-04. [Online]. Available: https://docs.openstack.org/devstack/latest/
[32] ——, “Devstack plugins,” accessed: 2019-06-14. [Online]. Available: https://docs.openstack.org/devstack/latest/plugins.html
[33] ——, “Openstack horizon,” accessed: 2019-06-14. [Online]. Available: https://wiki.openstack.org/wiki/Horizon
[34] F. octo Forster, “Collectd,” accessed: 2019-07-09. [Online]. Available: https://collectd.org/
[35] InfluxData, “Influxdb,” accessed: 2019-07-09. [Online]. Available: https://www.influxdata.com/products/influxdb-overview/
[36] Elasticsearch B.V., “Elasticsearch,” accessed: 2019-07-09. [Online]. Available: https://www.elastic.co/products/elasticsearch
[37] T. Berners-Lee, L. Masinter, and M. McCahill, “Uniform resource locators (url),” Internet Requests for Comments, IETF, RFC 1738, December 1994. [Online]. Available: https://www.ietf.org/rfc/rfc1738.txt
[38] D. Hardt and M. Jones, “The oauth 2.0 authorization framework: Bearer token usage,” Internet Requests for Comments, IETF, RFC 6750, October 2012. [Online]. Available: https://tools.ietf.org/html/rfc6750
[39] D. Hardt and Ed, “The oauth 2.0 authorization framework,” Internet Requests for Comments, IETF, RFC 6749, October 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749
[40] YAML, “YAML Ain’t Markup Lanague,” accessed: 2019-07-17. [Online]. Available: https://yaml.org/spec/1.0/
[41] Dantali0n, “Grafana upstream patch,” accessed: 2019-07-19. [Online]. Available: https://review.opendev.org/#/c/649341/
[42] InfluxData, “Influxdb retention periods,” accessed: 2019-07-22. [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/guides/downsampling_and_retention/
[43] Oracle, “Prepared sql statement syntax,” accessed: 2019-07-22. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-prepared-statements.html
[44] OpenStack Foundation, “Grafana datasource documentation,” accessed: 2019-07-22. [Online]. Available: https://docs.openstack.org/watcher/latest/datasources/grafana.html
[45] Postman Inc, “Postman,” accessed: 2019-07-22. [Online]. Available: https://www.getpostman.com/
[46] Canonical, “Grafana incorrectly logs errors when yaml is used for metrics,” accessed: 2019-07-22. [Online]. Available: https://bugs.launchpad.net/watcher/+bug/1837400
[47] OpenStack Foundation, “Formal datasource interface,” accessed: 2019-07-23. [Online]. Available: https://specs.openstack.org/openstack/watcher-specs/specs/train/approved/formal-datasource-interface.html
[48] Pocoo, “Sphinx python documentation generator,” accessed: 2019-07-23. [Online]. Available: http://sphinx-doc.org/
[49] D. Crockford, “The application/json media type for javascript object notation (json),” Internet Requests for Comments, IETF, RFC 4627, July 2006. [Online]. Available: Theapplication/jsonMediaTypeforJavaScriptObjectNotation(JSON)
[50] OpenStack Foundation, “Spec for data model scope,” accessed: 2019-06-27. [Online]. Available: https://specs.openstack.org/openstack/watcher-specs/specs/stein/implemented/scope-for-watcher-datamodel.html
[51] Gerrit, “Patch for data model scope,” accessed: 2019-06-27. [Online]. Available: https://review.opendev.org/#/c/640585/
[52] OpenStack Foundation, “Watcher opendev repository,” accessed: 2019-07-25. [Online]. Available: https://opendev.org/openstack/watcher
[53] ——, “Getting started - git basic,” accessed: 2019-07-25. [Online]. Available: https://git-scm.com/book/en/v1/Getting-Started-Git-Basics
[54] Scipy, “Python scipy polynomial fit,” accessed: 2019-07-02. [Online]. Available: https://docs.scipy.org/doc/numpy/reference/generated/numpy\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{p\global\mathchardef\accent@spacefactor\spacefactor}\accent10p\egroup\spacefactor\accent@spacefactorolyfit.html
[55] Gerrit, “Audit scope fixes,” accessed: 2019-06-27. [Online]. Available: https://review.opendev.org/#/c/653977/
[56] Canonical, “Matt riedemann,” accessed: 2019-07-01. [Online]. Available: https://launchpad.net/~mriedem
[57] OpenStack Foundation, “Allow searching for hypervisors and getting back details,” accessed: 2019-07-23. [Online]. Available: https://review.opendev.org/#/c/659886/
[58] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner, “United states data center energy usage report,” Tech. Rep., 06/2016 2016.
[59] EirGrid, “All-island generation capacity statement 2017-2026,” Tech. Rep., 04 2017.
[60] S. Murugesan, “Harnessing green it: Principles and practices,” *IT Professional*, vol. 10, no. 1, pp. 24–33, Jan 2008.
[61] OpenStack Foundation, “Welcome to futurist’s documentation!” accessed: 2019-07-31. [Online]. Available: https://docs.openstack.org/futurist/latest/index.html
[62] ——, “Taskflow,” accessed: 2019-07-31. [Online]. Available: https://docs.openstack.org/taskflow/latest/
[63] R. C. Martin, “Design principles and design patterns,” *Object Mentor*, vol. 1, no. 34, p. 597, 2000.
[64] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, oct 2004. [Online]. Available: https://www.xarg.org/ref/a/0131489062/
[65] R. C. Martin, *More C++ Gems (SIGS Reference Library)*. Cambridge University Press, jan 2000. [Online]. Available: https://www.xarg.org/ref/a/0521786185/
[66] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994.
[67] Python, “Python 2.7 EOL patch,” accessed: 2019-06-24. [Online]. Available: https://github.com/python/devguide/pull/344
[68] Matthew Treinish, “stestr,” accessed: 2019-07-09. [Online]. Available: https://github.com/mtreinish/stestr
[69] tox-devolopment team, “tox,” accessed: 2019-07-09. [Online]. Available: https://github.com/tox-dev/tox
[70] Canonical, “Python 3.7 deadlocks,” accessed: 2019-06-25. [Online]. Available: https://bugs.launchpad.net/watcher/+bug/1786326
[71] Eventlet, “Eventlet,” accessed: 2019-08-07. [Online]. Available: https://eventlet.net/

Mutli cell performance measurements

TABLE VII: Multi cell performance measurements before improvements
git checkout 36c2095254a24cbcfa7bc11b8bd453de0f659c5a

Cell ID(s)	**Number of compute nodes**	**Time in seconds**
60, 99	180	620
60, 99	180	612
60, 61	300	877
60, 61	300	868
60, 65	259	816
60, 65	259	850
60, 102	192	442
60, 102	192	446
61, 65	287	1128
61, 65	287	1125
61, 99	208	894
61, 99	208	896
61, 102	220	799
61, 102	220	773
65, 102	179	606
65, 102	179	583
99, 102	100	481
99, 102	100	483
99, 65	167	848
99, 65	167	836
60, 61, 99	344	1355
60, 61, 99	344	1342
61, 99, 102	264	815
61, 99, 102	264	812
60, 61, 65	423	1284
60, 61, 65	423	1272
61, 65, 102	343	1149
61, 65, 102	343	1138
61, 65, 99	331	1167
61, 65, 99	331	1159
60, 61, 102	356	935
60, 61, 102	356	888
65, 99, 102	223	755
65, 99, 102	223	744
99, 102, 60	236	502
99, 102, 60	236	498
61, 99, 102 ,65	387	1282
61, 99, 102 ,65	287	1283
60, 61, 99 ,102	400	1507
60, 61, 99 ,102	400	1402
60, 61, 99, 102, 65	523	2184
60, 61, 99, 102, 65	523	2112

Overview of measurements for combinations of cells before improvements.

TABLE VIII: Multi cell performance measurements after improvements with instance limit
git checkout 1e8b17ac46a9052234c9f56da42a2a4bb8250216

Cell ID(s)	**Number of compute nodes**	**Time in seconds**
60, 99	180	387.446
60, 99	180	384.842
60, 61	300	697.927
60, 61	300	712.997
60, 65	259	588.875
60, 65	259	592.453
60, 102	192	550.648
60, 102	192	550.281
61, 65	287	773.134
61, 65	287	750.082
61, 99	208	551.104
61, 99	208	550.228
61, 102	220	568.731
61, 102	220	576.249
65, 102	179	463.524
65, 102	179	468.651
99, 102	100	264.21
99, 102	100	261.478
99, 65	167	445.601
99, 65	167	447.505
60, 61, 99	344	1087.68
60, 61, 99	344	1081.828
61, 99, 102	264	693.764
61, 99, 102	264	682.532
60, 61, 65	423	1017.855
60, 61, 65	423	1020.404
61, 65, 102	343	898.471
61, 65, 102	343	888.279
61, 65, 99	331	881.073
61, 65, 99	331	873.355
60, 61, 102	356	831.178
60, 61, 102	356	830.58
65, 99, 102	223	589.878
65, 99, 102	223	579.599
99, 102, 60	236	512.932
99, 102, 60	236	535.225
61, 99, 102 ,65	387	1001.074
61, 99, 102 ,65	287	1010.962
60, 61, 99 ,102	400	1054.425
60, 61, 99 ,102	400	1212.645
60, 61, 99, 102, 65	523	1549.76
60, 61, 99, 102, 65	523	1560.273

Overview of measurements for combinations of cells after improvements with instance limit.

Single cell performance measurements

TABLE IX: Performance measurements single cell before improvements
git checkout 36c2095254a24cbcfa7bc11b8bd453de0f659c5a

Cell ID	Number of compute nodes	Number of instances	Time in seconds
99	44	351-353	314
99	44	351-353	301
99	44	351-353	294
99	44	351-353	295
99	44	351-353	291
60	136	546-670	330
60	136	546-670	323
60	136	546-670	319
60	136	546-670	323
60	136	546-670	302
61	164	1785	763
61	164	1785	738
61	164	1785	753
61	164	1785	750
61	164	1785	752
102	56	11-157	87
102	56	11-157	85
102	56	11-157	85
102	56	11-157	87
102	56	11-157	80
65	123	1640	707
65	123	1640	676
65	123	1640	671
65	123	1640	693
65	123	1640	655

Overview of measurements for single cells before improvements

TABLE X: Performance measurements single cell after improvements without instance limit
git checkout e4f80b54613d0858de47cee925d73927089acde7

Cell ID	Number of compute nodes	Number of instances	Time in seconds
99	44	351-353	742
99	44	351-353	661
99	44	351-353	638
99	44	351-353	669
99	44	351-353	645
60	136	546-670	512
60	136	546-670	546
60	136	546-670	525
60	136	546-670	552
60	136	546-670	545
61	164	1785	988
61	164	1785	967
61	164	1785	948
61	164	1785	943
61	164	1785	918
102	56	11-157	146
102	56	11-157	191
102	56	11-157	164
102	56	11-157	164
102	56	11-157	195
65	123	1640	900
65	123	1640	902
65	123	1640	915
65	123	1640	923
65	123	1640	917

Overview of measurements for single cells after improvements without instance limit

TABLE XI: Performance measurements single cell after improvements with instance limit
git checkout 1e8b17ac46a9052234c9f56da42a2a4bb8250216

Cell ID	Number of compute nodes	Number of instances	Time in seconds
99	44	351-353	239
99	44	351-353	242
99	44	351-353	236
99	44	351-353	238
99	44	351-353	227
60	136	546-670	322
60	136	546-670	329
60	136	546-670	325
60	136	546-670	316
60	136	546-670	333
61	164	1785	550
61	164	1785	525
61	164	1785	553
61	164	1785	531
61	164	1785	550
102	56	11-157	65
102	56	11-157	66
102	56	11-157	64
102	56	11-157	73
102	56	11-157	67
65	123	1640	388
65	123	1640	415
65	123	1640	428
65	123	1640	410
65	123	1640	408

Overview of measurements for single cells after improvements with instance limit

TABLE XII: Performance measurements single cell threadpool
git fetch https://review.opendev.org/openstack/watcher refs/changes/56/671556/3 && git checkout FETCH_HEAD

Cell ID	Number of compute nodes	Number of instances	Time in seconds
99	44	351-353	11.532
99	44	351-353	10.124
99	44	351-353	10.747
99	44	351-353	12.197
99	44	351-353	11.309
60	136	546-670	20.33
60	136	546-670	21.669
60	136	546-670	25.537
60	136	546-670	22.108
60	136	546-670	24.791
61	164	1785	36.517
61	164	1785	34.597
61	164	1785	36.072
61	164	1785	34.123
61	164	1785	41.216
102	56	11-157	14.896
102	56	11-157	14.325
102	56	11-157	12.065
102	56	11-157	11.881
102	56	11-157	12.166
65	123	1640	27.955
65	123	1640	30.030
65	123	1640	25.695
65	123	1640	27.686
65	123	1640	26.640

Overview of measurements for single cells with poc threadpool

TABLE XIII: Performance measurements single cell taskflow
git fetch https://review.opendev.org/openstack/watcher refs/changes/64/671264/8 && git checkout FETCH_HEAD

Cell ID	Number of compute nodes	Number of instances	Time in seconds
99	44	351-353	12.343
99	44	351-353	11.972
99	44	351-353	12.766
99	44	351-353	11.555
99	44	351-353	13.913
60	136	546-670	22.702
60	136	546-670	22.912
60	136	546-670	22.605
60	136	546-670	22.768
60	136	546-670	24.344
61	164	1785	36.891
61	164	1785	36.503
61	164	1785	37.308
61	164	1785	37.527
61	164	1785	37.117
102	56	11-157	12.243
102	56	11-157	14.501
102	56	11-157	13.308
102	56	11-157	12.271
102	56	11-157	14.823
65	123	1640	31.169
65	123	1640	30.268
65	123	1640	27.459
65	123	1640	26.185
65	123	1640	28.377

Overview of measurements for single cells with poc taskflow

Plotting program for data model scope

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import scipy
from scipy.stats import chisquare
from scipy.stats import linregress
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44,  44,  44,  44,  44,  56, 56, 56, 56, 56, 100, 100, 123, 123, 123,
123, 123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179,
179, 180, 180, 192, 192, 208, 208, 220, 220, 223, 233, 236, 236, 259, 259,
264, 264, 287,  287, 300, 300, 331,  331,  343,  343,  344,  344,  356,
356, 387,  387,  400,  400,  423,  423,  523,  523]

y = [314, 301, 294, 295, 292, 87, 85, 85, 87, 80, 481, 483, 707, 676, 671,
694, 655, 330, 323, 319, 323, 302, 763, 738, 753, 750, 752, 848, 836, 606,
583, 629, 612, 442, 446, 894, 896, 799, 773, 755, 744, 502, 498, 816, 850,
815, 812, 1128, 1125, 877, 868, 1167, 1159, 1149, 1138, 1355, 1342, 935,
888, 1282, 1283, 1507, 1402, 1284, 1272, 2184, 2112]

p1 = np.poly1d(np.polyfit(x, y, 1)) # first order polynomial
p2 = np.poly1d(np.polyfit(x, y, 2)) # second order polynomia

x = np.array(x, dtype=float); y = np.array(y, dtype=float)

def func_linear(x, a, b, c, d):
    return 4.160*x
popt, pcov = curve_fit(func_linear, x, y, bounds=(0, 1))
fit_linear = func_linear(x, *popt)

#xx = np.linspace(0, 523, 1000) # measured number of compute nodes
xx = np.linspace(0, 7991, 1000) # total number of compute nodes
plt.title("Data model scope estimates")
plt.plot(x, y, 'ro', label="Measured values", color='darkblue')
plt.xlabel('# of compute nodes', fontsize=18)
plt.ylabel('Seconds', fontsize=16)
# plt.plot(xx, func_linear(xx, *popt), label="Linear", color='red')
plt.plot(xx, p1(xx), '--k', label="First order polynomial", color='darkblue')
plt.plot(xx, p2(xx), label="Second order polynomial", color='cyan')
plt.legend(loc='upper left')
plt.show()

print(p2(7991))
print(p1(7991))
print(linregress(y, p1(x)))
print(linregress(y, p2(x)))
print(linregress(y, func_linear(x, *popt)))
```

Fig. 37: The program used to plot the different graphs in the estimation of time to build data models.

Plotting program for Nova API optimizations

```python
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44,  44,  44,  44,  44,  56, 56, 56, 56, 56, 100, 100, 123, 123, 123,
123, 123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179,
179, 180, 180, 192, 192, 208, 208, 220, 220, 223, 233, 236, 236, 259, 259,
264, 264, 287,  287, 300, 300, 331,  331,  343,  343,  344,  344,  356,
356, 387,  387,  400,  400,  423,  423,  523,  523]

y = [314, 301, 294, 295, 292, 87, 85, 85, 87, 80, 481, 483, 707, 676, 671,
694, 655, 330, 323, 319, 323, 302, 763, 738, 753, 750, 752, 848, 836, 606,
583, 629, 612, 442, 446, 894, 896, 799, 773, 755, 744, 502, 498, 816, 850,
815, 812, 1128, 1125, 877, 868, 1167, 1159, 1149, 1138, 1355, 1342, 935,
888, 1282, 1283, 1507, 1402, 1284, 1272, 2184, 2112]

z = [239, 242, 236, 238, 227, 65, 66, 64, 73, 67, 264, 261, 388, 415, 428,
410, 401, 322, 329, 325, 316, 333, 550, 525, 550, 553, 531, 445, 447, 463,
468, 387, 384, 550, 550, 551, 550, 568, 576, 589, 579, 512, 535, 588, 592,
693, 682, 773,  750, 697, 712, 881,  873,  898,  888,  1087, 1081, 831,
830, 1001, 1010, 1054, 1212, 1017, 1020, 1549, 1560]

p1 = np.poly1d(np.polyfit(x, z, 1)) # first order polynomial
p2 = np.poly1d(np.polyfit(x, y, 1)) # second order polynomial
p3 = np.poly1d(np.polyfit(x, z, 2)) # first order polynomial
p4 = np.poly1d(np.polyfit(x, y, 2)) # second order polynomial

plt.plot(x, y, 'ro',label="Before optimizations", color='darkblue')
plt.plot(x, z, 'ro',label="After optimizations", color='red')

x = np.array(x, dtype=float); z = np.array(z, dtype=float)
def func_linear427(x, a, b, c, d):
    return 4.16*x
popt, pcov = curve_fit(func_linear427, x, z)
def func_linear312(x, a, b, c, d):
   return 3.12*x
popt3, pcov3 = curve_fit(func_linear312, x, z)

plt.title("Comparison of build times before and after optimizations",fontsize=16)
plt.xlabel('compute nodes', fontsize=18)
plt.ylabel('Seconds', fontsize=16)
plt.plot(x, p2(x), label="unoptimized 1st polynomial", color='darkblue') # plot second order polynomial
plt.plot(x, p4(x), '--k', label="unoptimized 2nd polynomial", color='cyan') # plot second order polynomial
plt.plot(x, p1(x), label="optimized 1st polynomial", color='red') # plot first order polynomial
plt.plot(x, p3(x), '--k', label="optimized 2nd polynomial", color='pink') # plot first order polynomial
plt.legend(loc='upper left')
plt.show()
```

Fig. 38: The program used to plot the different graphs in the measurements of the Nova API improvements.

Plotting program for threadpool proof of concept

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44,  44,  44,  44,  44,  56, 56, 56, 56, 56, 100, 100, 123, 123, 123,
123, 123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179,
179, 180, 180, 192, 192, 208, 208, 220, 220, 223, 233, 236, 236, 259, 259,
264, 264, 287,  287, 300, 300, 331,  331,  343,  343,  344,  344,  356,
356, 387,  387,  400,  400,  423,  423,  523,  523]
y = [314, 301, 294, 295, 292, 87, 85, 85, 87, 80, 481, 483, 707, 676, 671,
694, 655, 330, 323, 319, 323, 302, 763, 738, 753, 750, 752, 848, 836, 606,
583, 629, 612, 442, 446, 894, 896, 799, 773, 755, 744, 502, 498, 816, 850,
815, 812, 1128, 1125, 877, 868, 1167, 1159, 1149, 1138, 1355, 1342, 935,
888, 1282, 1283, 1507, 1402, 1284, 1272, 2184, 2112]
z = [239, 242, 236, 238, 227, 65, 66, 64, 73, 67, 264, 261, 388, 415, 428,
410, 401, 322, 329, 325, 316, 333, 550, 525, 550, 553, 531, 445, 447, 463,
468, 387, 384, 550, 550, 551, 550, 568, 576, 589, 579, 512, 535, 588, 592,
693, 682, 773,  750, 697, 712, 881,  873,  898,  888,  1087, 1081, 831,
830, 1001, 1010, 1054, 1212, 1017, 1020, 1549, 1560]
a = [11.532, 10.124, 10.747, 12.197, 11.309, 14.896, 14.325, 12.065, 11.881,
12.166, 22.848, 22.032, 27.955, 30.03, 25.695, 27.686, 26.64,  20.33, 21.669,
25.537, 22.108, 24.791, 36.517, 34.597, 36.072, 34.123, 41.216, 38.059, 35.528,
34.703, 35.511, 32.336, 31.133, 33.725, 32.479, 45.053, 42.677, 49.695, 41.647,
45.823, 47.501, 39.368, 39.028, 48.572, 48.818, 61.342, 51.799, 58.205, 58.691,
59.016, 54.243, 65.301, 69.586, 68.422, 73.126, 65.89, 65.878, 65.257, 60.926,
79.514, 75.817, 76.444, 71.243, 77.773, 82.464, 102.171, 106.073]

p1 = np.poly1d(np.polyfit(x, z, 1))
p2 = np.poly1d(np.polyfit(x, y, 1))
p3 = np.poly1d(np.polyfit(x, a, 1))
# x = np.arange(1, 9000, 1)
x = np.array(x, dtype=float)
z = np.array(z, dtype=float); a = np.array(a, dtype=float)

plt.title("Comparison of build times with poc threadpool",fontsize=16)
plt.xlabel('# of compute nodes', fontsize=18)
plt.ylabel('Seconds', fontsize=16)
plt.plot(x, y, 'ro',label="Before optimizations", color='blue')
plt.plot(x, z, 'ro',label="After optimizations", color='red')
plt.plot(x, a, 'ro',label="With threadpool",color='purple')
plt.plot(x, p1(x), label="optimized 1st polynomial", color='pink')
plt.plot(x, p2(x), label="unoptimized 1st polynomial", color='cyan')
plt.plot(x, p3(x), label="threaded 1st polynomial", color='violet')
plt.legend(loc='upper left')
plt.show()
```

Fig. 39: The program used to plot the different graphs when measuring the threadpool performance.

Plotting program for comparing taskflow and threadpool

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44, 44, 44, 44, 44, 56, 56, 56, 56, 56, 100, 100, 123, 123, 123, 123,
123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179, 179,
180, 180, 192, 192, 208, 208, 220, 220, 223, 223, 236, 236, 259, 259, 264,
264, 287, 287, 300, 300, 331, 331, 343, 343, 344, 344, 356, 356, 387, 387,
400, 400, 423, 423, 523, 523]
y = [11.532, 10.124, 10.747, 12.197, 11.309, 14.896, 14.325, 12.065, 11.881,
12.166, 22.848, 22.032, 27.955, 30.03, 25.695, 27.686, 26.64,  20.33, 21.669,
25.537, 22.108, 24.791, 36.517, 34.597, 36.072, 34.123, 41.216, 38.059, 35.528,
34.703, 35.511, 32.336, 31.133, 33.725, 32.479, 45.053, 42.677, 49.695, 41.647,
45.823, 47.501, 39.368, 39.028, 48.572, 48.818, 61.342, 51.799, 58.205, 58.691,
59.016, 54.243, 65.301, 69.586, 68.422, 73.126, 65.89, 65.878, 65.257, 60.926,
79.514, 75.817, 76.444, 71.243, 77.773, 82.464, 102.171, 106.073]
z = [12.343, 11.972, 12.766, 11.555, 13.913, 31.169, 30.268, 27.459, 26.185,
28.377, 24.045, 22.855, 31.169, 30.268, 27.459, 26.185, 28.377, 22.702, 22.912,
22.605, 22.768, 24.344, 36.891, 36.503, 37.308, 37.527, 37.117, 36.596, 38.103,
37.558, 44.568, 35.189, 31.980, 32.281, 31.634, 43.949, 44.051, 52.840, 57.936,
47.512, 46.563, 43.428, 46.102, 49.053, 47.725, 54.716, 62.030, 76.841, 60.557,
57.277, 57.158, 71.441, 74.983, 68.794, 77.741, 66.508, 64.498, 65.112, 73.105,
89.003, 85.054, 79.198, 76.403, 87.435, 81.604, 101.536, 101.318]

p1 = np.poly1d(np.polyfit(x, y, 1)) # first order polynomial
p2 = np.poly1d(np.polyfit(x, z, 1)) # second order polynomial

x = np.array(x, dtype=float); z = np.array(z, dtype=float)
def func_linearthread(x, a, b, c, d):
    return 0.41*x
popt, pcov = curve_fit(func_linearthread, x, y)
def func_lineartaskflow(x, a, b, c, d):
    return 0.27*x
popt2, pcov2 = curve_fit(func_lineartaskflow, x, z)

plt.title("Comparison of threadpool and taskflow",fontsize=16)
plt.xlabel('# of compute nodes', fontsize=18)
plt.ylabel('Seconds', fontsize=16)
plt.plot(x, z, 'ro',label="Taskflow", color='blue')
plt.plot(x, y, 'ro',label="Threadpool", color='red')
#plt.plot(x, func_linearthread(x, *popt), label="Linear 0.41x", color='blue') # plot linear for mean 316
#plt.plot(x, func_lineartaskflow(x, *popt2), label="Linear 0.27x", color='red') # plot linear for mean 316
plt.plot(x, p2(x), label="taskflow 1st polynomial", color='cyan') # plot first order polynomial
plt.plot(x, p1(x), label="threadpool 1st polynomial", color='pink') # plot first order polynomial
plt.legend(loc='upper left')
plt.show()
```

Fig. 40: The program used compare taskflow and threadpool PoC's

Scatter plot program for measured vs estimated data model scope values

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import scipy
from scipy.stats import chisquare
from scipy.stats import linregress
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44,  44,  44,  44,  44,  56, 56, 56, 56, 56, 100, 100, 123, 123, 123,
123, 123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179,
179, 180, 180, 192, 192, 208, 208, 220, 220, 223, 233, 236, 236, 259, 259,
264, 264, 287,  287, 300, 300, 331,  331,  343,  343,  344,  344,  356,
356, 387,  387,  400,  400,  423,  423,  523,  523]
y = [314, 301, 294, 295, 292, 87, 85, 85, 87, 80, 481, 483, 707, 676, 671,
694, 655, 330, 323, 319, 323, 302, 763, 738, 753, 750, 752, 848, 836, 606,
583, 629, 612, 442, 446, 894, 896, 799, 773, 755, 744, 502, 498, 816, 850,
815, 812, 1128, 1125, 877, 868, 1167, 1159, 1149, 1138, 1355, 1342, 935,
888, 1282, 1283, 1507, 1402, 1284, 1272, 2184, 2112]

p1 = np.poly1d(np.polyfit(x, y, 1)) # first order polynomial
p2 = np.poly1d(np.polyfit(x, y, 2)) # second order polynomia

x = np.array(x, dtype=float); y = np.array(y, dtype=float)

def func_linear(x, a, b, c, d):
    return x
popt, pcov = curve_fit(func_linear, x, y, bounds=(0, 1))

def func_linear_est(x, a, b, c, d):
    return 4.16*x
popt2, pcov2 = curve_fit(func_linear_est, x, y, bounds=(0, 1))

xx = np.linspace(0, 2112, 1000)
plt.title("Scatter plot comparing measured vs estimated values")
plt.plot(y, p1(x), 'ro', label="first order polynomial", color='blue')
plt.plot(y, p2(x), 'ro', label="second order polynomial", color='red')
#plt.plot(y, func_linear(x, *popt2), 'ro', label="4.16*x", color='darkblue')
plt.xlabel('Measured values', fontsize=18)
plt.ylabel('Estimated values', fontsize=16)
plt.plot(xx, func_linear(xx, *popt), '--k', label="Diagonal", color='green')
plt.legend(loc='upper left')
plt.show()

print(linregress(y, p1(x)))
print(linregress(y, p2(x)))
```

Fig. 41: Scatter plot program for measured vs estimated data model scope values.

Scatter plot program for comparing threadpool and taskflow PoC's

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import scipy
from scipy.stats import chisquare
from scipy.stats import linregress
from scipy.optimize import curve_fit
import numpy as np
import sympy as sym

x = [44, 44, 44, 44, 44, 56, 56, 56, 56, 56, 100, 100, 123, 123, 123, 123,
123, 136, 136, 136, 136, 136, 164, 164, 164, 164, 164, 167, 167, 179, 179,
180, 180, 192, 192, 208, 208, 220, 220, 223, 223, 236, 236, 259, 259, 264,
264, 287, 287, 300, 300, 331, 331, 343, 343, 344, 344, 356, 356, 387, 387,
400, 400, 423, 423, 523, 523]
# threadpool
y = [11.532, 10.124, 10.747, 12.197, 11.309, 14.896, 14.325, 12.065, 11.881,
12.166, 22.848, 22.032, 27.955, 30.03, 25.695, 27.686, 26.64,  20.33, 21.669,
25.537, 22.108, 24.791, 36.517, 34.597, 36.072, 34.123, 41.216, 38.059, 35.528,
34.703, 35.511, 32.336, 31.133, 33.725, 32.479, 45.053, 42.677, 49.695, 41.647,
45.823, 47.501, 39.368, 39.028, 48.572, 48.818, 61.342, 51.799, 58.205, 58.691,
59.016, 54.243, 65.301, 69.586, 68.422, 73.126, 65.89, 65.878, 65.257, 60.926,
79.514, 75.817, 76.444, 71.243, 77.773, 82.464, 102.171, 106.073]
# taskflow
z = [12.343, 11.972, 12.766, 11.555, 13.913, 31.169, 30.268, 27.459, 26.185,
28.377, 24.045, 22.855, 31.169, 30.268, 27.459, 26.185, 28.377, 22.702, 22.912,
22.605, 22.768, 24.344, 36.891, 36.503, 37.308, 37.527, 37.117, 36.596, 38.103,
37.558, 44.568, 35.189, 31.980, 32.281, 31.634, 43.949, 44.051, 52.840, 57.936,
47.512, 46.563, 43.428, 46.102, 49.053, 47.725, 54.716, 62.030, 76.841, 60.557,
57.277, 57.158, 71.441, 74.983, 68.794, 77.741, 66.508, 64.498, 65.112, 73.105,
89.003, 85.054, 79.198, 76.403, 87.435, 81.604, 101.536, 101.318]

x = np.array(x, dtype=float); y = np.array(y, dtype=float)
def func_linear(x, a, b, c, d):
    return x
popt, pcov = curve_fit(func_linear, x, y, bounds=(0, 1))

xx = np.linspace(0, 107, 1000)
plt.title("Scatter plot comparing threadpool vs taskflow")
plt.plot(y, z, 'ro', label="Data", color='blue')
#plt.plot(y, func_linear(x, *popt2), 'ro', label="4.16*x", color='darkblue')
plt.xlabel('Threadpool values', fontsize=16)
plt.ylabel('Taskflow values', fontsize=16)
plt.plot(xx, func_linear(xx, *popt), '--k', label="Diagonal", color='green')
plt.legend(loc='upper left')
plt.show()

print(linregress(y, z))
```

Fig. 42: Scatter plot program for comparing threadpool and taskflow PoC's.