ProfiT-HPC: Documentation and avoiding of double work

Azat Khuziyakhmetov

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

Workshop zu HPC-Schulung, -Ausbildung und -Dokumentation
(Universität Hamburg, Regionales Rechenzentrum)
30.-31.07.2019

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG)
Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)
KO 3394/14-1, OL 241/3-1, RE 1389/9-1, VO 1262/1-1, YA 191/10-1

Overview

Organization

- ProfiT-HPC: Profiling Toolkit for High Performance Computing
- Duration: 01.02.2017 - 31.01.2020
- Partners:

Goals

Motivation

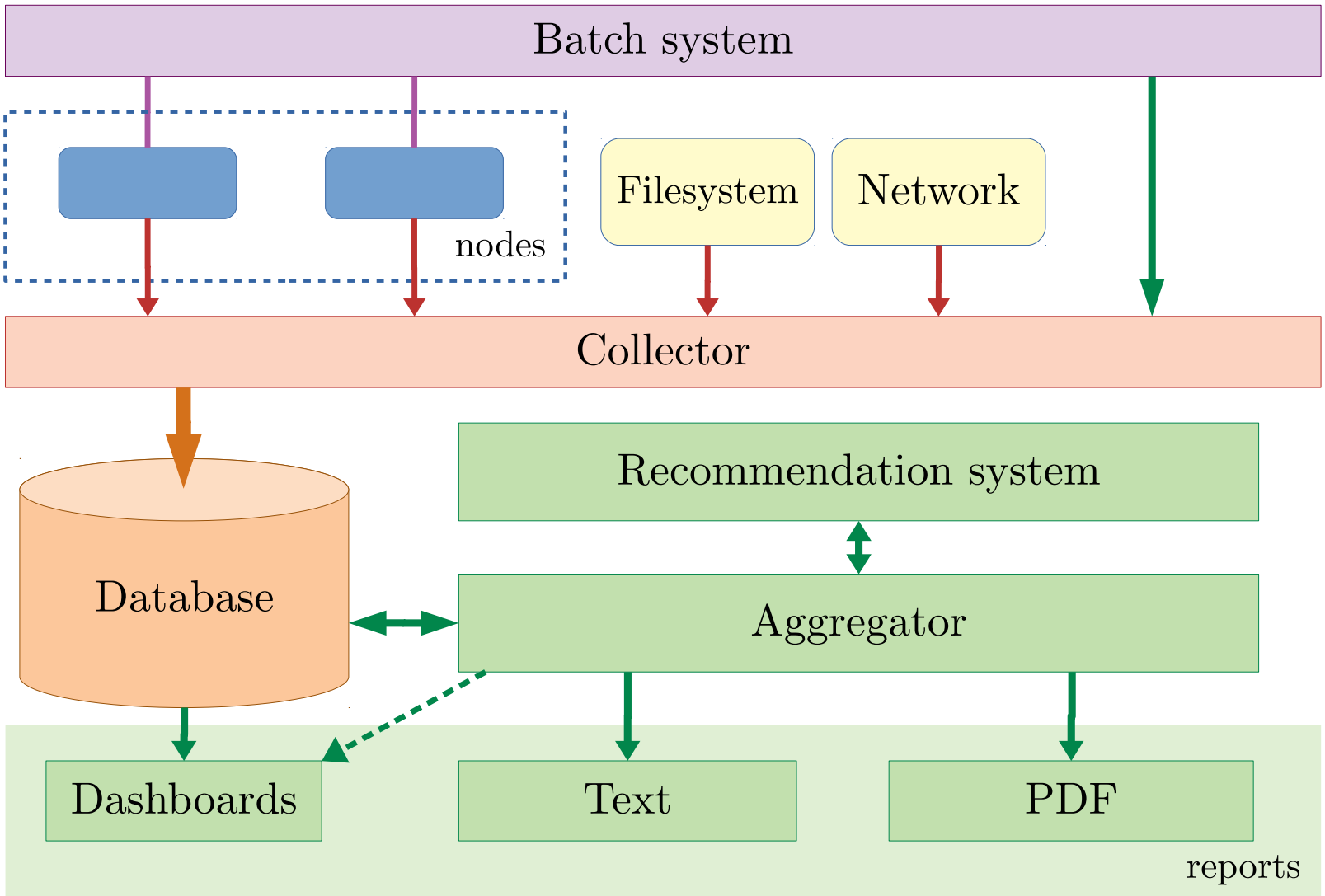
Comprehensive performance analysis of jobs.
Users get reports with necessary information about the job to evaluate its efficiency and tweak submission parameters or programs if needed.

Goals

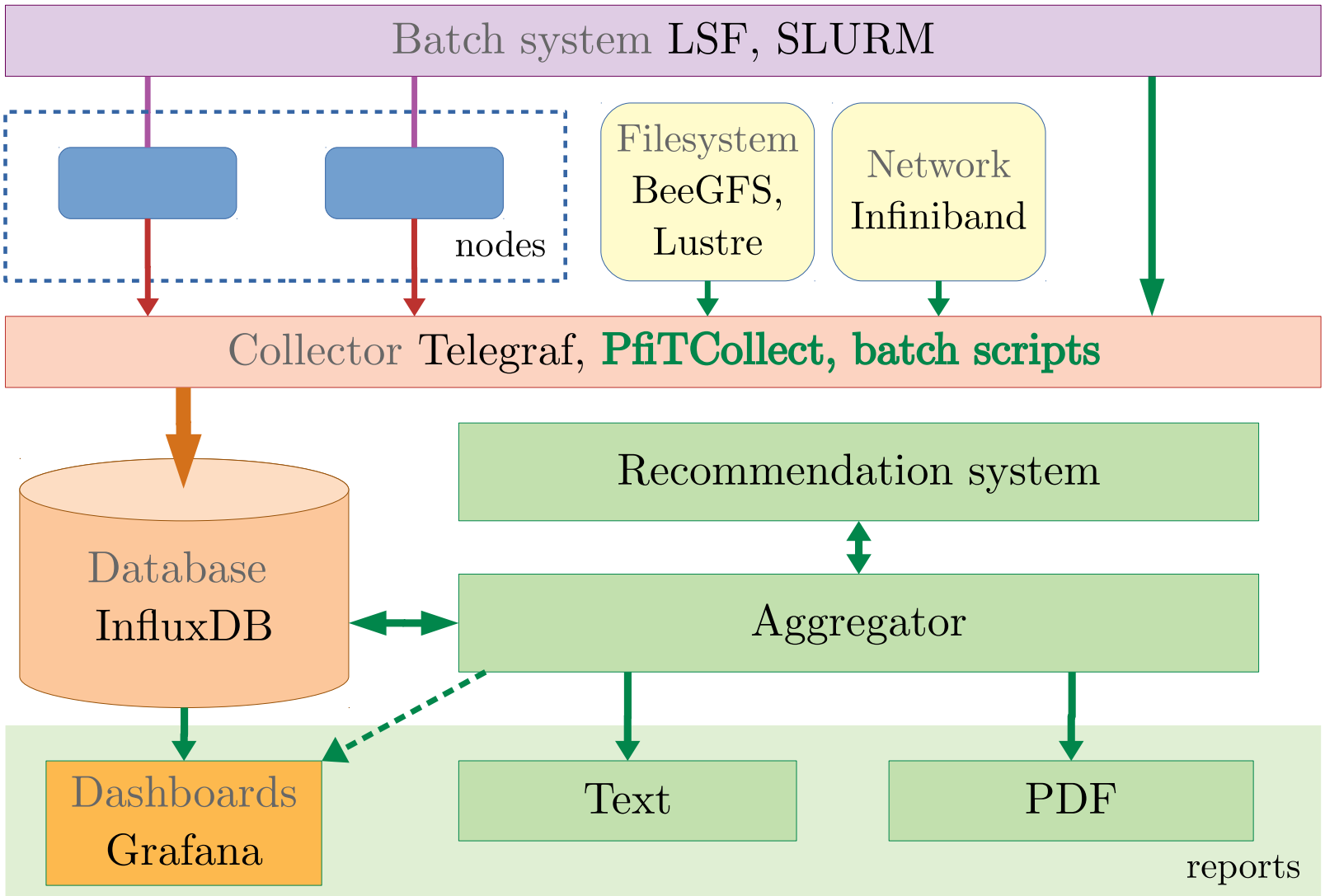
- Auto generated reports with following data:
  - chosen metrics and performance indicators
  - recommendations to reach higher efficiency
- Improved resources utilization of HPC

Architecture

Architecture



Architecture Implementation



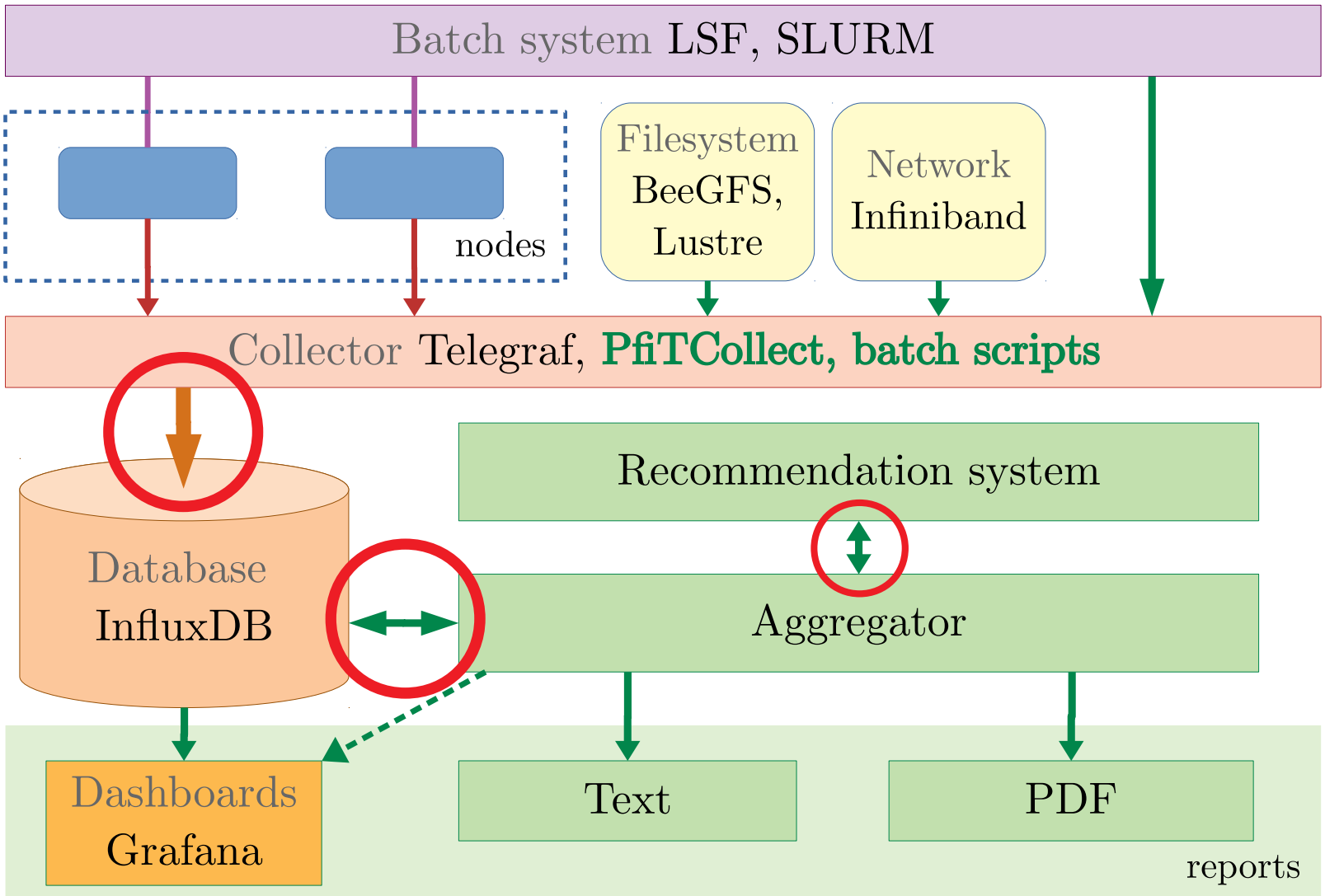
Avoiding of double work

One of the main reasons for modular architecture was avoiding of double work in implementing features of the project.

Many parts of the toolkit should be implemented/documented only once:

- Collecting the measurements
- Reading data from DB (not only InfluxDB)
- Reformatting the values
- Data analysis
- Generating reports/output

Architecture. Avoiding of double work



Challenges in avoiding of double work

Encapsulating everything is not easy. Sometimes double work is necessary.

Imagine developing software for multiple platforms/devices/cpu archs/browsers

Collectors

If you want fast and light collector for measurements, then it should be implemented from scratch.

Reports

Multiple reports are needed for different purposes:
text for output in terminal and plain text mails
pdf for more details and easier comprehension
dashboards for dynamic reports

Documentation

Internal documentation

Generally internal documentation in our project can be divided into 3 groups:

- **input** documentation (what it gets)
- **functional** documentation (what it does)
- **output** documentation (what it returns)

Modules

For every module from the previous slide (architecture), these documentations are essential

Input documentation

Input documentation is very important documentation since some of modules work with 3rd party software which we should support.

It contains:

- **format** of input data (JSON, REST API, parsers)
- **type** of input data (integers, float, string, date, self defined)
- valid **values** (>10Kb, <1 year)

This documentation can be used for validation and sanity checks of the input.

Functional documentation

The *hidden* functionality behind the module, needed to understand what happens under the hood. It also includes so called functional specifications.

This documentation is needed for maintenance purposes and for validating that data manipulation is done correctly and according to the agreed algorithms.

It includes everything related to implementation, such as: functions, algorithms, libraries used in the module and installation docs.

Output documentation

Output documentation is particularly important for report generating modules such as pdf and text reports.

It contains:

- **layout** of the output (headers, body, titles, toc)
- **format** of the output data (graphs, values, text)
- **type** of values (integers, float, string, date, self defined)
- other docs to help navigate for users

This documentation is used for internal modules as well as for modules generating reports. It is also used for making documentation for end users.

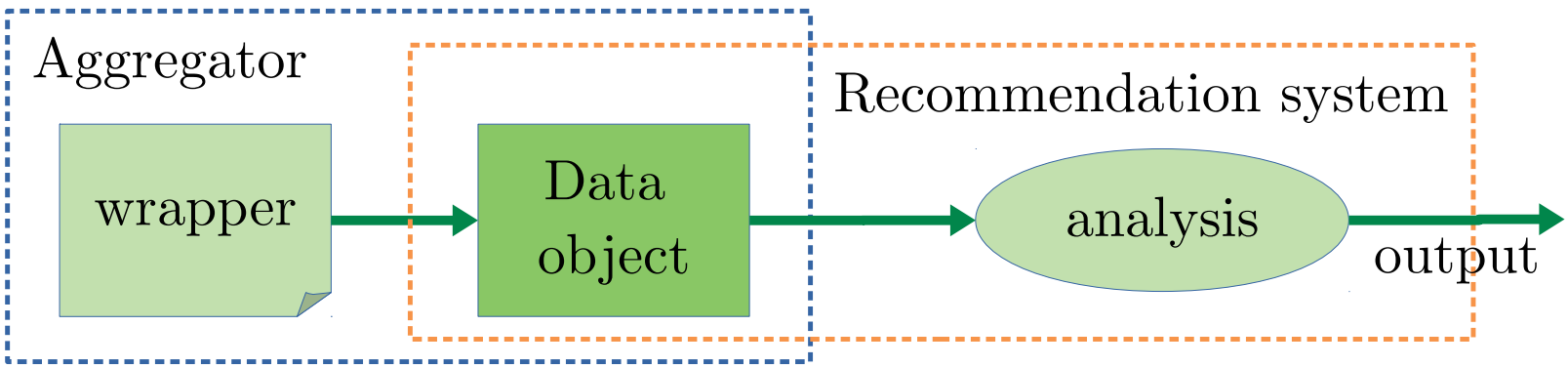
Recommendation system

Recommendation system

Definition

Recommendations are comprehensive instructions for users to help with improving efficiency of their jobs

Recommendation system is designed to be completely separated from other parts of the toolkit. Only data should be provided.



Rule based recommendations

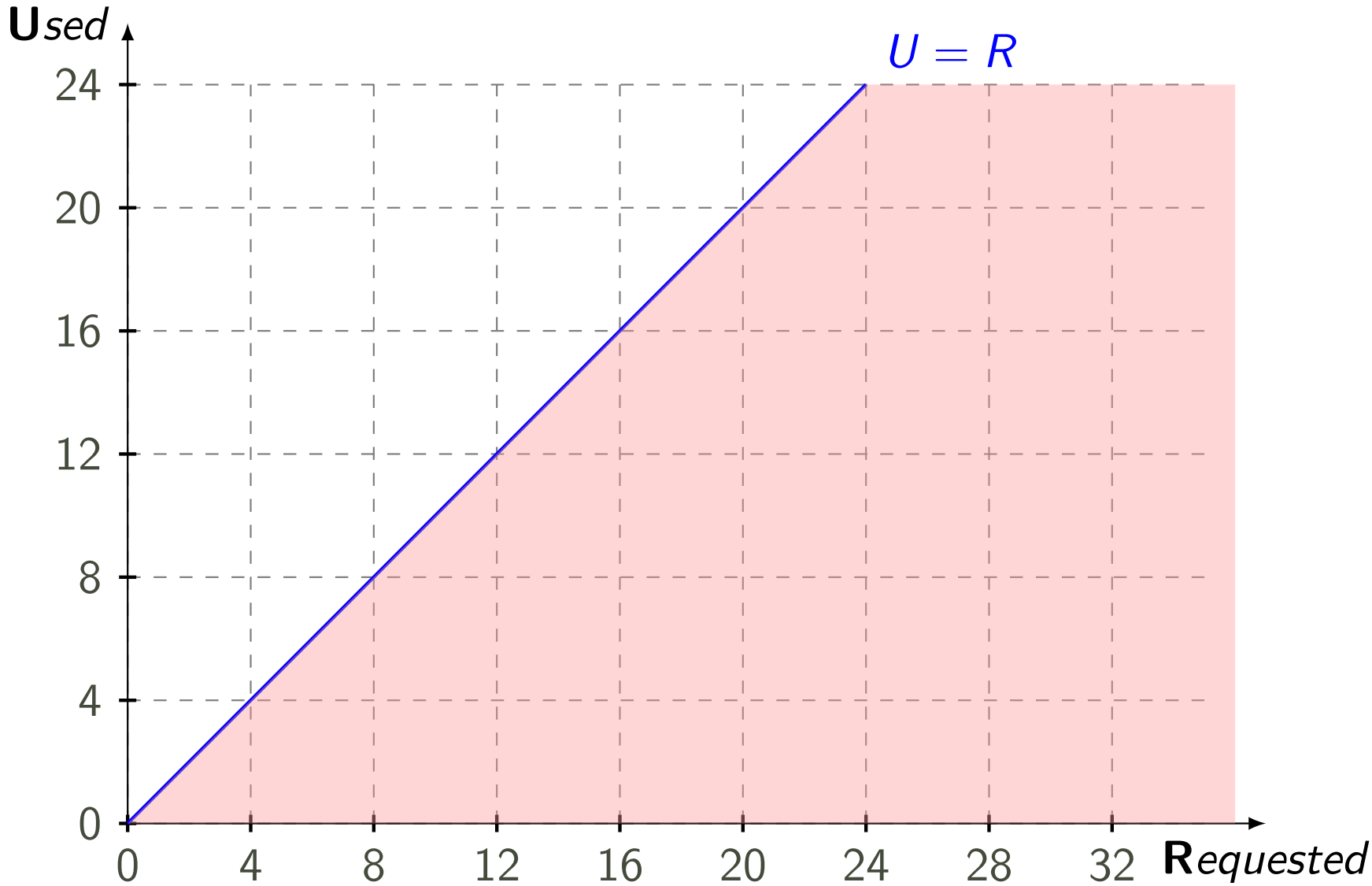
Rules are natural way to give recommendations in current systems by experts

Example

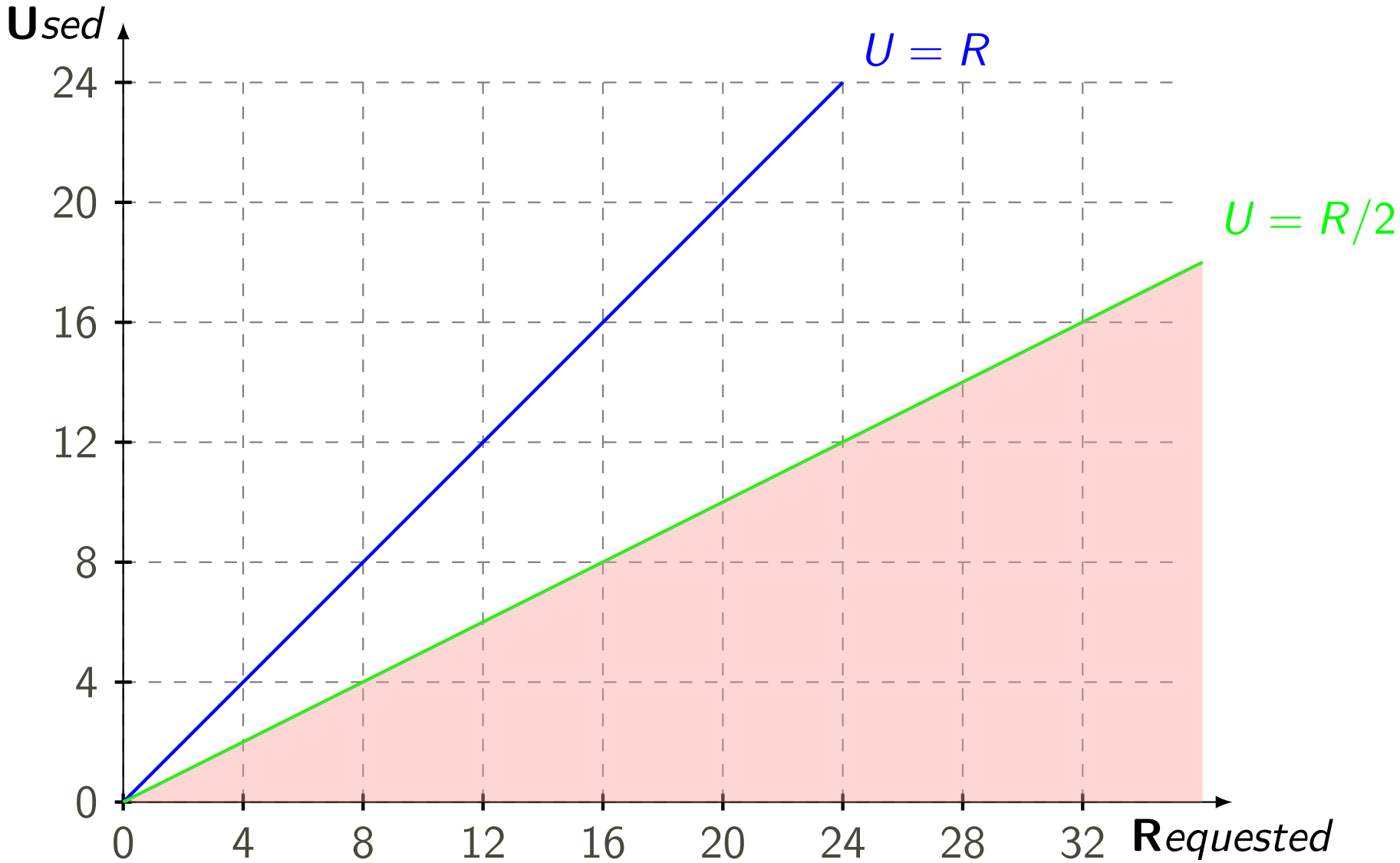
Case	Users request high wall time, resulting to long pending state
Rule	If runtime of the job is half or less than requested walltime ($\texttt{runtime} \leq \texttt{requested}/2$)
Advice	Request less walltime

However, deriving such rules are complicated, even for such simple cases...

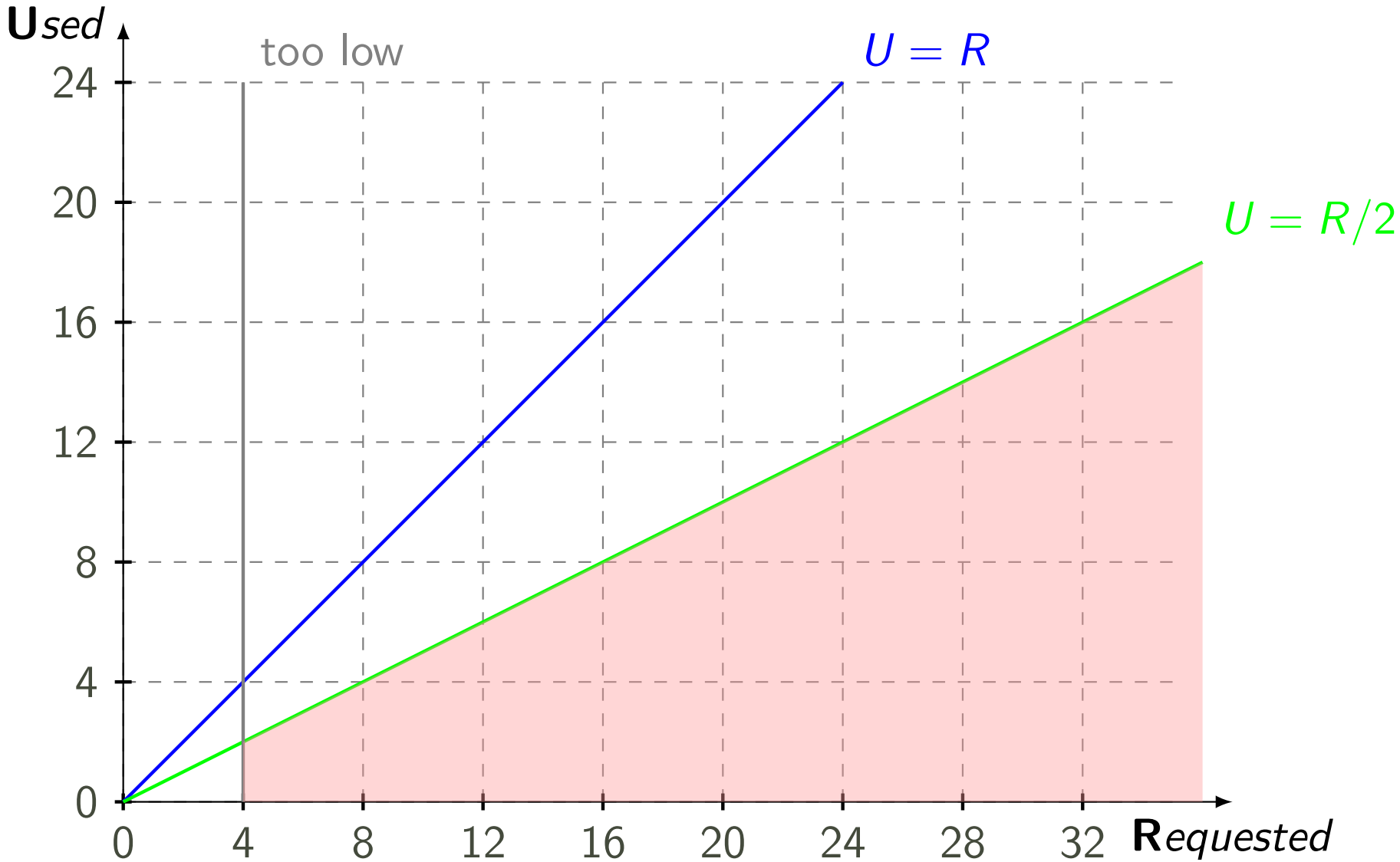
Walltime example



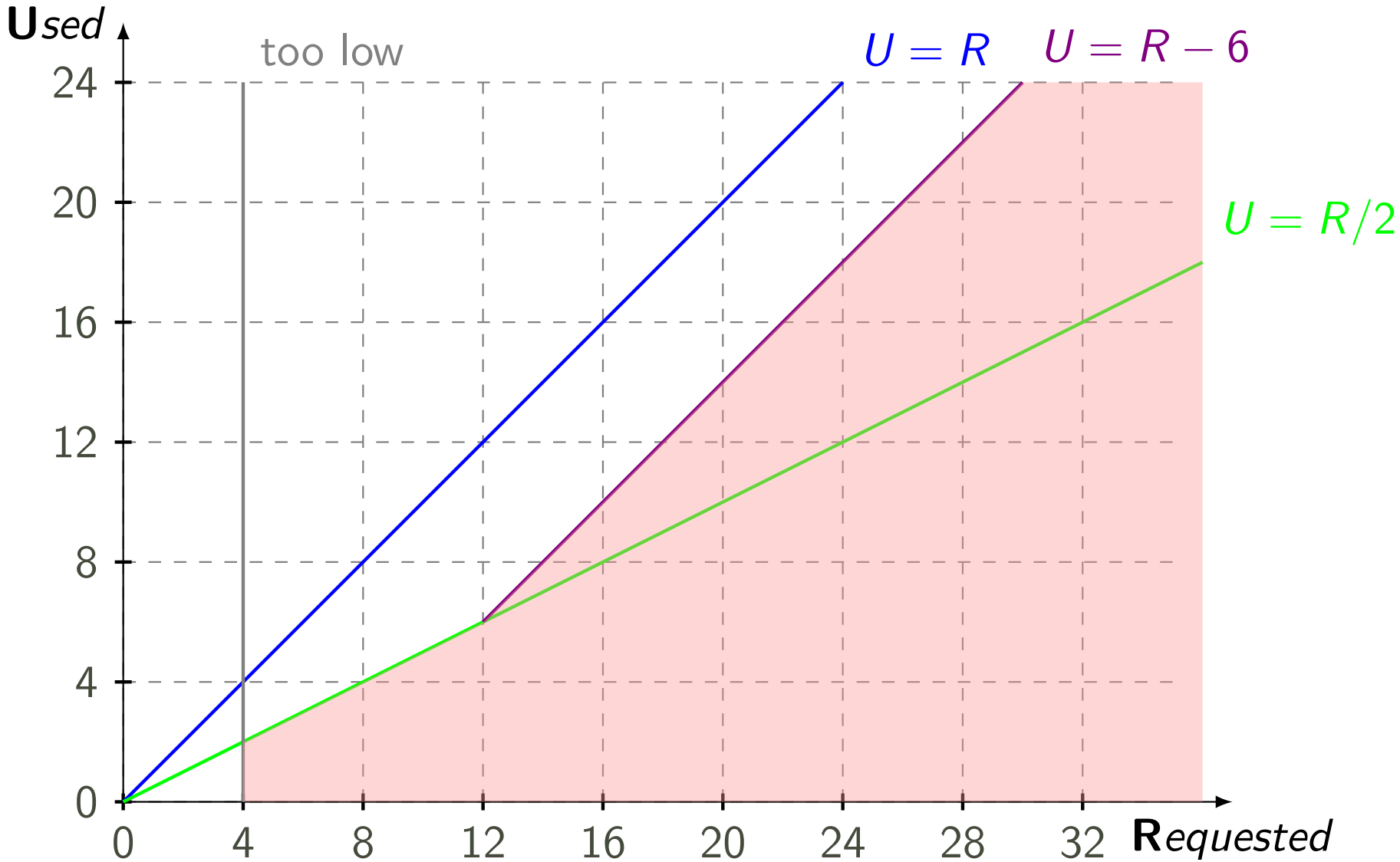
Walltime example



Walltime example



Walltime example



Rules and Attributes

Attribute is a property of the job with predefined values

$$A \in \{V_1, V_2, ..., V_n\}$$

For example, *used_walltime* attribute might be `LOW` or `NORM`

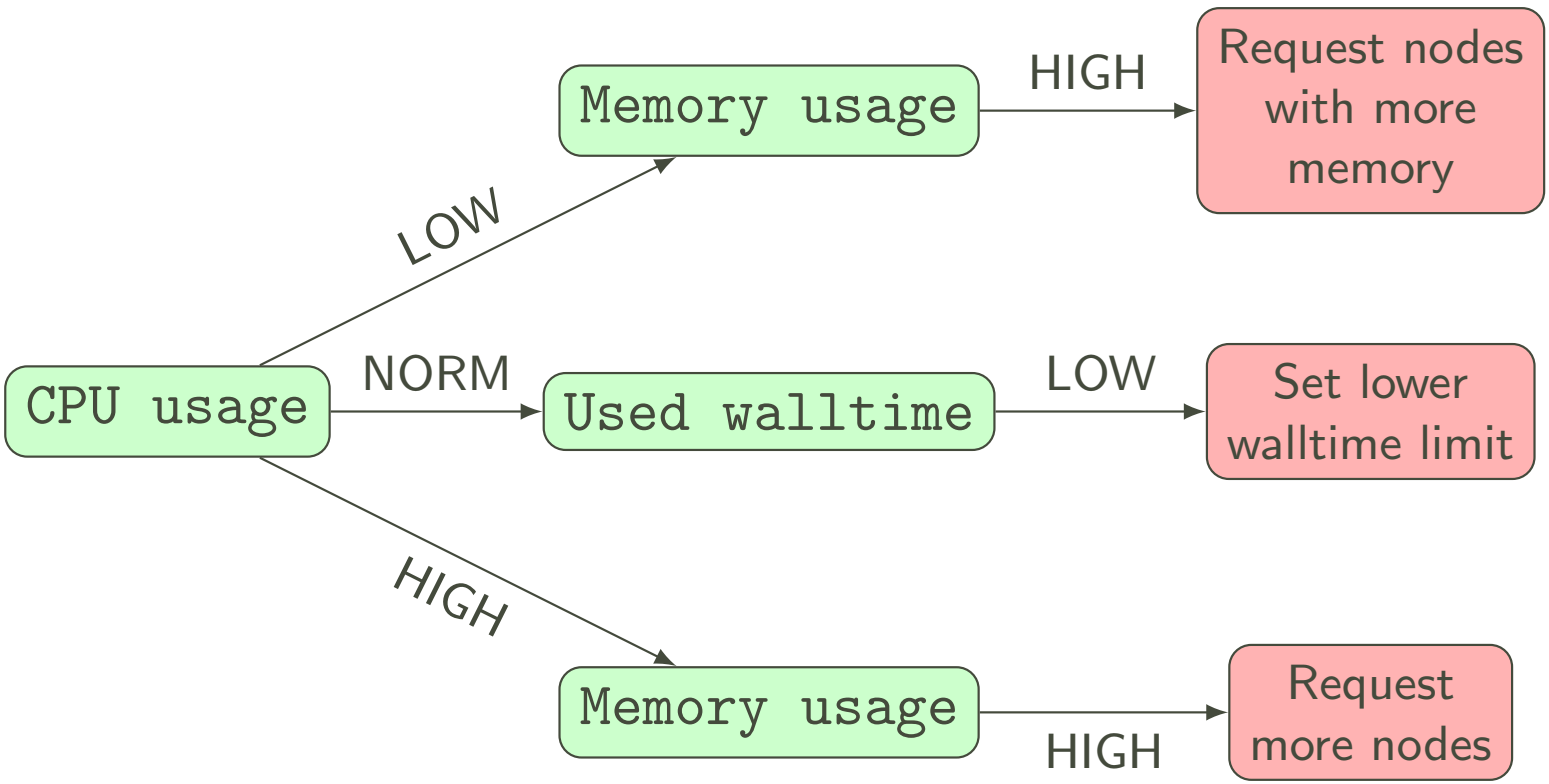
Rule is a set of attributes with specified values

$$R : A_1 = V_1 \ \& \ A_2 = V_2 \ \& \ ... \ \& \ A_k = V_k$$

For example, *fix_requested_walltime* rule might be formed by attributes like *used_walltime* = `LOW` & *cpu_usage* = `NORM`

Decision trees

The set of the rules can be represented and visualized as a decision tree

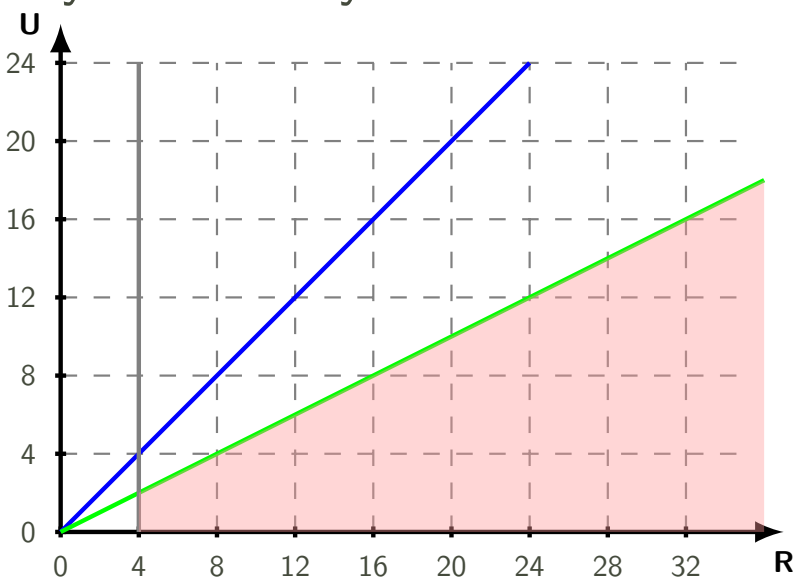


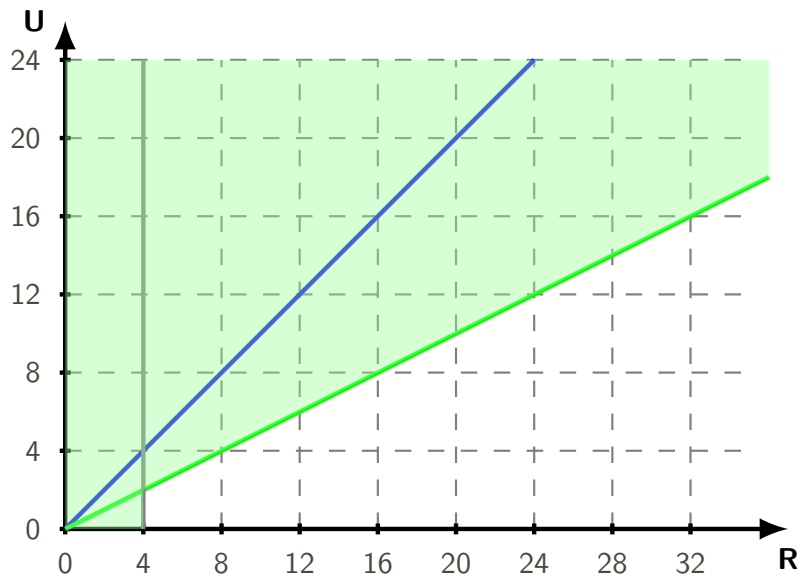
User documentation from rules

With strictly defined rules and attributes, they can be reversed to make an advanced user documentation to run jobs efficiently.

Disadvantage of transforming of rules to docs is that documentation will be complicated to follow, since terms will be very technical.

Recommendation system should provide easier documentation for users only when they need it.





Avoiding double work among projects

It would be great to avoid double work not only within a single project, but also among projects generally.

Key concepts to avoid double work among projects:

- Use version control for code and docs (GIT)
- Make use of well supported languages and tools (Python, C, Grafana)
- Share documentation and *code basis* (not important)
- Anonymize data and share for dev and tests

For smaller projects **sharing the code** basis is usually not important, especially for tools with narrow usage.

The documentation on the other hand might help developers and users to avoid making the same mistakes and improve code/usage quality.

Summary

Key take-away points:

Architecture

- Modular architecture to avoid double work in code
- Complete internal documentation for clarity
- Full specification of the code

Documentation

- Sharing documentation among projects/users
- Generating docs from specs or automatic tools
- Full specification of the code

Thank you!

Questions?

https://profit-hpc.de info@profit-hpc.de