D6.3 Final report on the Experimental Execution Environment

Due Date	Month 36
Delivery	Month 36
Lead Partner	BADW-LRZ
Dissemination Level	Public
Status	Final
Approved	Internal review
Version	1.0

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 671564.

DOCUMENT INFO

Date and version number	Author	Comments
03.09.2018 v0.1	Vytautas Jančauskas	First draft
19.09.2018 v0.2	Vytautas Jančauskas	Second draft, after internal reviewer comments
25.09.2018 v1.0	Vytautas Jančauskas	Final revision

CONTRIBUTORS

Contributor	Role
Vytautas Jančauskas	WP6 leader and author of this deliverable
Tomasz Piontek	WP5 leader/ internal reviewer
Saad Alowayyed	WP2 contributor/ internal reviewer

TABLE OF CONTENTS

1 Executive summary

This deliverable is an update to the deliverable D6.2. Here we provide up to date information about the state of the software and hardware infrastructure that the Experimental Execution Environment (EEE) of the ComPat project consists of. We also highlight new additions to the EEE – namely the queue wait time and performance prediction databases and services. The aim of Work Package 6 (WP6) is to provide an Experimental Execution Environment (EEE) for the project's scientists and external users of ComPat tools. EEE aims to be a collection of software, computing resources and custom developed tools that allows users to submit simulations to be performed at the projects' computing sites. To this end, the sites have to provide a common software infrastructure and the EEE must have a single, unified job submission system. The submission system we use is QCG (https://www.qoscosgrid.org/), developed at the Poznan Supercomputing and Networking Center. The uniform software infrastructure is provided using environment modules (http://modules.sourceforge.net/). ComPat specific naming conventions are used for these modules to achieve consistency. It is important for the goals of the project that the user does not know or care in which particular site their software is run. The computing sites and computing nodes on those sites have to be automatically chosen in order to optimise energy consumption and in order to best suit a particular application.

Some new functionalities (like multi-criteria brokering) have been added to other already existing services like QCG-Broker or QCG-Computing. New versions have been deployed on EEE. Please refer to D5.3 for details. We have also developed and integrated several important new services into the EEE. These are the queue wait time prediction and performance prediction services. The queue wait time prediction service predicts, based on historical data, the amount of time that a job will spend in the scheduler queue. This will depend on various factors, such as the current system use, how many jobs the user has queued, etc. The EEE tracks this information for the sites participating in the project. Bayesian inference techniques are then used to estimate the probability that a job will spend a given time in the queue. This is then provided as a service to the rest of the infrastructure. The service API is provided in detail in this deliverable. A paper describing the statistical methods behind the prediction was submitted for publication. We provide it here as an appendix.

Performance prediction database is hosted at LRZ and was developed along with WP4. For each run submitted to the EEE we store information about energy consumption and runtime in the database. If a new job is submitted and there is information about a similar job in the database the performance data is then interpolated using a model fitted over the historical data. This information is then used to select job plans and execute part of the simulation on resources in a way that optimizes energy consumption.

2 The Experimental Execution Environment (EEE)

The EEE is a collection of hardware resources, software modules and policies that make up a unified environment that the users of the project can access. EEE provides unified access to the computing resources using grid certificates. A user gets access to all systems participating in ComPat with the same grid certificate. Furthermore, the users will find the same software infrastructure on all computing sites. This is accomplished by maintaining a set of common software modules across all systems. Brokering services are provided by QCG developed by PSNC. These allow the users to submit jobs in a convenient way, without having to worry about the scheduling system at a particular site. More about it in the D5.3 deliverable. On each site data is collected about every job submitted to the EEE. From this data we do various things, such as performance and queue wait time prediction which then are used to influence the decisions of the brokering service as to where to send various parts of a multi-scale simulation.

3 EEE hardware infrastructure

Currently, the hardware for the EEE is provided by three supercomputing sites. We summarize the available machines below. One of the goals of the project is to find ways to assign computing jobs to machines that are most suited for executing them – be it in terms of energy efficiency or compute time, or both. Therefore, a heterogeneous hardware stack is an advantage.

3.1 LRZ

Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities (LRZ) makes SuperMUC (Figure 1: SuperMUC) available to the project. SuperMUC currently consists of two phases (physically separate subsystems) – Phase 1 and Phase 2. Both phases are available to the project.

Figure 1: SuperMUC

Summary of hardware available in each phase is given below. SuperMUC uses the LoadLeveler job scheduling system by IBM.

Table 1: LRZ Hardware

	Phase 1			**Phase 2**
Installation date	2011	2012	2013	2015
Processor	Westmere-EX Xeon E7-4870 10C	Sandy Bridge-EP Xeon E5-2680 8C	Ivy-Bridge (IvyB) and Xeon Phi 5110P	Haswell Xeon Processor E5-2697 v3
Number of nodes	205	9,216	32	3,072
Number of cores	8,200	147,456	3,840 (Phi)	86,016
Peak performance (PFlop/s)	0.078	3.2	0.064 (Phi)	3.58
Memory (TB)	52	288	2.56	194
Power consumption (MW)	<2.3			~1.1

3.2 PSNC

Poznan Supercomputing and Networking Center (PSNC) make the Eagle machine available to the project (*Figure 2: Eagle*). A summary of it's capabilities is provided in the table. Eagle uses SLURM for job scheduling.

Figure 2: Eagle

Table 2: PSNC Hardware

	Eagle	**Inula**
Processor	Intel Xeon E5-2697	AMD (Interlagos), Intel Xeon E5-2697
Number of nodes	1,233	681
Number of cores	32,984	3,360
Peak performance (PFlop/s)	1.4	0.1384
Memory (TB)	120.6	6.56
Power consumption (MW)	n/a	n/a

3.3 STFC

Science & Technology Facilities Council's Hartree Centre provides their Neale machine (*Figure 3: Neale*) to the ComPat project. Neale is an immersion cooling (environmentally friendly) machine.

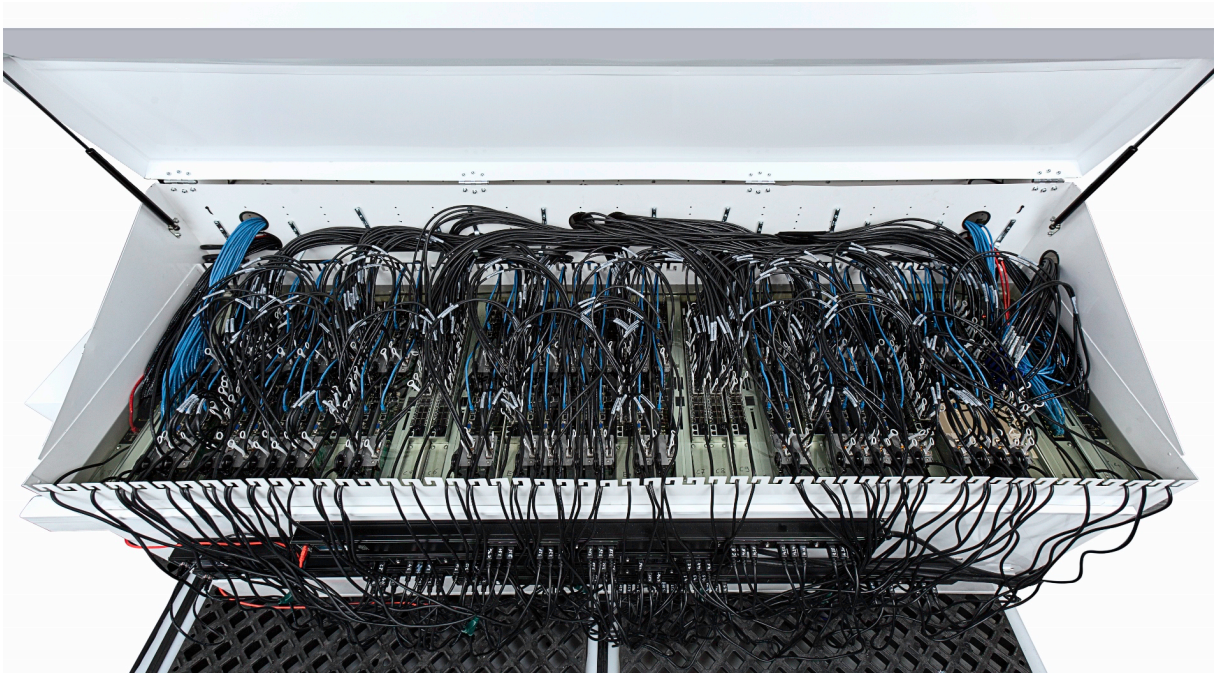

Figure 3: Neale

Table 3: STFC Hardware

	Neale
Processor	Intel Xeon (Ivy Bridge E5-2650v2. 2.6GHz)
Number of nodes	120
Number of cores	1,920
Peak performance (PFlop/s)	n/a
Memory (TB)	7.68
Power consumption (MW)	n/a

4 EEE software infrastructure

In the following sections, we provide the current state of software organisation and software solutions we have developed. Primary focus is to provide a consistent user experience between computing sites and allow for easy deployment of software on those sites. We monitor all relevant software services running on our sites using Nagios (*Figure 4: Nagios*). Nagios (https://www.nagios.org/) is a software system for monitoring computer infrastructure. It uses various (including user defined) probes to determine if computer systems are online and functioning properly.

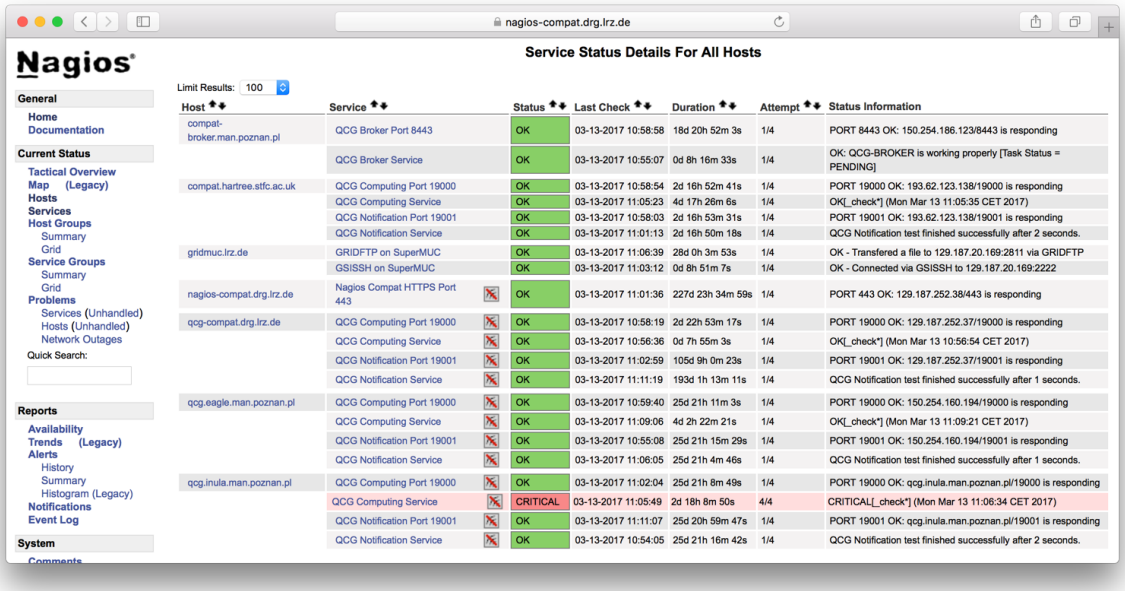

Figure 4: Nagios in Compat

4.1 EEE module system and user software infrastructure

The computing resources of the ComPat project are split between different computing sites located in different organizations and different countries. The users of these resources should not be expected to

adapt to each site to run their software. One of the goals of the project is to automatically run jobs on hardware most suited for that particular job. To this end, a unified software infrastructure is needed across the sites. This means: commonly used libraries have to be built the same way and provided using the same naming conventions. This unified software infrastructure is provided via environment modules (http://modules.sourceforge.net/). On each computing site, software is organized in to an agreed upon directory structure. The users are meant to follow the directory structure when building their own software. Commonly used libraries and tools that are specific to the project also have associated module files for easy access. This partly relies on sites providing a shared directory that can be accessed by all the users that are part of the project. This directory can be accessed via the `$COMPAT_SHARED` variable. The modules currently available are described in *Table 4*.

Table 4: EEE Modules

Module	Description
compat	Will load ComPat environment variables, for example $OMPAT_SHARED, which is used to access the ComPat shared directory.
compat/common/ruby	RUBY programming language needed by MUSCLE2.
compat/common/muscle2	MUSCLE 2 - Multiscale Coupling Library and Environment is a portable framework to do multiscale modeling and simulation on distributed computing resources.
compat/common/amuse	AMUSE - Astrophysical Multipurpose Software Environment.
compat/common/namd	NAMD - a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems.
compat/common/amber	AmberTools – a package of molecular simulation programs.
compat/apps/bio/flow	Blood flow simulation software.
compat/apps/bio/ISR2D	In-stent restenosis simulations in 2D.
compat/apps/bio/ISR3D	In-stent restenosis simulations in 3D.
compat/apps/fusion	Software for fusion simulations.
compat/common/allinea	Allinea profiling and debugging tools. They are used to measure energy consumption of multi-scale simulations in ComPat.

Here is the relevant section of the `module avail` command. It shows the modules and versions of software that are installed on EEE as of time of writing.

```
------------------- /gpfs/work/pr92ge/di25pul/Modules --------------------
compat/1.0(default)
compat/apps/bio/flow/1.0(default)
compat/apps/bio/ISR2D/1.0(default)
```

```
compat/apps/bio/ISR3D/1.0(default)
compat/apps/fusion/1.0
compat/apps/fusion/1.1
compat/apps/fusion/1.2
compat/apps/fusion/1.3
compat/apps/fusion/1.4(default)
compat/common/allinea/7.1
compat/common/allinea/18.0
compat/common/allinea/18.1.1
compat/common/allinea/18.2(default)
compat/common/amber/16
compat/common/amuse/089e701
compat/common/muscle2/612248f
compat/common/muscle2/compat-1.1(default)
compat/common/muscle2/compat-1.2
compat/common/namd/2.10
compat/common/ruby/1.9.3(default)
compat/dev/fusion/1.0(default)
```

4.2 Service availability statistics

Key performance indicators (KPIs) for measuring service availability were proposed in D6.1. In order to calculate the value of these performance indicators, we need to track service availability. This is done via Nagios. We implemented a database that is updated by a `cron` job that captures Nagios state and writes it to the database. The job is executed every hour and writes a record that shows the state of the service along with other metadata. This gives us an option to calculate service availability statistics in various ways. One such way measures the percentage of time that the service was functioning correctly over a user specified period of time. That allows us to estimate the quality of service, availability in the project and highlights problem areas. We provide a web service that is accessible to all members of the project:

https://nagios-compat.drg.lrz.de:5000/performance/dd.mm.yyyy/dd.mm.yyyy

To access the indicators, users indicate the start and end dates of the period they are interested in in the template URL. Service availability statistics are calculated for that period and plots are generated and downloaded to the browser. An example of that is given in (*Figure 5: Service availability statistics*). This shows service availability from 6 April 2017 to 1 August 2018. The start of this period corresponds to when we have finally finished integrating various sites in to the EEE. Namely, we have finished integrating the Neale machine at Hartree in March of this year. Please note that each Nagios monitoring probe has its own pie chart in the figure. The low availability of the Inula machine is explained by the fact that it has been decommissioned and removed from the EEE. There were also issues with the probe we have used to monitor the Eagle machine at PSNC which also results in a low availability figure. In reality it was available to the users through most of the project period without major interruptions.

Figure 5: Service availability statistics

5 Queue Wait Time and Performance Prediction Services

The largest amount of work during the last 18 months of the project with regards to the EEE went into developing the queue wait time prediction and performance prediction services. The goal of the queue wait time prediction service is to predict the amount of time that a job will stay in the batch system queue. This is needed by the multi-criteria brokering service that is described in the deliverable D5.3. To this end, we have extended the EEE with capabilities that allow us to store information about each job submitted to each of the computing sites. From this information, we have used machine learning and statistical techniques to estimate the probabilities that a job will stay in a queue a given amount of time. These probabilities are dependent on the resources requested by the job submitted and the state of the system at the given moment. This information is taken into account when predicting queue wait time probabilities. The performance prediction service uses curve fitting to fit resource consumption models to predict job runtime. First, a model is fitted using historical data and then we estimate job runtime from this model. In this section, we will discuss how these two services fit into the EEE.

5.1 Queue Wait Time Prediction Service

The queue wait time prediction service consists of the following three main components: scheduling system monitor service, database and prediction service. The purpose of the monitoring system is to poll the scheduler for information about recently submitted, running and finished jobs. This information is then stored in the database. From the data in the database, predictions are made and provided to the user via a web service. In the following subsections, we will discuss each part in detail.

5.1.1 Data Collection Service

For each site participating in the project, we collect information about the scheduler state. Thus, we collect data from all the sites participating in the project. Data collection is done by parsing the output of various scheduler tools. The tools and their output will depend on the resource manager (e.g. SLURM, LoadLeveler, PBS) used on that system. Therefore, for each scheduler (so far it is only SLURM and LoadLeveler) we had to implement a separate data gathering script. All of them work in the following way:

- The system is monitored until a new job is found in the list of jobs in the system. These are retrieved using such commands as `llx` on LoadLeveler or `squeue` on SLURM or the equivalent on other systems. The jobs are identified by their ids. The moment a job with an unknown id appears it is added to the in-memory list of jobs and its status is tracked.
- Once the job starts running, the amount of time it has spent in the queue is recorded.
- Once a job is finished, information about it is added to the database. The exact information stored and the technical details will be described in subsection 5.1.2.

The service polls the schedulers every minute. From the technical point of view this is done by logging in to each system using the grid certificate, executing command that displays the scheduler state and parsing the results. The grid certificates for each system are already stored on the virtual machine we use for this, since it is the same machine we use for monitoring the state of various services that make up the EEE.

A short side, not about the procedure above, since we rely on historical data for the predictions, it might seem natural instead of gathering the data ourselves to rely on the logs of the scheduling systems. There are a few problems with that approach as far as this work is concerned. First of all, the logs are generally not available publicly therefore access to them has to be negotiated with the administrators at each of the sites. Depending on the policies in place at the given site the administrators may not agree to provide access to the logs due to, for example, privacy issues. Another issue is that we need this information in real-time. This is because we need to evaluate the state of the system at the time of job submission.

5.1.2 Queue Wait Time Prediction Database

We use a simple SQL database with one table to track the jobs. In practice, this proved a good choice since it allows for efficient pre-processing in the form of SQL queries. The following information about jobs is stored:

- **(1-8) Job attributes**
  - **site** (string) - the name of the site to which the job was submitted
  - **username** (string) - name of the user
  - **class** (string) - queue/class/partition name
  - **when_submitted** (integer) - UNIX timestamp for when the job was submitted
  - **time_requested** (integer) - how much wallclock time the user has requested
  - **cpu_time_requested** (integer) - how much cpu time the user has requested
  - **tasks_requested** (integer) - how many tasks (cpus) the user has requested
  - **nodes_requested** (integer) - how many nodes the user has requested
- **(9 - 21) System attributes** - describe the state of the whole cluster
  - **sys_jobs_running** (integer) - how many jobs are currently running
  - **sys_jobs_queued** (integer) - how many jobs are currently queued for execution
  - **sys_tasks_running** (integer) - how many tasks are currently running (how many cpus are currently used)
  - **sys_tasks_queued** (integer) - how many tasks are currently queued
  - **sys_nodes_running** (integer) - how many nodes are currently running
  - **sys_nodes_queued** (integer) - how many nodes are currently queued
  - **sys_time_running** (integer) - the sum of wallclock time requested by the jobs already running

    - **sys_time_used** (integer) - the sum of time already consumed by running tasks (the sum of time the tasks are already running for)
    - **sys_time_queued** (integer) - the sum of wallclock time requested by the jobs that are currently waiting for execution
    - **sys_cpu_time_running** (integer) - the sum of CPU time requested by the jobs that are currently running
    - **sys_cpu_time_used** (integer) - the sum of CPU time already used up by the running jobs
    - **sys_cpu_time_queued** (integer) - how much CPU time is currently queued
- **(22 - 34) Class attributes** - describes the state of the queue or partition. These are the same as system state variables but with class prefix instead of sys prefix. The meaning of attributes is analogous. For example **class_tasks_running** means how many tasks are currently running in that queue.
- **(35 - 47) User attributes** - describes the state of the user (information about the user's jobs). Again, identical to sys and class attributes in meaning, but describes jobs currently running, queued, etc. for particular users. The prefix is user instead of sys or class. For example, **user_tasks_running**.

We have used a MySQL database to store this data. The database server runs on a high availability virtual machine dedicated to that purpose. The database will not be accessed directly by either the EEE users or services, however, the data is only accessed indirectly via the queue wait time prediction service.

These are the queues for which data about jobs is stored:

- `SuperMUC (LRZ)`
    - `general`
    - `test`
    - `fat`
    - `fatter`
    - `tmp2`
    - `large`
    - `fattest`
    - `special`
- `Eagle (PSNC)`
    - `standard`
    - `plgrid`
    - `plgrid-long`

- bigmem
- plgrid-testing
- fast
- epyc
- centos7

Here are some statistics concerning the state of the database:

Table 5: Basic queue wait time statistics for SuperMUC at LRZ

Name	Number of jobs	Average Queue Wait Time
General	36983	08:41h
Test	54670	00:22h
Fat	39008	05:14h
fatter	682	12:18h
tmp2	3962	00:38h
large	2542	27:00h
fattest	2696	00:20h
special	68	00:02h

Table 6: Basic queue wait time statistics for Eagle at PSNC

Name	Number of jobs	Average Queue Wait Time
standard	1963794	00:23h
plgrid	55042	00:27h
plgrid-long	848	01:38h
bigmem	3416	09:16h
plgrid-testing	942	01:08h
fast	23071	01:54h
epyc	39	02:05h
centos7	41	08:04h

We started collecting the data on 16 June 2017. Over time, a large amount of data was collected. The database is now several gigabytes in size. We can use statistical techniques to deduce information about potential queue wait times from this data. For exact details, please consult the paper that is attached to this document as an annex.

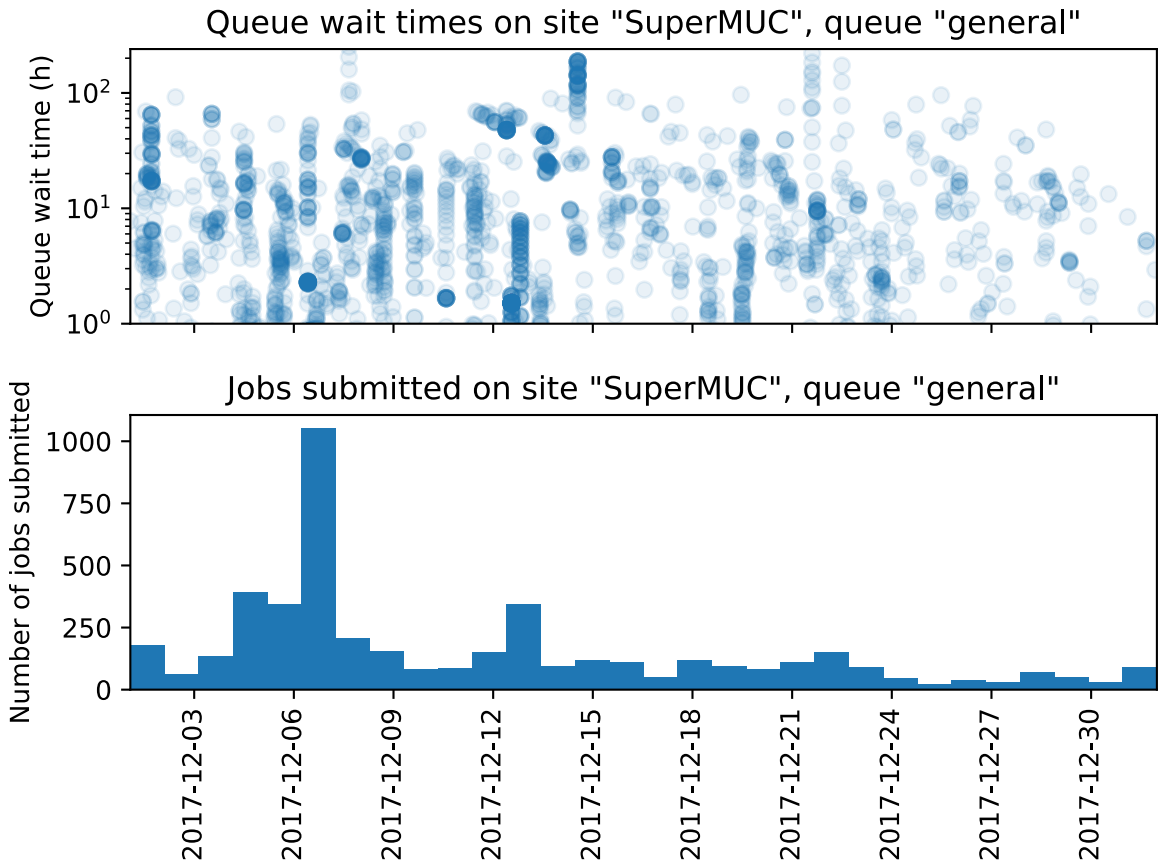

Figure 6: Queue Wait Times

In *Figure* ***6***, we see queue wait times plotted. The time range selected was from December 2017 to January 2018 for purposes of illustration. Each job submitted will have a record in the database consisting of all the fields described above. In the first figure, we see the queue wait times of submitted jobs over that period. The times are plotted in logarithmic scale. The second figure shows the number of jobs submitted each day over that time period. Similar data can be found for each queue.

The full description of the API that is used to communicate with the queue wait time prediction service is provided in the annex. So is the paper we have prepared, that details the various aspects of the operation of this service.

5.2 Performance Prediction Service

The performance prediction service is used to model application run-time (and thus energy consumption). It does this by fitting mathematical models of expected application performance to historical data we collect when application are submitted to the EEE. This service is used to evaluate multi-scale simulation execution plans in order to minimize energy consumption. Details are to be found in the D4.3 deliverable. Here we consider aspects of the service that are relevant to the EEE.

The performance prediction database consists of the following tables. It is stored in the form of MySQL database on a high availability virtual machine at LRZ. Communication with the database is done using the SQLAlchemy ORM for Python. Here we describe the technical details of the database, such as the database schema and what type of information is stored here. For higher level details please refer to the deliverable D4.3 (the corresponding WP4 deliverable). See the database schema in Figure 7. The tables are described in more detail in the following subsections.

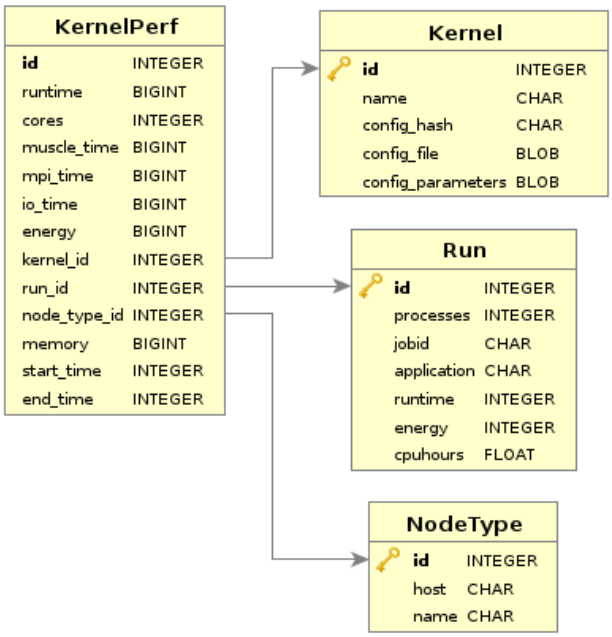

Figure 7: Performance database schema

5.2.1 Run

The primary record in the database is a 'Run', this represents a conceptual simulation. Each run may contain multiple kernel executions, but is contained within a single QCG job, the 'jobid' field is unique and taken from the QCG Job ID environment variable during the run. A number of the fields in the Run table are aggregated from the associated kernel runs. For example, the energy field represents the total energy consumption of the QCG job, however, depending on the execution pattern of the kernels this is not just a linear summation. Therefore, we must build logic into aggregation depending on the application, and how it was executed.

Name	Type	Description
`processes`	Integer	Number of processes launched by the run.
`jobid`	Integer	QCG job id.

application	String	Application name (e.g. FUSION).
runtime	Integer	Runtime in seconds.
energy	Integer	Energy in Watts.
cpuhours	Float	How many CPU hours the job used.

5.2.2 NodeType

This table contains the types of nodes available. They are linked to the 'KernelPerf' table in order to track per node energy consumption.

Name	Type	Description
Host	String	Site that hosts this node type.
Name	String	Node type name.

5.2.3 Kernel

The entries in the 'Kernel' table are more than just a single kernel (e.g. the Gem kernel for the Fusion application), we also store the problem specific parameters against each entry. We do this in two ways, firstly we construct a hash of the problem, using an MD5 hash algorithm on the configuration files to the kernel (e.g. gem.xml and unified.cxa.rb for the Gem kernel). This hash allows us to perform a uniqueness test when rerunning problems – have we seen this exact problem configuration before (but perhaps on a different system or core count). For more details, we refer to deliverable D4.3.
Further, we try to extract key performance parameters from these configuration files – such as iteration count, mesh size, spatial information, which will impact performance. This information can then be used in the performance prediction phase, discussed in the deliverable D4.3. We note that this parameters field is stored as a BLOB type as we populate it with the JSON object representing the key value pairs for the parameters. The motivation is that the available parameters are specific to the individual kernel being profiled, and unlikely to be common between them. This data format makes it easy to interpret these values during the analysis stage for performance prediction.

Name	Type	Description
name	String	Kernel name.
config_hash	String	The hash value of the concatenated configuration files used to run this kernel.
config_file	String	The configuration files used to run this kernel.
config_parameters	String	The parameters relevant to runtime.

5.2.4 KernelPerf

This table contains the performance information from a specific application run, it is reduced to the key performance metrics of relevance, including ComPat specific metrics such as MUSCLE2 time.

Name	Type	Description
runtime	Integer	Run time in seconds.
cores	Integer	Number of cores used by th kernel.
muscle_time	Integer	MUSCLE time in seconds.
mpi_time	Integer	MPI time in seconds.
io_time	Integer	I/O time in seconds.
energy	Integer	Energy consumed by the kernel.
kernel_id	Integer	Kernel id.
run_id	Integer	Run id.
node_type_id	Integer	Node id.
memory	Integer	Memory used by the kernel in bytes.
start_time	Integer	Start time in seconds.
end_time	Integer	End time in seconds.

Each performance entry is made up of a run of a known kernel on a known node type, so this information is stored in the Kernel and NodeType tables respectively

6 Conclusions

Before the month 36, we have successfully achieved a unified and stable execution environment, built on top of resources provided by our partners. We have also prepared and made available to project users the EEE users' manual. The manual is actively being updated by project members. The queue wait time prediction and performance prediction databases were integrated into the EEE and track queue wait time and energy/time consumption of each of the jobs submitted to the system. The queue wait and performance data is then used to predict queue wait time and performance respectively. In case of queue wait time prediction, we have used statistical techniques (namely Bayesian classifiers) to calculate the probability that a job starts within a time provided by the user. In case of performance prediction we fit performance models to the data and use those models to interpolate the values for new jobs. This work was done in conjunction with WP4 for the performance prediction and WP5 for the queue wait time prediction.

We also track service availability, we have a large library of ComPat specific modules on each of our partner sites. These modules are used to achieve a uniform environment for running simulations. In other words, jobs can run unaltered independently of the computing site. QCG is deployed in each of the computing sites and provides a middleware layer to the project.

7 Annexes

In this Annex we describe the queue wait time prediction service API. The paper describing the queue wait time prediction service is also attached to this Annex. The paper was submitted for publication to the Philosophical Transactions of the Royal Society issue on "Multiscale Modelling, Simulation & Computing: from the Desktop to the Exascale" on the 18th of June 2018.

7.1.1 Service API

The queue wait time prediction service will receive a request from the client, be it a person or another service, and return a response. The request contains information about the resources needed and constraints for the queue wait time. The response contains information about queues that fulfil the requirements for the nodes and satisfy queue wait time constraints. The constraints can be specified in several different ways, depending on the specific usecase. We will describe the request and response formats in this subsection. The service uses JSON for its API and the formats are given in the JSON Schema format (http://json-schema.org/). The schema fully define the service as seen by its users.

7.1.1.1 Request

The schema for the request to the service is given below. All inputs to the service are validated against this schema and violations of it will result in an error.

```
{
  "type" : "object",
  "properties" : {
    "user" : {"type" : "string"},
    "queue" : {"type" : "string"},
    "site" : {"type" : "string"},
    "memory" : {"type" : "integer"},
    "grant" : {"type" : "string"},
    "walltime" : {"type" : "integer"},
```

```
    "nodes" : {
      "type" : "array",
      "items" : {
        "type" : "object",
        "properties" : {
          "node_properties" : {
            "type" : "array",
            "items" : {"type" : "string"}
          },
          "n_cores" : {"type" : "integer"},
          "n_nodes" : {"type" : "integer"}
        }
      }
    },
    "time" : {"type" : "integer"},
    "certainty" : {"type" : "number"}
  },
  "required" : ["user", "nodes"]
}
```

The meanings of the properties `user`, `site` and `queue` are the user name, the name of the computing site and the queue name respectively. The queue name is optional. The optional property `memory` contains information about how much memory does the job require in bytes. This will be useful if the user only needs one core, but all of the nodes memory, for example. The property `grant` specifies the grant name where applicable. The grant (also may be called project in certain systems) may limit access to certain queues. The property `walltime` describes the requested walltime in seconds. The dictionaries stored in the `nodes` array describe the requirements for computing resources. In the simplest case, it will contain the description of a single type of node that fulfills the requirements for running a specific job. In the general case, it can contain a request for heterogeneous resources. Each node is described by a dictionary consisting of an array `node_properties` that contains arbitrary strings describing that node, number of required cores - `n_cores` and number of required nodes - `n_nodes`. Properties `time` and `certainty` provide two different ways of specifying constraints for queue wait time. The property `time` is used to specify time in seconds. A number from 0 to 1 specifies `certainty`. Either one of them or both or neither can be used. There are four different possibilities. In the following example, we consider all of the different possibilites.

An example request using this API might look like the following:

```
{
  "user" : "username",
  "walltime" : 3600,
  "nodes" : [
    {
      "node_properties" : ["intel", "haswell",
                           "haswell_2600mhz",
                           "fdr", "huawei", "128GB"],
      "n_cores" : 128,
      "n_nodes" : 8
    },
    {
      "node_properties" : ["phi"],
      "n_cores" : 512,
      "n_nodes" : 8
    }
  ],
  "certainty" : 0.9
}
```

In this case, we request two different types of nodes. We specify that we want the lowest queue wait times for which the certainty is 0.9. We also specify that we expect the job to complete in one hour. The service will gather all of the other information needed to do the prediction by querying the batch system (the job scheduler) in real time. Therefore, the user needs to supply very little information about their job.

7.1.1.2 Response

Suppose we have three queues: "queue 1", "queue 2" and "queue 3". The service uses three discrete time points when doing the prediction: 1min, 1h and 24h. The probabilities that a specific job (described by the user and also taking into account current system state) will start before 1min, 1h and 24h are then calculated. Suppose the results are as shown in *Table 7*.

Table 7: Queue Wait Time Prediction Example

Queue	1min	1h	24h
queue 1	0.1	0.8	0.9
queue 2	0.8	0.85	0.9
queue 3	0.1	0.2	0.9

If the user does not specify either the certainty or time, the output of the service will contain all of the information in *Table 7*. If the user specifies only the certainty (say 0.8), then the results will contain the row for queue 2, since it provides the lowest queue time with that probability. If the user only specifies time (say 1 minute), the output will consist of the row corresponding to queue 2 as well, since it provides the highest certainty for that time point. If the user specifies both the certainty and time, the service will attempt to find queues that can start executing the job before that time with that or higher certainty. Therefore, if the user specifies 1 hour and certainty of 0.8, the service will return the rows for queue 1 and queue 2.

The schema for the response follows. The service always returns an array of dictionaries, one for each queue. The dictionaries will contain probabilities for each time point for that queue.

```
{
  "type" : "object",
  "properties" : {
    "queues" : {
      "type" : "array",
      "items" : {
        "type" : "object",
        "properties" : {
          "site" : {"type" : "string"},
          "name" : {"type" : "string"},
          "certainties" : {
            "type" : "array",
            "items" : {
              "type" : "object",
              "properties" : {
                "time" : {"type" : "integer"},
                "certainty" : {"type" : "number"}
              }
            }
```

```
          }
        }
      }
    }
  },
  "required" : ["queues"]
}
```

The exact response format will depend on whether `certainty` and `time` properties are defined. There are four different possibilities. If both `certainty` and `time` are specified, then the service returns a list of queues that can guarantee (statistically at least) that the queue wait time will be lower than `time` with the given `certainty`. If only `certainty` is specified, then the service will return all the queues and for each queue the minimum time that it can guarantee with that certainty (or the closest higher certainty). If only `time` is specified, then the service will return certainties for that time for each queue. If neither is specified, it will return full information for all the queues.

We present an example of what a response from the service might look like below. Here neither `time` nor `certainty` was specified.

```
{
  "queues" :
    [
      {
        "name" : "queue_1",
        "certainties" : [
          {"time" : 0, "certainty" : 0.3},
          {"time" : 600, "certainty" : 0.5},
          {"time" : 3600, "certainty" : 0.7},
          {"time" : 21600, "certainty" : 0.8},
          {"time" : 86400, "certainty" : 0.99}
        ]
      },
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 0, "certainty" : 0.6},
          {"time" : 600, "certainty" : 0.8},
```

```
          {"time" : 3600, "certainty" : 0.9},
          {"time" : 21600, "certainty" : 0.95},
          {"time" : 86400, "certainty" : 0.99}
        ]
      },
      {
        "name" : "queue_3",
        "certainties" : [
          {"time" : 0, "certainty" : 0.1},
          {"time" : 600, "certainty" : 0.2},
          {"time" : 3600, "certainty" : 0.8},
          {"time" : 21600, "certainty" : 0.9},
          {"time" : 86400, "certainty" : 0.99}
        ]
      }
    ]
}
```

If `time` was specified and the value was 600, for example, the response will look like the one given below.

```
{
  "queues" :
    [
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 600, "certainty" : 0.8}
        ]
      }
    ]
}
```

If `certainty` was specified as 0.9 and `time` was omitted the response would look like the following.

```
{
  "queues" :
```

```
    [
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 3600, "certainty" : 0.9}
        ]
      }
    ]
}
```

If `time` was specified as 600 and `certainty` as 0.9, the service would try to find the queue that can guarantee 600 second wait time with 0.9 certainty. Since such a queue does not exist, it would return an empty list:

```
{
  "queues" :
    [
    ]
}
```

PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:
Multi-Scale computing

Keywords:
Multi-Scale computing; machine learning; Naive Bayes

Author for correspondence:
Vytautas Jancauskas
e-mail: jancauskas@lrz.de

Predicting Queue Wait Time Probabilities for Multi-Scale Computing

Vytautas Jancauskas[1], Tomasz Piontek[2], Piotr Kopta[2], Bartosz Bosak[2]

[1]Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Boltzmannstraße 1, 85748 Garching near Munich, Germany

[2]Poznan Supercomputing and Networking Center, Institute of Bioorganic Chemistry of the Polish Academy of Sciences, ul Z. Noskowskiego 12/14, 61-704 Poznan, Poland

We describe a method for queue wait time prediction in supercomputing clusters. It was designed for use by brokering mechanisms for resource selection in a multi-site HPC environment. The aim is to incorporate the time jobs stay queued in the scheduling system into the selection criteria. Our method can also be used by the end users to estimate the time to completion of their computing jobs. It uses historical data about the particular system to make predictions. It returns a list of probability estimates of the form (t_i, p_i) where p_i is the probability that the job will start before time t_i. Times t_i can be chosen more or less freely when deploying the system. Compared to regression methods that only return a single number as a queue wait time estimate (usually without error bars) our prediction system provides more useful information. The probability estimates are calculated using the Bayes theorem with the naive assumption that the attributes describing the jobs are independent. They are further calibrated to make sure they are as accurate as possible, given available data. We describe our service and its REST API and the underlying methods in detail and provide empirical evidence in support of the method's efficacy.

THE ROYAL SOCIETY PUBLISHING

1. Introduction

The issue of queue wait times comes up in many situations in HPC. We define queue wait time as the number of seconds between the time a job gets submitted to the resource manager (SLURM, PBS, LoadLeveller, etc.) and the time that the job starts executing. There is a trade-off here, in the sense that the more resources are requested, the faster will the computing job be finished, however, it will take more time for the resource manager to find the required resources. There can be many situations in which the knowledge of the queue wait time is desirable. The users, for example, are interested in the queue wait time because it counts towards the total time between submission of a simulation or other computing task and retrieving its results (time to completion). To the user the time spent waiting in the queue is no different from the time spent waiting until their simulation is done executing. Therefore, it is of interest to the user to have an estimate of how long their job will spend queuing before submitting it. Having this information, they may tweak the amount of resources they request in order to get it executing sooner. They may also consider submitting to another, less crowded, system. The issue of queue wait times can come-up in scenarios where jobs are submitted by automated systems such as HPC middleware. This is especially true for multi-scale simulations. It can also be beneficial in urgent computing [2] [10] [9] (when paired with preemption mechanisms).

The motivation for this work is the use of queue wait time prediction in brokering strategies. In this case the user specifies certain parameters that describe acceptable queue wait times and the brokering mechanism attempts to find a resource that satisfies those parameters. The brokering system that uses this service via the provided API is part of QCG [15] developed at the Poznan Supercomuting and Networking Center [14]. QCG is a set of middleware services and tools for HPC.

Multi-scale and multi-physics scenarios come up very often in scientific simulations [3] [22]. In the ComPat project [4] we attempt to distribute parts of multi-scale jobs across heterogeneous resources [1]. The resources may not be part of the same computing cluster or in the same geographical location. Parts of the coupled multi-scale simulation may require different type of computing resources to run on. Among the requirements are the constraints for the acceptable queue wait time. A typical usage scenario for such a service would be:

(i) A user submits node description with the requirements for computing resources.
(ii) The system finds resources/sites/clusters that satisfy those requirements.
(iii) For each queue we return information about the predicted queue wait times.
(iv) The user can decide which queue to choose in one of the following ways:

- Via a specification of acceptable probability and time. I.e. the user specifies that they require that their job starts within one hour with a certainty of 90%. If no such queue exists the job won't be submitted and an error will be returned;
- By specifying an acceptable probability, say 90%. The brokering algorithm will pick the option with the shortest queue wait time with that probability. The probability in this case represents how important is the queue wait time as a factor to the user. If the user specifies the probability 0.5 (instead of 0.9), then they might get a shorter queue wait time, but the risk will also be higher that the prediction is not reliable;
- Manually, having in mind all the information about the queue wait times and their probabilities.

There are several ways we could take to approach this problem. Batch schedulers often have functionality that allows them to estimate how much time will a submitted job stay in the queue. This estimate is adjusted as resources become available. However, the job has to be submitted first and estimates change with time. Therefore we cannot use this approach in the kind of scenarios we are interested in. We need a good estimate before submitting. We could use a regression approach and for each queue return an estimate for how long will the job queue. There are

various ways of doing this, starting from simply finding the most similar job in terms of Euclidean distance in attribute space to more elaborate symbolic regression methods. Without any kind of confidence analysis this would present the user with a single number for each queue. It is not quite clear how a user should interpret those numbers. In particular how much trust to put into them.

There is some literature already when it comes to tackling this problem. W. Smith [18] [19] [12] explores the use of execution and queue wait time predictions in order to select resources for executing applications on computational grids. The author relies on application run time prediction data in order to perform a simulation of the scheduler (in their case it is a simple "first come, first served" scheduler). This is not suitable for our needs. Getting an accurate estimate of program execution times would be really difficult across different systems with many users. Furthermore the schedulers are different and implement different scheduling strategies. Some of them quite complex (or impossible) to accurately simulate (e.g. fair-share scheduling). H. Li et al. [12] explore queue wait time prediction in a similar context as in our paper. They take a similar approach to [19] and simulate a specific scheduler using job run time predictions. However, as they assume the Maui scheduler [7] which is not widely used anymore, and as they base their work on simulations of the scheduler, their work cannot be applied in our case.

In this paper we propose a novel method based on the well known technique of Naive Bayes classification. The time is discretized to a certain granularity that is chosen when deploying the system. We divide time in pairs of intervals and solve the problem of assigning jobs to two classes - whether the job will start before time t_i or if it will start after. We use the Naive Bayes formula (Bayes theorem with the assumption that attributes are independent of each other) and probability calibration techniques to arrive at a probability for each of the two events. The output of the method is a list of tuples of the form (t_i, p_i) where t_i is time in seconds and p_i is the probability that the job will start before time t_i. The user then has a set of probabilities to manually examine or they can let the brokering algorithm choose the resources for them.

In Section 2 we describe the method itself. This includes the data collection techniques we have employed, numeric attribute discretisation and probability calibration. Each of these are important to the performance of our method. We then provide empirical evidence about the quality of the performance of our method in Section 3 and finally conclude our findings with Section 4.

2. Naive Bayes for Queue Wait Time Prediction

In this section we will explain the proposed method. The whole process is divided into three main subtasks - data collection, model building and the actual prediction service. We start by describing the service API (it is a web service) since it sets the requirements for the implementation. We then describe the methods we have used to satisfy those requirements.

(a) Service API

The service will receive a request from the client, be it a person or another service, and return a response. The request contains information about the resources needed and constraints for the queue wait time. The response contains information about queues that fulfill the requirements for the nodes and satisfy queue wait time constraints. The constraints can be specified in several different ways, depending on the specific usecase. We will describe the request and response formats in this subsection. The service uses JSON for its API and the formats are given in the JSON Schema format (`http://json-schema.org/`). The schemas fully define the service as seen by its users hence we have decided to provide it here in full.

(i) Request

The schema for the request to the service is given below. All inputs to the service are validated against this schema and violations of it will result in an error.

```
{
  "type" : "object",
  "properties" : {
    "user" : {"type" : "string"},
    "queue" : {"type" : "string"},
    "site" : {"type" : "string"},
    "memory" : {"type" : "integer"},
    "grant" : {"type" : "string"},
    "walltime" : {"type" : "integer"},
    "nodes" : {
      "type" : "array",
      "items" : {
        "type" : "object",
        "properties" : {
          "node_properties" : {
            "type" : "array",
            "items" : {"type" : "string"}
          },
          "n_cores" : {"type" : "integer"},
          "n_nodes" : {"type" : "integer"}
        }
      }
    },
    "time" : {"type" : "integer"},
    "certainty" : {"type" : "number"}
  },
  "required" : ["user", "nodes"]
}
```

The meanings of the properties `user`, `site` and `queue` are the user name, the name of the computing site and the queue name respectively. The queue name is optional. The optional property `memory` contains information about how much memory does the job require in bytes. This will be useful if the user only needs one core, but all of the nodes memory, for example. The property `grant` specifies the grant name where applicable. The grant (also may be called project in certain systems) may limit access to certain queues. The property `walltime` describes the requested walltime in seconds. The dictionaries stored in the `nodes` array describe the requirements for computing resources. In the simplest case it will contain the description of a single type of node that fulfills the requirements for running a specific job. In the general case it can contain a request for heterogeneous resources. Each node is described by a dictionary consisting of an array `node_properties` that contains arbitrary strings describing that node, number of required cores - `n_cores` and number of required nodes - `n_nodes`. Properties `time` and `certainty` provide two different ways of specifying constraints for queue wait time. The property `time` is used to specify time in seconds. A number from 0 to 1 specifies `certainty`. Either one of them or both or neither can be used. There are four different possibilities. Let us consider all of them with an example.

An example request using this API might look like the following:

```
{
  "user" : "username",
  "walltime" : 3600,
  "nodes" : [
```

```
    {
      "node_properties" : ["intel", "haswell",
                           "haswell_2600mhz",
                           "fdr", "huawei", "128GB"],
      "n_cores" : 128,
      "n_nodes" : 8
    },
    {
      "node_properties" : ["phi"],
      "n_cores" : 512,
      "n_nodes" : 8
    }
  ],
  "certainty" : 0.9
}
```

In this case we will request two different types of nodes. We specify that we want the lowest queue wait times for which the certainty is 0.9. We also specify that we expect the job to complete in one hour. The service will gather all of the other information needed to do the prediction by querying the batch system (the job scheduler) in real time. Therefore the user needs to supply very little information about their job.

(ii) Response

Suppose we have three queues: "queue 1", "queue 2" and "queue 3". The service uses three discrete time points when doing the prediction: 1min, 1h and 24h. The probabilities that a specific job (described by the user and also taking into account current system state) will start before 1min, 1h and 24h are then calculated. Suppose the results are as shown in Table 1.

Table 1. Example prediction

Queue	1min	1h	24h
queue 1	0.1	0.8	0.9
queue 2	0.8	0.85	0.9
queue 3	0.1	0.2	0.9

If the user does not specify either the certainty or time, the output of the service will contain all of the information in Table 1. If the user specifies only the certainty (say 0.8), then the results will contain the row for `queue 2`, since it provides the lowest queue time with that probability. If the user only specifies time (say 1 minute), the output will consist of the row corresponding to `queue 2` as well, since it provides the highest certainty for that time point. If the user specifies both the certainty and time, the service will attempt to find queues that can start executing the job before that time with that or higher certainty. So if the user specifies 1 hour and certainty of 0.8, the service will return the rows for `queue 1` and `queue 2`.

The schema for the response follows. The service always returns an array of dictionaries, one for each queue. The dictionaries will contain probabilities for each time point for that queue.

```
{
  "type" : "object",
  "properties" : {
    "queues" : {
      "type" : "array",
      "items" : {
        "type" : "object",
```

```
        "properties" : {
          "site" : {"type" : "string"},
          "name" : {"type" : "string"},
          "certainties" : {
            "type" : "array",
            "items" : {
              "type" : "object",
              "properties" : {
                "time" : {"type" : "integer"},
                "certainty" : {"type" : "number"}
              }
            }
          }
        }
      }
    }
  },
  "required" : ["queues"]
}
```

The exact response format will depend on whether `certainty` and `time` properties are defined. There are four different possibilities. If both `certainty` and `time` are specified then the service returns a list of queues that can guarantee (statistically at least) that the queue wait time will be lower than `time` with the given `certainty`. If only `certainty` is specified the service will return all the queues and for each queue the minimum time that it can guarantee with that certainty (or the closest higher certainty). If only `time` is specified then the service will return certainties for that time for each queue. If neither is specified it will return full information for all the queues.

We present an example of what a response from the service might look like below. Here neither `time` nor `certainty` was specified.

```
{
  "queues" :
    [
      {
        "name" : "queue_1",
        "certainties" : [
          {"time" : 0, "certainty" : 0.3},
          {"time" : 600, "certainty" : 0.5},
          {"time" : 3600, "certainty" : 0.7},
          {"time" : 21600, "certainty" : 0.8},
          {"time" : 86400, "certainty" : 0.99}
        ]
      },
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 0, "certainty" : 0.6},
          {"time" : 600, "certainty" : 0.8},
          {"time" : 3600, "certainty" : 0.9},
          {"time" : 21600, "certainty" : 0.95},
          {"time" : 86400, "certainty" : 0.99}
        ]
      },
      {
        "name" : "queue_3",
```

```
        "certainties" : [
          {"time" : 0, "certainty" : 0.1},
          {"time" : 600, "certainty" : 0.2},
          {"time" : 3600, "certainty" : 0.8},
          {"time" : 21600, "certainty" : 0.9},
          {"time" : 86400, "certainty" : 0.99}
        ]
      }
    ]
}
```

If `time` was specified and the value was 600, for example, the response will look like the one given below.

```
{
  "queues" :
    [
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 600, "certainty" : 0.8}
        ]
      }
    ]
}
```

If `certainty` was specified as 0.9 and `time` was omitted the response would look like the following.

```
{
  "queues" :
    [
      {
        "name" : "queue_2",
        "certainties" : [
          {"time" : 3600, "certainty" : 0.9}
        ]
      }
    ]
}
```

If `time` was specified as 600 and `certainty` as 0.9, the service would try to find the queue that can guarantee 600 second wait time with 0.9 certainty. Since such a queue does not exist it would return an empty list:

```
{
  "queues" :
    [
    ]
}
```

(b) Method Outline

The method we have chosen to use consists of several consecutive stages. We outline them in the list below. This is not a completely straightforward implementation of the Naive Bayes method.

First of all we have numerical attributes (they are integers, but the range is too broad to treat them as categorical). Therefore either discretization, density function fitting or density function estimation has to be used to get the probabilities required to apply the Bayes formula. We have decided to use discretization since it is not exactly clear what the probability density functions are for these attributes and using kernel density estimation gives poor results in comparison. Since simple binning did not prove satisfactory we employ a more sophisticated method that works by assigning bin boundaries in order to maximize information about the class. The exact method used is described in one of the following subsections. We then use several Naive Bayes classifiers - one for each queue wait time point. For example, if the user specifies 4 points - 10 minutes, 1 hour, 6 hours and 24 hours there will be four different classifiers. The first one to decide whether the job will queue for less than 10 minutes or not, the second one to decide whether the job will queue for less than 1 hour or not and so on. There are practical reasons for doing it this way.

The probabilities produced by Naive Bayes classifiers are known to be inaccurate. This will discussed in the (f) subsection. The probabilities can be calibrated but methods for doing this generally only work for two class problems. Therefore a multi-class problem such as ours needs to be converted into several two class problems. There are several known ways for doing this. The one we have chosen is a simple binning based approach that will be discussed in the (d) subsection.

Our method consists of the following stages:

(i) Collect data and store it. This step is continuously ongoing. Discussed in more detail in subsection (c).

(ii) Build models from that data (this step is triggered on demand, for example, daily):

- Discretize numerical attributes. Discussed in more detail in subsection (d).
- For each time boundary, construct and store the corresponding Naive Bayes model. Discussed in more detail in subsection (e).
- Calculate the information that will be needed to calibrate the probabilities. Discussed in more detail in subsection (f).

(iii) Predict queue wait times for incoming jobs:

- For each time boundary calculate the probability that the job will start executing before that time boundary and the probability that it will start after.
- Calibrate the probabilities for more accurate results.
- Return the probability data to the person or service that requested it.

Let us consider these stages in more technical detail. Most of the methods here are well known in the machine learning circles. However, this exact combination was determined by us to have the best performance for our problem.

(c) Data Collection and Analysis

Queue wait time will likely depend on many factors. Primarily it is the result of the system state at the moment (that is how much resources are taken vs. what resources exist) and the amount of resources requested for the job in question. The more resources the user requests - the more time it will take to allocate them. We tried to decide which specific factors will influence this. These can be grouped as follows:

- The amount of resources requested. Usually the more resources are requested in order to execute a job the more time it will take for the resource manager to secure those resources.
- The state of the system. This will usually boil down to how much resources are currently taken by other jobs. Usually this will be the resources available to a particular queue. In general we should suppose that the more resources are currently in use - the longer it will take for the scheduler to gather the requested resources.

- Historic use of the system for particular users. Generally it can be assumed that the more a particular user used the system in the near past the harder it will be for them to get resources. This, of course, is only true if the resource manager tries to implement some sort of fairness in the resource management process.
- The future states of the system. That is when the currently running jobs finish, how many new jobs are submitted and with what priority, etc. We will not take these factors into consideration for obvious reasons.

We collect data from all the sites participating in the project. Data collection is done by parsing the output of various scheduler tools. The tools and their output will depend on the resource manager (e.g. SLURM, LoadLeveler, PBS) used on that system. Therefore for each scheduler we had to implement a separate data gathering script. All of them work in the following way:

- The system is monitored until a new job is found in the list of jobs in the system. These are retrieved using such commands as llx on LoadLeveler or squeue on SLURM or the equivalent on other systems. The jobs are identified by their ids. The moment a job with an unknown id appears it is added to the in-memory list of jobs and its status is tracked.
- Once the job starts running, the amount of time it has spent in the queue is recorded.
- Once a job is finished, information about it is added to the database.

We use a simple SQL database with one table to track the jobs. In practice this proved a good choice since it allows for efficient preprocessing in the form of SQL queries. The following attributes are stored:

(i) **(1 - 8)** Job attributes

(a) `site` (string) - the name of the site to which the job was submitted
(b) `username` (string) - name of the user
(c) `class` (string) - queue/class/partition name
(d) `when_submitted` (integer) - UNIX timestamp for when the job was submitted
(e) `time_requested` (integer) - how much wallclock time the user has requested
(f) `cpu_time_requested` (integer) - how much cpu time the user has requested
(g) `tasks_requested` (integer) - how many tasks (cpus) the user has requested
(h) `nodes_requested` (integer) - how many nodes the user has requested

(ii) **(9 - 21)** System attributes - describe the state of the whole cluster

(a) `sys_jobs_running` (integer) - how many jobs are currently running
(b) `sys_jobs_queued` (integer) - how many jobs are currently queued for execution
(c) `sys_tasks_running` (integer) - how many tasks are currently running (how many cpus are currently used)
(d) `sys_tasks_queued` (integer) - how many tasks are currently queued
(e) `sys_nodes_running` (integer) - how many nodes are currently running
(f) `sys_nodes_queued` (integer) - how many nodes are currently queued
(g) `sys_time_running` (integer) - the sum of wallclock time requested by the jobs already running
(h) `sys_time_used` (integer) - the sum of time already consumed by running tasks (the sum of time the tasks are already running for)
(i) `sys_time_queued` (integer) - the sum of wallclock time requested by the jobs that are currently waiting for execution
(j) `sys_cpu_time_running` (integer) - the sum of CPU time requested by the jobs that are currently running
(k) `sys_cpu_time_used` (integer) - the sum of CPU time already used up by the running jobs
(l) `sys_cpu_time_queued` (integer) - how much CPU time is currently queued

(iii) **(22 - 34)** Class attributes - describes the state of the queue or partition. These are the same as system state variables but with class prefix instead of sys prefix. The meaning of attributes is analogous. For example `class_tasks_running` means how many tasks are currently running in that queue.

(iv) **(35 - 47)** User attributes - describes the state of the user (information about the user's jobs). Again, identical to sys and class attributes in meaning, but describes jobs currently running, queued, etc. for particular users. The prefix is user instead of sys or class. For example, `user_tasks_running`.

(d) Discretizing Numerical Values

There is an additional complication to using Naive Bayes to our purposes. It is very easy to calculate the required probabilities if the attributes are nominal (consisting of a small set of distinct values). Our attributes are generally numeric. While they are integers the ranges are so large that they cannot in practice be treated as nominal. There are two main ways one can approach this.

(i) Try to estimate the probability distribution. This can be done by either fitting a probability distribution to data (provided we have a reasonable assumption about what that probability distribution is) or using kernel density estimation techniques.

(ii) Discretize numerical attributes, turning them into nominal attributes. The values are then binned into a certain number of bins. Then the numerical attributes are changed to nominal attributes by replacing the attribute values by the bin ids that the attribute falls in.

The second approach (discretization) worked much better in our preliminary tests, therefore that is what we decided to use.

The method we chose for discretizing numerical values is that of Fayaad and Irani [6]. It is a fairly sophisticated method intended to maximize information content about which class the decision vector belongs to when assigning bin boundaries. This is in contrast to simple, unsupervised discretization methods, which proved to perform poorly during the development of our method.

(e) Naive Bayes Classifiers

We have chosen to use a Naive Bayes classifier approach to solve our problem. The Naive Bayes classifier is a simple machine learning technique based around the Bayes theorem. It has been studied extensively since the 1950's. Empirical studies of it were performed by I. Rish et. al [17], D. D. Lewis [11] and many others. D. J. Rennie [16] did a study of the assumptions employed by the Naive Bayes classifiers. There are several reasons for choosing it in our case. First of all, instead of just reporting the class that the classifier predicts an object belongs to, it returns probabilities of an object belonging to each class. This is exactly what we need. The probabilities that the Naive Bayes classifiers reports will not be accurate because of the independence assumption that the classifiers relies on (the assumption that the attributes are mutually independent). However we can correct this using a simple callibration procedure.

The process of predicting queue wait time probabilities boils down to determining what to put in the Bayes formula:

$$p(c|E) = \frac{p(E|c)p(c)}{p(E)} \tag{2.1}$$

Here, the probability $p(c|E)$ is the probability of an object E belonging to class c. Probability $p(E|c)$ is the probability that if an object is of class c it is the object E. Probability $p(c)$ is the probability of any object of belonging to class c. Probability $p(E)$ is the probability of an object being the object E. The value of $p(c)$ is easy to accurately estimate provided there is enough

training data. In our case the supply of training data is not limited in any sense. We can simply count the number of instances of a given class in the training set and divide by the number of samples.

The most complicated part is calculating this conditional probability:

$$p(E|c) = p(x_1, x_2, \ldots, x_n|c) \tag{2.2}$$

Here x_i is the event that attribute i takes on a value x_i. Accurately estimating $p(E|c)$ requires to know the exact dependencies between variables (their joint probability distribution). However, a simpler approach that often works in practice is to just assume that the attributes are independent. In that case to get the probability we just multiply the conditional probabilities of each attribute (given that the object belongs to class c) together. Values $p(x_i|c)$ can be calculated from the training data by counting the relative frequencies of attribute i taking on value x_i.

$$p(E|c) = \prod_{i=1}^{n} p(x_i|c) \tag{2.3}$$

We are left with one part of the equation and that is $p(E)$. Since it does not depend on the class it is merely a scaling factor. Instead of trying to calculating it we can ignore it altogether. Then the output of the formula won't be a probability. However, since we know that the values of $p(t < t_i|E)$ and $p(t \geq t_i|E)$ have to add up to one we can simply scale them afterwards by dividing each of them by their sum. Here, t is the time the job spent queuing and t_i is one of the time boundaries.

Let us now suppose that we want to predict queue wait time using one hour intervals. If we want to limit prediction to, say, 72 hours we will have the following time boundaries: $t_1 = 0, t_2 = 3600, t_3 = 7200, \ldots, t_{72} = 259200$ and the following models: $p(t < t_1|E), p(t < t_2|E), p(t < t_3|E), \ldots, p(t < t_{72}|E)$. Now when given an attribute vector E we will calculate the output of each such model. There will be 72 different outputs representing 72 probabilities.

(f) Calibrating Probabilities

It is well known that the probabilities of Naive Bayes classifiers tend to be skewed. Generally (in practice) they will be squashed either towards one or zero depending whether the decision vector belongs to the given class or not, see [5]. This is not a problem when using them strictly for classification. That is because if we have two decision vectors $\mathbf{x}$ and $\mathbf{y}$ and $s(\mathbf{x}, c)$ and $s(\mathbf{y}, c)$ are the probabilities given by the Naive Bayes Classifier that the vectors belong to class c, then from $s(\mathbf{x}, c) < s(\mathbf{y}, c)$ it follows that $p(c|\mathbf{x}) < p(c|\mathbf{y})$, see [5]. The reason why the probabilities given by the Naive Bayes Classifier are different from actual probabilities is because of the naive assumption employed by the classifier. In general, this assumption is false as there will be correlations between attributes. However, the real values of the probabilities are not important when using it simply as a classifier, as long as the above inequalities and the relationship of logical implication between them holds. This is further investigated and more rigorously shown by P. Domingo and M. Pazzani [5]. We are interested in the actual probabilities as opposed to classifier scores, because it is based on the actual probabilities that the user will want to make the decision. In particular they will rely on what is to them an acceptable probability that a job will start before a specific time.

It is possible to calibrate the probabilities and thus increase their accuracy. Several ways of doing this were proposed in the literature. One of the first simple and practically useful methods was published by John C. Platt [13]. It is meant to transform the outputs of Support Vector Machines (SVMs) into posterior probabilities that a decision vector belongs to a certain class. The method assumes binary classification problems and works by sorting the distances between the support vector and a decision vector in increasing order and fitting a sigmoid function through it. This function is then used to translate SVM outputs to probabilities. Further work in this area was done by B. Zadrozny and C. Elkan [20] [21] who examine the use of probability calibration

for SVMs, decision trees and Bayes classifiers. They also propose a new method based on isotonic regression. For Naive Bayes they propose a method based on simple binning of vectors sorted by their Naive Bayes score. This is the method we chose to use for our problem. It is simple to implement and we have a considerable amount of data which makes binning feasible.

For the benefit of the reader we provide a short explanation of the method here. Note that we are dealing with a two class problem. The classes are: "the job will start executing sooner than t_i amount of time" and "the job will start later than t_i". The value t_i is the i-th time point. If we have decided, for example, to calculate the probabilities with the granularity of one hour then t_1 will be - the job will start within an hour, t_2 will be - the job will start within two hours and so on. For each t_i, we calculate the Naive Bayes probability using the method described in Section (e). This is done for each job in the training set. We then sort all the jobs according to this value. In the end there will as many such sorted sequences of jobs as there are time points.

Then we bin the sorted jobs using equal frequency binning. That is, the number of records in each bin has to be the same. If that is not exactly possible (the number of records does not divide by the number of bins) it has to be as equal as possible. The number of bins will define the coarseness of the adjusted probability estimates. The fewer bins - the fewer discrete values probabilities can take. There are several considerations when choosing the number of bins. More bins means higher accuracy. However it also means there needs to be enough training data to fill those bins. Bin boundary values are then the uncallibrated Naive Bayes probabilities of the first and last job in the bin.

The calibration itself is simple. For each probability to be calibrated we calculate the bin it falls in. The bin boundaries were calculated in advance using the method we described in the previous paragraph. Once we have the bin we calculate the class ratio for that bin. The class ratio is the number of jobs in the training data in that bin that belong to the before class versus the after class. This ratio is the callibrated probability.

3. Results

In this section we will provide some empirical proof of the efficacy of our method. Two experiments were performed, each examining different aspect of the accuracy of the predictions performed by the service. We have chosen one of the queues for which data was collected during the course of the ComPat project [4]. It was chosen on the basis of being heavily used and therefore having a relatively long average queue wait time. Underused systems are unlikely to be interesting in terms of queue wait time predictions. Almost one year's worth of data was used for the experiment. Starting from 2017 August 23 and continuing until 2018 April 26. The data consists of information about 220185 jobs submitted during that time period to the Linux Cluster (CooLMUC-2) at Leibniz Supercomputing Centre of Bavarian Academy of Sciences. The cluster is quite actively used and uses SLURM as its scheduler. The average queue wait time for this system is roughly 15 hours. As such we believe it is a good case to study.

(a) Prediction Accuracy

One of the aspects by which the proposed service can be evaluated is the accuracy of the binary predictions that it makes. As was stated before, for each time division point it returns the probability for the event that the job will start before that point. If that probability is higher than 0.5 we have a reason to believe it will, otherwise we have a reason to believe it wont. Here we measure how often is this prediction correct. We treat it as a binary classification problem at each time point. We have assumed 15 such points and hence 15 problems. For each, we measure the ratio of incorrect guesses to the number of all jobs in the data set and plot the results.

We perform the experiment for prediction accuracy using 10-fold cross-validation [8]. It works as follows:

(i) Divide time using 60×2^i where $i \in [0, 15]$ seconds as division points. This results in the division points of 60, 120, 240, 480 and so on seconds up to 983040 seconds. These time points are the same ones as used by the service in operation. The logic behind choosing exponentially increasing time points is that the service likely becomes less accurate with longer queue times because of factors like newly submitted jobs, jobs finishing, etc. Therefore it does not make sense to use fine grained time points for longer time periods.

(ii) Divide data at random into 10 equally sized parts.

(iii) For each part build a model using the remaining nine parts. Measure prediction accuracy on that part by comparing model predictions to actual data. A job will be considered misclassified if the probability is higher for the wrong class. The two classes being that a job will start before and after the given time point.

(iv) For each of the 15 time points, plot the number of jobs that were misclassified, jobs that were correctly classified, etc.

We can see the results in Figure 1. The y axis is linear and the x axis is logarithmic (each time point is double of the previous time). Job counts are displayed in the y axis and the time points are in the x axis. As can be seen from the plot there is a slight tendency to misclassify jobs as starting before the given time point. The exact reason for this is unclear. The error rate (the ratio between the number of incorrectly classified instances and the number of all instances) is shown in parentheses in the x axis labels. Since this is a queue with a long average queue wait time, a lack of accuracy at lower wait times is not surprising. However, it can be seen that the proposed method is relatively accurate - resulting in below 10% incorrectly classified instances in most cases, going down to around 5% for cases close to the average queue wait time for this data set.

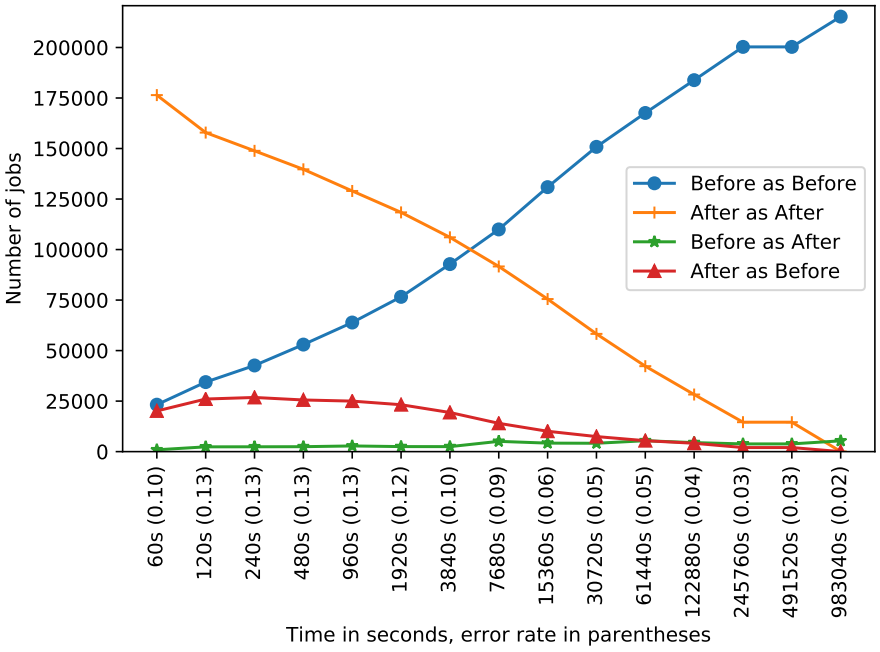

Figure 1. Prediction accuracy plot. It shows the numbers of correctly and incorrectly classified jobs for each time point. The x axis shows the time points in seconds. The y axis shows the number of jobs that were correctly and incorrectly classified. There are four cases - jobs correctly classified as starting before the time point, jobs correctly classified as starting after the time point, jobs incorrectly classified as starting before the time point and jobs incorrectly classified as starting after the time point.

(b) Probability Accuracy

It is important to define what we call the probability that a job will start within t seconds. A simple intuitive definition should satisfy at least the following criterion: if the probability is reported as x, then we expect approximately $x \times 100$ percent of the jobs with that reported probability to start within t seconds as the number of jobs in the testing set increases. For example, if we have 100 jobs that were all given a probability of 0.2 to start within 10 minutes we expect that around 20 of them will start within 10 minutes.

In Figure 2 we see a plot done using the same data as for the previous experiment. In this case we have chosen only one of the time points, namely 7680 seconds or around two hours. The x axis shows the reported probability and the y axis shows the actual proportion in the testing data. As can be seen the calibrated probabilities are close to the ideal which is the line $y = x$. While the uncalibrated probabilities are skewed. The numbers on the plot points signify the number of jobs in that bin.

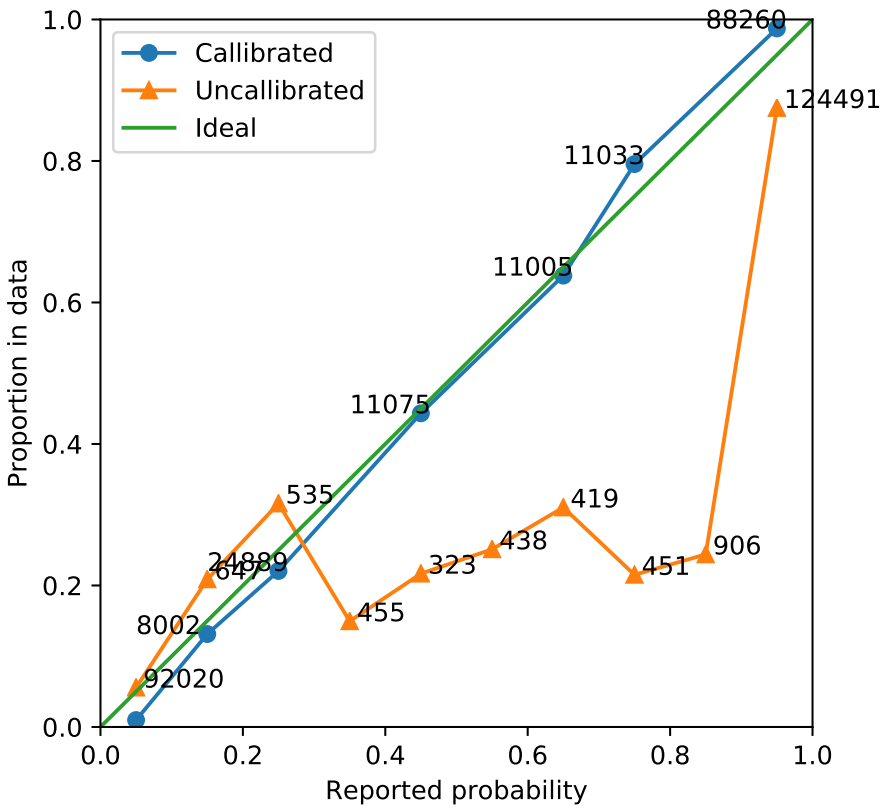

Figure 2. Probability calibration plot for a SLURM based cluster at the time point of 2 hours. Ideally, the callibrated line should fall exactly on the $y = x$ diagonal.

4. Conclusions

In this paper we have described a method for calculating accurate probabilities for discrete queue wait times. The method allows us to give the client a probability of the job starting before

a specified time. We have also developed a service that reports these probabilities for a user specified set of times. Therefore the user can choose the confidence level they are comfortable with when submitting a job. Several computing resources can be then compared in this regard. For example, if it is important to the user that a job starts running as soon as possible they will submit a request for resources at a site that gives a high probability for a low time. If it is not very important the user may accept a riskier option. This is then used as a basis for a brokering mechanism for multi-scale computing problems. Namely queue wait time probabilities are incorporated into the decision of where to submit portions of multi-scale jobs. Parts of multi-scale simulations will have different requirements for optimal execution and queue wait times may be a part of them.

The proposed method is based on the Naive Bayes formula and the probabilities are further calibrated to ensure greater accuracy. The results show that the reported probabilities correspond well to the intuitive notion of a probability for a queue wait time. That is, if we have a certain number of jobs and for each of them the assigned queue wait time probability is, for example, 0.3, we expect 30% of them to start within the given time period. The same is true for other probabilities. Evaluating a large sample of jobs on the LRZ Linux Cluster we provide empirical evidence that this is indeed the case when using the proposed approach.

A web service API is provided that was then used by the brokering system to make automated decisions where to submit parts of multi-scale simulations taking in to account queue wait time probabilities. This API is very general and can be used by brokering systems and for other purposes. The service is being used by the brokering mechanism in the ComPat project. It runs on a Virtual Machine and communicates via the API described in this paper. We hope that with further work it can provide an accurate and useful method to incorporate queue wait times in to the multi-scale brokering decision making.

Funding. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671564 (ComPat project). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 800925 (VECMA project)"

References

1. Saad Alowayyed, Derek Groen, Peter V Coveney, and Alfons G Hoekstra.
   Multiscale computing in the exascale era.
   Journal of Computational Science, 22:15–25, 2017.
2. Pete Beckman, Suman Nadella, Nick Trebon, and Ivan Beschastnikh.
   Spruce: A system for supporting urgent high-performance computing.
   In *Grid-Based Problem Solving Environments*, pages 295–311. Springer, 2007.
3. Joris Borgdorff, Jean-Luc Falcone, Eric Lorenz, Carles Bona-Casas, Bastien Chopard, and Alfons G Hoekstra.
   Foundations of distributed multiscale computing: Formalization, specification, and analysis.
   Journal of Parallel and Distributed Computing, 73(4):465–483, 2013.
4. ComPat consorcium.
   ComPat Project.
   http://www.compat-project.eu/, 2015.
   [Online; accessed 29-June-2018].
5. Pedro Domingos and Michael Pazzani.
   Beyond independence: Conditions for the optimality of the simple bayesian classi er.
   In *Proc. 13th Intl. Conf. Machine Learning*, pages 105–112, 1996.
6. Usama M Fayyad and Keki B Irani.
   On the handling of continuous-valued attributes in decision tree generation.
   Machine learning, 8(1):87–102, 1992.
7. David Jackson, Quinn Snell, and Mark Clement.
   Core algorithms of the maui scheduler.
   In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.
8. Ron Kohavi et al.

A study of cross-validation and bootstrap for accuracy estimation and model selection.
In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

9. Siew Hoon Leong, Anton Frank, and Dieter Kranzlmüller.
Leveraging e-infrastructures for urgent computing.
Procedia Computer Science, 18:2177–2186, 2013.
10. Siew Hoon Leong and Dieter Kranzlmüller.
Towards a general definition of urgent computing.
Procedia Computer Science, 51:2337–2346, 2015.
11. David D Lewis.
Naive (bayes) at forty: The independence assumption in information retrieval.
In *European conference on machine learning*, pages 4–15. Springer, 1998.
12. Hui Li, David Groep, Jeffrey Templon, and Lex Wolters.
Predicting job start times on clusters.
In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 301–308. IEEE, 2004.
13. John Platt et al.
Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods.
Advances in large margin classifiers, 10(3):61–74, 1999.
14. Poznan Supercomputing and Networking Center.
PSNC.
http://www.man.poznan.pl/online/en/.
[Online; accessed 10-July-2018].
15. Poznan Supercomputing and Networking Center.
QCG.
http://www.qoscosgrid.org/trac/qcg.
[Online; accessed 10-July-2018].
16. Jason D Rennie, Lawrence Shih, Jaime Teevan, and David R Karger.
Tackling the poor assumptions of naive bayes text classifiers.
In *Proceedings of the 20th international conference on machine learning (icml-03)*, pages 616–623, 2003.
17. Irina Rish et al.
An empirical study of the naive bayes classifier.
In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.
18. Warren Smith, Valerie Taylor, and Ian Foster.
Using run-time predictions to estimate queue wait times and improve scheduler performance.
In *JSSPP*, pages 202–219. Springer, 1999.
19. Warren Smith and Parkson Wong.
Resource selection using execution and queue wait time predictions.
NASA Ames Research Center TR NAS-02-003, 2002.
20. Bianca Zadrozny and Charles Elkan.
Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers.
In *ICML*, volume 1, pages 609–616, 2001.
21. Bianca Zadrozny and Charles Elkan.
Transforming classifier scores into accurate multiclass probability estimates.
In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 694–699. ACM, 2002.
22. Simon F Portegies Zwart, Stephen LW McMillan, Arjen van Elteren, F Inti Pelupessy, and Nathan de Vries.
Multi-physics simulations using a hierarchical interchangeable software interface.
Computer Physics Communications, 184(3):456–468, 2013.