MASARYK UNIVERSITY

FACULTY OF INFORMATICS

PBSMon2, web service for displaying MetaCenter status

Master's Thesis

BC. MARCEL LUKČO

Brno, Fall 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

PBSMon2, web service for displaying MetaCenter status

Master's Thesis

BC. MARCEL LUKČO

Advisor: Mgr. Miroslav Ruda

Department of design and development of software systems

Brno, Fall 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Marcel Lukčo

Advisor: Mgr. Miroslav Ruda

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Mgr. Miroslav Ruda, for his valuable guidance, insightful comments, and continuous support throughout the development of this thesis.

I would also like to thank Mgr. Václav Chlumský, for his technical assistance and expertise, which significantly contributed to the practical part of this work.

My thanks also go to Ing. František Řezníček, and Mgr. Ivana Křenková for their collaboration, helpful advice, and support during the integration with related systems.

Abstract

TBD

Keywords

cloud, cloud computing, distributed computing, MetaCentrum

Contents

List of Tables

List of Figures

Introduction

Currently, high-performance computing plays a key role in the implementation of research projects across a wide range of scientific disciplines. In the academic sector, it is therefore essential to ensure access to sufficient computing power to enable the implementation of demanding simulations, the processing of large data sets, and the effective analysis of results. National computing infrastructures, including CESNET, have been created for this purpose. This academic institution offers computing resources dedicated exclusively to research and academic purposes.

The PBSmon web service was developed to monitor these resources and provide an overview of their status. This application ensures the regular collection of metadata from computing nodes, queues, servers, and running jobs using native calls to PBS servers. It then transforms this data into a visual form accessible to users. Thanks to PBSmon, users can monitor the current system load, the status of individual jobs, and the availability of computing resources in real time.

However, the original PBSmon application is monolithic and uses outdated libraries, such as Java library Stripes, which is no longer maintained. Growing demands for security, sustainability, and support for containerized deployment, and the need to cover both PBS and OpenStack require a fundamental refactoring and extension of the system.

This thesis deals with the design and implementation of a new version of the PBSmon system, built on modern technologies: backend within the NestJS framework and frontend within the React framework.

The goal is to create a modular, maintainable, and cloud-native architecture that not only preserves the functionality of the original solution, but also extends it to support the OpenStack cloud and to be ready for further development and adaptation to current technological standards, including new features required by the assignment.

1 Analysis of the Existing Solution – PBSmon

The PBSmon application serves as a central tool for collecting and visualizing data from various systems related to computing resource management. It automatically collects information from the PBS (Portable Batch System), the Perun identity management system, and virtual machines (OpenStack). It then processes this data and provides users with a comprehensive view of the current status of the infrastructure, jobs, and user activities within the entire MetaCentrum infrastructure.

The web interface is used to visualize data from the PBS, Perun, and OpenStack systems. The information displayed includes a personal view of jobs, available computing resources, an overview of computing machines and their utilization, a list of users, and other statistics. Among other things, the following data is collected:

- PBS nodes (status, load, available resources),
- information about virtual machines
- physical machines (availability, utilization, technical parameters),
- users (identity, activity, jobs),
- queues (status, priority, configuration),
- jobs (status, start time, resources used, user).

1.1 System Architecture

The existing solution is built as a monolithic Java application that collects data from various systems (PBS, Perun, OpenStack) and displays it in a web interface. The architecture is tightly coupled and uses multiple technologies, including C libraries and shell scripts.

1.1.1 Data collection environments

PBSmon collects following data from the following environments:

- PBS environment
  - Compute clusters managed by PBS systems
  - Collection of Fairshare metrics from pbscache, which determine the user's priority when launching a job
  - Collection of the /etc/group file from PBS, essential for assigning users to groups used in access control lists (ACLs)
- Perun environment - an identity and group management system. Returns information about users and the list of physical machines.
- OpenStack environment - Information about virtual machines running on physical hosts

Each of these environments has its own method of communication and data representation. Within the system architecture, these differences are abstracted through interfaces and adapters that ensure unified data processing. All of these enviroments and their data structure are described in the following sections.

1.1.2 Data Processing and Unification

The data obtained from various sources — such as compute clusters (PBS), cloud platforms (e.g., OpenStack), and the identity management system (Perun) — differ in their formats and representations. After collection, the data are first stored in an in-memory cache and subsequently mapped and unified to establish logical relationships between entities from different systems. Among other things, the following relationships are established:

- Hierarchical structure: user(Perun) → jobs (PBS) → queues (PBS)
- Mapping: virtual machines (OpenStack) and physical machines (Perun)

1.1.3 Web Presentation

The presentation of data is performed through server-side rendering of HTML pages. Only logged in users, that have approved access to PBSmon can access the web interface. The authorization is performed by third party system using OICD token, that will be described in detail in section 1.6.

These pages display detailed views of jobs, nodes, system states, and other information relevant to both users and administrators. As mentioned earlier, the web interface is strictly read-only and does not provide any functionality for data input or modification. The frontend is not implemented as a standalone application; it is an integral part of the monolithic Java system.

One of the key features is so called **QSUB assembler**, which allows user to add required parameters for the job submission and PBSmon will notify the user about the nodes that fullfils the requirements. Among other, it produces a shell script that can be executed to submit the job to the PBS server with the required parameters

All of this will be described in more detail in section 1.7.

1.2 Data Collection from PBS Environment

The Portable Batch System (PBS) is a distributed workload management system designed to schedule and monitor computational jobs across multiple compute nodes in a cluster environment. [1]

In a typical configuration, the PBS environment consists of a central management server and a set of compute nodes, which are individual physical or virtual hosts providing computational resources such as CPUs, memory, GPUs, and local storage. [1]

A compute node (also referred to as a host) represents the fundamental execution unit in the cluster — it is the machine where user jobs are actually executed. Each node communicates with the central PBS server, which manages job submission, scheduling, and resource allocation. [1]

1.2.1 Entities in PBS Environment

PBS provides structured data that enables continuous monitoring of the computing environment operation, scheduler behavior, and resource utilization. From the PBS perspective, the following main entities are collected: [1]

Server

The PBS server is the authority for the entire cluster: it receives and registers jobs, maintains queues and nodes, tracks their states, and publishes global statistics (e.g., counts of jobs by state). It provides the following data: [1]

- **Policies and limits** – defines and enforces scheduling and resource utilization rules (limits on job and CPU counts, job array rules, rescheduling on node failure, scheduling enable/disable).
- **Resources and default settings** – manages available/assigned resources and defaults (e.g., resources_available, resources_default, default_chunk, default_queue) and provides summary utilization (assigned memory/CPU/nodes).
- **Integration with scheduler and reservations** – provides parameters for the scheduler (iteration, backfill, sorting/fairshare

formula) and supports time-based resource reservations (advance/standing/maintenance).

- **Security and access** – manages access policies through ACLs (hosts, users, managers/operators), supports Kerberos/realm policies, and handles credential management/renewal.
- **Operations and management** – configures logging, email notifications, license quotas/counters, and system version; allows management of hooks and other server-level objects.[1]

Jobs

A job is the basic computational unit submitted by a user to the PBS server—it contains command(s) to execute and resource requirements (CPU, memory, time, GPU). The server queues it, schedules it to nodes, monitors its progress, and upon completion evaluates its result (including outputs and return code).[1]

A job can be a standalone task or a member of a job array (with multiple subjobs) sharing the same resource template. PBS provides the following data about jobs:[1]

- **Identity and ownership** – job ID and name, owner (Job_Owner), project/VO (project), target queue (queue), server, and submit host.
- **Lifecycle and scheduling** – state and substate (job_state, substate), priority, run count, holds (Hold_Types), rerunnability (Rerunable), credentials and validity (credential_id, credential_validity).
- **Requested resources and placement** – Resource_List.* (e.g., select, ncpus, mem, walltime, place, scratch_*, mpiprocs, ompthreads, nodect) and runtime identifiers (session_id).
- **Timing metrics** – ctime, qtime, stime, mtime, obittime, etime + derived indicators (e.g., eligible_time).
- **I/O and environment** – working directory (jobdir), stdout/stderr paths (Output_Path, Error_Path), submission arguments (Submit_arguments), and environment variables (Variable_List).

- **Result and diagnostics** – return code (Exit_status) and auxiliary fields for auditing and progress tracking.[1]

Queues

A queue is a logical structure for accepting and processing jobs—it defines its type (Execution vs. Route), resource defaults and limits, access rules, and how jobs are either executed on nodes or redirected to target queues. PBS provides the following data about queues: [1]

- **Identity and type** – queue name, queue_type (Execution/Route), Priority (weight in scheduling), operational state (enabled/started), and optionally hasnodes.
- **State and utilization** – aggregates such as total_jobs and state_count, and currently assigned resources resources_assigned.* (e.g., memory, CPUs, nodes, MPI processes).
- **Policies and limits** – resource boundaries resources_max.* and minimums resources_min.* (including GPU and walltime), extra rules like kill_delay, backfill_depth, or from_route_only (accepts jobs only via routing).
- **Defaults and placement** – resources_default. (e.g., CPUs, walltime, placement, GPUs) and default_chunk.* (e.g., implicit chunk size, queue_list for targeting).
- **Routing (Route queues)** – route_destinations defines target execution queues to which jobs are automatically redirected.
- **Access and security** – ACL toggles and lists: acl_user_enable/acl_users, acl_group_enable/acl_groups, or acl_host_enable/acl_hosts.
- **Organizational tags** – optional attributes such as fairshare_tree (fairshare hierarchy) or partition (infrastructure label/partition) for logical segmentation and policy purposes. [1]

Nodes

A node represents a physical machine in the PBS environment that provides computational resources for job execution. Each node is managed by the PBS server and can be in various operational states depending on its availability and current workload. PBS provides the following data about nodes: [1]

- **Identity and location** – node name (vnode), hostname (host), cluster identifier (resources_available.cluster), and the name of the execution daemon (Mom) that manages the node.
- **State and availability** – primary state (state) indicating the operational status (e.g., free, job-busy, down, offline) and auxiliary state (state_aux) providing additional context.
- **Available resources** – total capacity of the node defined by `resources_available.*` attributes, including:
  - CPU resources: ncpus (number of CPUs), pcpus (physical CPUs), cpu_vendor, cpu_flag (CPU feature flags)
  - Memory: mem (total memory), vmem (virtual memory), hpmem (high-performance memory)
  - Accelerators: ngpus (number of GPUs), gpu_mem, gpu_cap (compute capability), cuda_version
  - Scratch storage: scratch_local, scratch_shared, scratch_ssd, scratch_shm (shared memory)
  - Network: ethernet_speed, infiniband
  - Software: singularity (container support), os, osfamily, arch
  - Organizational: queue_list (queues accessible from this node), cluster-specific tags
- **Assigned resources** – currently allocated resources tracked by `resources_assigned.*` attributes, including:
  - ncpus, ngpus, naccelerators (number of assigned CPUs, GPUs, accelerators)

  - mem, vmem, hbmem, accelerator_memory (assigned memory)
  - scratch_local, scratch_ssd (assigned storage)

- **Active jobs** – list of job identifiers (jobs) currently running on the node, with each job identified by its ID and task index (e.g., 14964063.pbs-m1.metacentrum.cz/0).
- **Sharing and placement** – sharing mode (`sharing`) that determines resource allocation (e.g., `default_shared`, `force_exclusive`), and reservation support (`resv_enable`).
- **Timestamps** – last_state_change_time (timestamp of the last state transition) and last_used_time (timestamp when the node was last utilized). [1]

The distinction between available and assigned resources enables monitoring of node utilization, while the state information provides insight into node availability for job scheduling. The jobs list allows tracking which specific jobs are consuming resources on each node, which is essential for resource accounting and troubleshooting.

Reservations

Reservations are a mechanism in PBS that allows nodes to be reserved for exclusive use during a specific time period. When a reservation is created, the specified nodes become unavailable for regular job scheduling. For each reservation, PBS automatically creates a dedicated queue that provides access to the reserved resources. [1]

PBS provides the following data about reservations:

- **Identity** – reservation name (Reserve_Name), unique identifier (name), and the associated queue name (queue) that is created for the reservation.
- **Ownership and access** – reservation owner (Reserve_Owner) and list of authorized users (Authorized_Users).
- **State information** – reservation state (reserve_state) and substate (reserve_substate) indicating the current status of the reservation (e.g., active, confirmed, or in transition).

- **Time constraints** – reservation start time (reserve_start), end time (reserve_end), and duration (reserve_duration) specified as Unix timestamps.
- **Resource requirements** – the resources reserved by the reservation, including:
  - Memory: Resource_List.mem (total memory reserved)
  - CPUs: Resource_List.ncpus (number of CPUs)
  - GPUs: Resource_List.ngpus (number of GPUs, if applicable)
  - Nodes: Resource_List.nodect (number of nodes) and Resource_List.select (detailed node selection specification)
  - Placement: Resource_List.place (placement policy, e.g., `free`, `exclhost`)
  - Walltime: Resource_List.walltime (maximum execution time for jobs in the reservation)
- **Reserved nodes** – list of actual nodes allocated to the reservation (resv_nodes) with their specific resource allocations.
- **Metadata** – creation time (ctime), modification time (mtime), submission host (Submit_Host), and partition information (partition, if applicable).
- **Retry information** – reservation count (reserve_count) and retry attempts (reserve_retry) for tracking reservation lifecycle. [1]

Resources

Resources in PBS represent units of computational capacity that can be requested by jobs, allocated to nodes, and managed by queues and the server. Each resource is defined with a unique name and attributes that specify its data type and where it can be used within the PBS system. [1]

PBS provides a comprehensive list of all available resources in the system, where each resource definition includes: [1]

- **Resource name** – the identifier used to reference the resource in job submissions, node configurations, and queue settings (e.g., `cput`, `mem`, `ncpus`, `ngpus`, `walltime`).
- **Type** – the data type of the resource value, which determines how the resource is interpreted and validated. Common types include:
  - `long` – integer values (e.g., CPU count, GPU count)
  - `size` – memory or storage values with units (e.g., bytes, KB, MB, GB)
  - `string` – text values
  - `float` – floating-point numeric values
  - `time` – time duration values
- **Flag** – a string of characters indicating where the resource can be used or referenced:
  - `h` – can be used at the host/node level
  - `q` – can be used at the queue level
  - `n` – can be used at the node level
  - `m` – can be used at the server level

  [1]

Scheduler status

The PBS scheduler is responsible for making decisions about which jobs to run, when to run them, and on which nodes to execute them. A PBS system can have multiple schedulers, each managing a specific partition or set of resources. The scheduler continuously evaluates queued jobs against available resources and applies scheduling policies to optimize resource utilization and meet job requirements. [1]

PBS provides status information about each scheduler instance, including: [1]

- **Identity and location** – scheduler name, the host where the scheduler daemon runs (sched_host), and the partition it manages (partition).

- **Operational state** – current state of the scheduler (e.g., `scheduling`, `idle`) and whether scheduling is enabled (scheduling).
- **Scheduling cycle** – scheduler cycle length (sched_cycle_length) defining how frequently the scheduler evaluates and schedules jobs, and the current iteration count (scheduler_iteration).
- **Processor set configuration** – settings for processor set (pset) handling: do_not_span_psets (prevents jobs from spanning multiple processor sets) and only_explicit_psets (restricts scheduling to explicitly defined processor sets).
- **Scheduling mode** – throughput_mode indicates whether the scheduler prioritizes throughput optimization, and opt_backfill_fuzzy (if present) specifies the backfill optimization level.
- **Preemption settings** – configuration for job preemption: preempt_queue_prio (priority threshold for preemption), preempt_prio (queues or job types that can be preempted), preempt_order (preemption order strategy), and preempt_sort (sorting method for preemption selection, e.g., `min_time_since_start`).
- **Integration and hooks** – job_run_wait specifies the hook that controls when jobs can start execution (e.g., `runjob_hook`).
- **File system paths** – sched_priv (path to scheduler private directory) and sched_log (path to scheduler log files).
- **Logging and monitoring** – log_events (bitmask specifying which events to log) and server_dyn_res_alarm (alarm threshold for dynamic resource changes). [1]

Fairshare metrics

A list of users, their fairshare values, and the timestamp when the record was last modified. fairshare values are used per scheduler.[1] This data is retrieved for both the QSUB assembler and user monitoring purposes.

1.2.2 Data Collection Mechanism

Batch Interface Library (IFL)

PBS provides a C library, which represents the programming interface (API) of the PBS system, also known as the Batch Interface Library (IFL). [1]

This library allows external applications and tools to communicate with the PBS server. It provides functions for remote management of batch jobs, querying the system state, and managing computational resources through TCP/IP communication. Using the library, it is possible to implement a client application that: [1]

- establishes a connection to the server (pbs_connect),
- authenticates the user,
- creates and submits jobs (pbs_submit),
- queries their status (pbs_statjob, pbs_selstat),
- modifies or deletes jobs (pbs_alterjob, pbs_deljob),
- works with information about the server, queues, nodes, or scheduler.[1]

Thus, library represents a key component for implementing a tool that enables data collection and monitoring of jobs managed by the PBS server.

Subsequently, current PBSmon implementation contains C code that, when invoked, retrieves information from the PBS server using the library functions and then stores this data into a file, which the Java application later processes and saves into the in-memory store (see Listing 1.1).

This collection process is triggered whenever the user wants to display any PBSmon page and when the data in the memory cache is older than 60 seconds.

Another important note is that collected data represents the current state of the PBS server. It does not include any historical data.

Listing 1.1: Partial code snippet for data collection from PBS server

```
#include <pbs_error.h>
#include <pbs_ifl.h>

int main(int argc, char **argv) {
    // ...
    con = pbs_connect(server);
    if(con<0) {
        return 1;
    }
    /* get server info */
    bs = pbs_statserver(con, NULL, NULL);
    process_data(bs,"servers");
    /* get queues info */
    bs = pbs_statque(con, "", NULL, NULL);
    process_data(bs,"queues");
    /* get nodes info */
    bs = pbs_statnode(con, "", NULL, NULL);
    process_data(bs,"nodes");
    /* get jobs info: t - job arrays, x -
       finished jobs*/
    bs = pbs_statjob(con, "", NULL, "tx");
    process_data(bs,"jobs");
    /* get reservations info */
    bs = pbs_statresv(con, NULL, NULL, NULL);
    process_data(bs,"reservations");
    /* get resources info */
    bs = pbs_statrsc(con, NULL, NULL, NULL);
    process_data(bs,"resources");
    /* get scheduler info */
    bs = pbs_statsched(con, NULL, NULL);
    process_data(bs,"schedulers");
    /* end connection */
    pbs_disconnect(con);
    return 0;
}
```

Fairshare metrics collection

Fairshare metrics are collected using the bash command.

Listing 1.2: Metrics collection script

```
list_cache <pbsServer> fairshare{.elixir}
```

This shell command returns a CSV file with columns:
user, last_modified, fairshare.

Group file collection

In addition to entities and caches, the shell script located on PBS server automatically pushes its /etc/group file. These files provide information about local UNIX groups and their members. Whenever time it has changed, the PBS server writes the updated version to the PBSmon server's filesystem under the following path: **/etc/pbsmon/-group/<pbsServer>**.

1.3 Data Collection from Perun

Perun is an open-source system developed in Java that serves for comprehensive management of identities, groups, attributes, and access to various resources and services. It is a modular solution designed for efficient management of users, organizations, and projects. The system is built with an emphasis on operation in distributed environments and on integration with existing identity systems in the fields of research and education.[2]

1.3.1 Entities in Perun

Entities retrieved from Perun are two independent data domains — users and machines. Each domain is exported by Perun as a separate JSON file. The detailed description of these entities:

Users

The user dataset contains information about all registered users in the MetaCentrum infrastructure, including their identifiers, names, organizational affiliations, and assigned virtual organizations. These data are used to enrich job statistics and to provide a link between computational activity and user identity.

Machines

In addition to management of users, Perun also collect information about physical computing resources of the MetaCentrum infrastructure. These data are then passed to PBSmon from Perun. Data are hierachically structured and grouped by Organization -> Cluster -> Computing Node.

Each record is an institution that has at least one cluster. Cluster is a group of computing nodes that are owned by the same institution. Computing node is a physical machine that is part of the cluster.

Each computing node is described by the following metadata:

- CPU configuration
- Memory

- Storage
- Owner institution
- The list of individual node hostnames

These data are neccessary to get additional information about the complete information about the computing node for the running PBS jobs. Additionally, these data independetly from PBS are important for knowing the complete information about the computing nodes for whole MetaCentrum infrastructure.

1.3.2 Data Collection Mechanism

The integration of the PBSmon system with Perun is designed using a PUSH model. Instead of direct access to the Perun database or invoking its API, Perun periodically generates JSON files containing all relevant information about users and computing resources. These files are then transferred via SSH directly to the PBSmon server, where they are stored in the filesystem, and subsequently loaded and processed by PBSmon. These files are stored in the filesystem as following:

- **/etc/pbsmon/pbsmon_users.json**
- **/etc/pbsmon/pbsmon_machines.json**

Change Detection and Synchronization

Currently, whenever user opens any page in PBSmon, the system checks if the files were modified since last load. If they were, the files are loaded and processed by PBSmon.

1.4 Data Collection from Virtual Machines (OpenStack)

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms. [3]

Beyond standard infrastructure-as-a-service functionality, additional components provide orchestration, fault management and service management amongst other services to provide operators flexibility to customize their infrastructure and ensure high availability of user applications. [3]

Within Metacentrum infrastructure, OpenStack is used to provide virtual machines to the users with specific needs that are not covered by the PBS environment. There are some computing nodes that are part of MetaCentrum infrastructure, but are reserved ad hoc for OpenStack. Within nodes returned by Perun mentioned in 1.3.1 are also nodes that are used for OpenStack.

Current PBSmon implementation gets very limited information about OpenStack virtual machines for each cluster. For each cluster, there is a list of virtual machines with their reserved CPU, name and user ID.

1.4.1 Data Collection Mechanism

The integration of the PBSmon system with OpenStack is designed using a PUSH model. Instead of direct access to the API, JSON files are transferred via SSH directly to the PBSmon server, where they are stored in the filesystem, and subsequently loaded and processed by PBSmon. For each cluster, there is a separate file with the list of virtual machines with their reserved CPU, name and user ID.

An example of the response structure for the OpenStack cluster named "glados" is provided in Appendix A.

1.5 Retrieval of historical data

As mentioned in section 1.2.2, the data collected by PBSmon represents only the current state of the PBS infrastructure. To access historical information about completed jobs, their resource usage, and long-term statistics, a separate application is responsible for collecting and storing historical PBS data in a PostgreSQL database.

This historical data collection system continuously records information about finished jobs, including their execution times, resource consumption, and user associations.

1.6 Authentication

The current PBSmon solution uses third-party authentication provided by Perun's e-INFRA system. Authentication is implemented using the OpenID Connect (OIDC) protocol, which allows users to authenticate through the centralized identity provider managed by Perun. [2]

This approach ensures that only authorized users with valid credentials from the Perun identity management system can access the PBSmon web interface. [2]

The authentication flow follows the standard OIDC protocol, where users are redirected to the Proxy IdP for login, and upon successful authentication, an OIDC token is issued and used to grant access to the PBSmon application. [2]

1.7 Web Layer and User Interface

The current PBSmon application is integrated as part of the MetaVO portal (`https://metavo.metacentrum.cz`), which serves as the main web interface for MetaCentrum services. Within MetaVO, PBSmon is encapsulated as a single subsection called "Current state" (see 1.1). This section is available only to logged-in users who have been granted access to PBSmon through the authentication system described in section 1.6.

The "Current state" subsection consists of the following pages:

- **Personal view** - A dashboard with information about the logged-in person, including their job statistics, jobs queues and quick access to other relevant pages
- **QSUB assembler** - A tool for assembling commands for job submission, described in detail in subsection 1.7.2
- **Physical machines** - A list of Metacentrum grid infrastructure retrieven from Perun nodes with PBS mapping
- **PBS node state** - A list of all PBS nodes and their current status. Without mapping to the Perun physical machines.
- **Virtual machines** - A list of all virtual machines (no longer relevant in the current infrastructure)
- **Jobs queues** - A list of queues and reservations
- **Jobs** - A page with total statistics and navigation hub to "My jobs", "All jobs", and "Suspicious jobs"
- **User** - A list of users from Perun and their current CPU usage
- **Machine properties** - A list of all PBS nodes and their properties (e.g., architecture, OS family, cluster, cgroups, etc.). Page allows to click on the property to see all the nodes that have this property.
- **List of hardware** - A list of organizations and their clusters
- **Cloud** - Cloud-related information (no longer relevant)

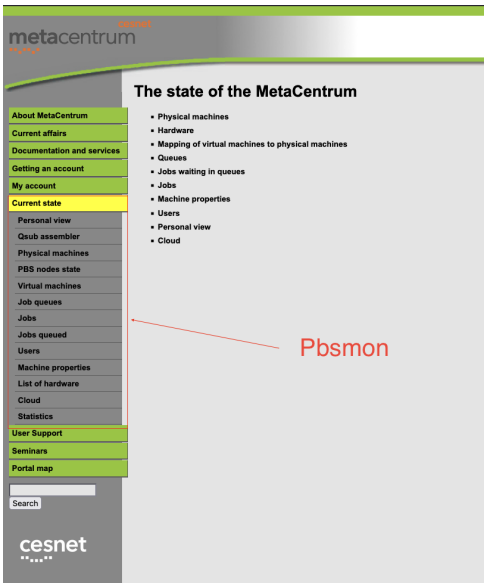

Figure 1.1: PBSmon layout

1.7.1 Personal view

The Personal view serves as the main dashboard for users, representing one of the most important and frequently used features of PBSmon. This view provides users with an overview of their job statistics and quick access to other relevant pages.

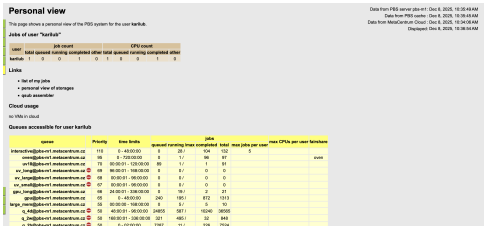

Figure 1.2: Personal view dashboard in the old PBSmon

The dashboard displays total counts of jobs associated with the user. However, since PBS always returns the current state of the system, the Personal view focuses on showing counts of *relevant* jobs—those that are considered active. Specifically, this includes jobs that are currently in the queue, currently running, or have finished within the previous

few days. This filtering ensures that users see meaningful information about their recent and ongoing computational work, rather than being overwhelmed by historical data from all past jobs.

The Personal view also contains links to other pages within the system. One such link leads to the "My jobs" page, which displays detailed information about the user's jobs. For user the list of his jobs are highly relevant, therefore it can be directly incorporated into the new dashboard within the new solution.

The page also displays all jobs queues that are accessible by the user. While this information is useful, it is considered less relevant and it is not necessary to be included in the new dashboard.

One limitation of the current implementation is that the dashboard only displays total CPU usage and does not include total GPU resources. This limitation stems from the fact that GPU usage was not a significant concern at the start of the old PBSmon development, but has since become increasingly important as GPU computing has gained prominence in the MetaCentrum infrastructure.

1.7.2 QSUB assembler

The QSUB assembler is one of the most important features of the PBSmon web interface, designed to assist users in submitting jobs to the MetaCentrum infrastructure. The `qsub` command is a CLI (Command Line Interface) command that allows users to submit jobs to the PBS system. This page provides a form-based interface for configuring job submission parameters.

The QSUB assembler page presents users with an extensive form containing numerous parameters that can be set for job submission, such as resource requirements (CPU flags, memory, scratch space, architecture), queue selection, walltime, and other MetaCentrum-specific options.

The current user interface presents parameters in a form equivalent to their syntax in the `qsub` command, which has the advantage that users gradually learn the `qsub` command syntax. However, the interface lacks supplementary explanations for individual fields, and therefore it is not always clear what the purpose of some less known or specific parameters is (e.g., `luna`, `umg`). The UI contains names directly according to PBS attributes, but without context or brief help, it can be

Figure 1.3: QSUB assembler interface

difficult for users to understand what exactly these choices represent or how they will affect job execution.

After configuring the desired parameters, the page automatically generates a complete shell command that can be used in two ways:

- Directly inserted into a terminal and executed to submit the job immediately
- Inserted as a comment directive in a PBS job script (e.g., `#PBS -l ...`) for later execution

Upon confirmation (submit), the page performs a query against the available MetaCentrum infrastructure to determine job feasibility. The results are presented in two key overviews:

- **Number of nodes meeting the requirements**: This shows the total count of nodes that are compatible with the specified parameters (e.g., nodes matching the CPU flags, memory requirements, scratch space, architecture, and other constraints)
- **Number of available nodes**: This displays the count of free nodes from the compatible set at the current moment, representing the real-time availability for job execution

Following these summary statistics, the page displays a detailed list of all nodes that match the specified requirements, providing users with list of nodes that can be used to submit the job.

1.8 Summary of the Analysis

The analysis of the existing PBSmon solution reveals a system that has successfully served its purpose of monitoring and visualizing MetaCentrum infrastructure data. The application effectively integrates data from multiple sources (PBS, Perun, and OpenStack) and provides users with views of computational resources, jobs, and system status. Key strengths include the QSUB assembler feature, which assists users in constructing job submission commands, and the successful integration with Perun's authentication system.

However, the analysis also identifies several significant limitations and areas for improvement. The system suffers from a **monolithic architecture** that tightly couples all components, making it difficult to maintain, extend, or deploy in modern containerized environments. The absence of dockerization further complicates deployment and scaling. The application relies on **outdated libraries**, most notably the Java library Stripes, which is no longer maintained, creating security and sustainability concerns.

From a user experience perspective, the interface presents parameters in a form that mirrors the `qsub` command syntax, which helps users learn PBS commands but lacks supplementary explanations for individual fields. This makes it difficult for users to understand the purpose of less common parameters (e.g., `luna`, `umg`) or how their choices will affect job execution. The web interface uses **server-side rendering** and is not implemented as a standalone frontend application, limiting flexibility and modern user experience capabilities.

The system also exhibits limitations in meeting current infrastructure demands. As GPU computing has gained prominence in MetaCentrum, the dashboard only displays total CPU usage statistics and does not include GPU resources or GPU time usage metrics. This reflects a common occurence with long-serving projects.

Another concern identified is a **potential GDPR violation**: the system displays information about all users that currently have active jobs and their jobs within MetaCentrum, which may not comply with data protection regulations regarding the visibility of personal computational activity to other users.

In summary, while PBSmon has been functional and valuable, its monolithic architecture, outdated technology stack, lack of con-

tainerization support, suboptimal user experience, incomplete feature coverage for modern computing needs, and potential privacy compliance issues necessitate a fundamental redesign and modernization of the system.

2 Design of the New Solution

Building upon the analysis in Chapter 1, this chapter focuses on the design of a modern replacement for the original PBSmon system. The goal is to propose a solution that not only preserves the existing functionality, but also improves maintainability, usability and alignment with current infrastructure and security requirements.

In addition to the functional rewriting of the existing features, the new design must introduce first-class support for working with OpenStack. In particular, it has to enable users to view OpenStack projects, the virtual machines assigned to them, and other relevant metadata in a clear and structured manner consistent with how resources are organised in the underlying cloud environment.

Furthermore, the design addresses the potential GDPR issue identified in Chapter 1, where personal computational activity may become visible to other users. To mitigate this risk, the new solution introduces a simple role model with two distinct roles: *user* and *admin*. This separation of privileges restricts access to sensitive information and ensures that operations affecting other users or system-wide configuration are only available to administrators.

The remainder of this chapter is structured as follows. Section 2.1 defines the functional requirements derived from the analysis and target use cases. Section 2.2 outlines the user interface design of the new application. Section 2.3 describes the overall system architecture, including the division into backend and frontend parts. Finally, Section 2.4 justifies the selection of the technologies used to implement the proposed design.

2.1 Functional Requirements

The goal of the new solution is to provide a read-only monitoring application that aggregates information from multiple data sources used in MetaCentrum. The system does not modify the state of the underlying infrastructure; instead, it offers a unified user interface for viewing job information, infrastructure topology and cloud resources. Access to this information is controlled by a simple role model with two roles: *user* and *admin*. An administrator can perform all actions available

to a regular user, plus additional operations related to support and diagnostics.

2.1.1 Domain Entities and Data Sources

The new solution reuses most of the domain model of the original PBSmon application and extends it with entities from the OpenStack environment. PBS and Perun entities were already introduced in the description of the existing solution (see Chapter 1); this section briefly summarises them and then focuses on the new OpenStack concepts that motivate several of the following requirements.

PBS and Perun (recap)

From PBS, the application uses primarily jobs, nodes, queues, reservations and fairshare entries, together with UNIX groups obtained from the `/etc/group` file to determine relationships between users. From Perun, it reuses the list of users with richer metadata (full name, organisation, ...) and the description of the MetaCentrum infrastructure, including organisations, clusters and their nodes. For a detailed description of these entities, see Chapter 1.

OpenStack entities

The new solution also obtains more detailed information from the OpenStack cloud environment. Two main entities are relevant for the design:

- **Project** – a logical container that owns cloud resources. Each project has an owner (either an individual user or a group), a date of creation and assigned resources (quotas). Projects can therefore represent both individual and group activities.
- **Virtual machine (VM)** – a virtual machine instance assigned to a specific project. For each VM, the system needs to track at least its name, associated project and creation time, so that it can be attributed to the correct user or group in the monitoring interface.

These OpenStack entities are used in the functional requirements below, in particular in the personal view for users and in the infrastructure overview.

2.1.2 Roles and Access Control

The system defines two roles:

- **User** – a regular MetaCentrum user. A user has access primarily to information related to their own activity and to entities connected to their UNIX groups. Global information is either anonymised or aggregated where necessary.
- **Admin** – an administrative user. An admin can see complete, non-anonymised information across all users and resources and can impersonate other users for troubleshooting and support. An admin has all capabilities of a regular user.

2.1.3 Functional Requirements for Users

A regular user must be able to use the application as a personal monitoring dashboard and as a tool for exploring the infrastructure relevant to their work. The following functional requirements apply:

FR-01 Personal overview The system shall provide a personal view (dashboard) where a user can see a summary of their total usage statistics, their recent and active PBS jobs, and their OpenStack projects.

FR-02 View own jobs The system shall allow a user to list and inspect the details of their own PBS jobs, including state, queue, submission time, requested resources and basic runtime information.

FR-03 View global jobs with anonymisation The system shall allow a user to browse all PBS jobs in the system. Jobs owned by other users shall be anonymised, except when the owner is:

- the current user, or

- a user belonging to one of the same UNIX groups as the current user.

In these cases, the real user identity shall be shown.

FR-04 View OpenStack projects and VMs The system shall allow a user to view all OpenStack projects associated with them and to list the virtual machines assigned to each of these projects, including basic metadata (such as name and creation date).

FR-05 View queues The system shall provide an overview of all PBS queues, including their basic configuration and purpose, so that users can understand where their jobs are being scheduled.

FR-06 View infrastructure and reservations The system shall provide a view of the MetaCentrum infrastructure based on Perun data, showing organisations, clusters and nodes, together with information about existing reservations on these resources. Only reservation information shall be displayed; actual usage of the resources is out of scope due to missing data.

FR-07 View users within groups The system shall allow a user to see basic information about other users who belong to the same UNIX groups, using data from `/etc/group`.

FR-08 View own group membership The system shall display the current user's group membership as derived from `/etc/group`, so that the user understands which groups they are associated with.

FR-09 View fairshare information The system shall display fairshare information relevant to the current user, derived from the PBS fairshare entries, and optionally allow comparison with other users in the same groups.

FR-10 QSUB assembler The system shall provide a "QSUB assembler" – an interactive interface that helps the user construct a valid `qsub` command by selecting queues, resources and other parameters. The result shall be presented in a form that can be copied and used in a terminal. Alltogether with the list of nodes that fullfils the requirements.

FR-11 Detail pages For all major entities (job, node, queue, project, VM, reservation, user), the system shall offer detail pages accessible from overview tables, with information limited according to the user's role and the anonymisation rules described above.

2.1.4 Functional Requirements for Admins

Administrators require a complete, non-anonymised view of the system in order to support users and diagnose problems. In addition to FR-01 – FR-11, the following requirements apply:

FR-12 Full visibility The system shall allow an admin to see all available data for all users, jobs, queues, infrastructure elements, projects and VMs without anonymisation.

FR-13 User impersonation The system shall allow an admin to impersonate a selected user, i.e., to temporarily view the application exactly as that user would see it (including all access restrictions and anonymisation). This functionality is intended solely for support and debugging.

These functional requirements form the basis for the graphical user interface design described in Section 2.2 and for the architecture of the new solution discussed in Section 2.3.

2.2 Graphical User Interface Design

A key part of the new solution is the redesign of the graphical user interface (GUI). The existing application has grown over time and its interface is not intuitive for new users, which makes common tasks unnecessarily difficult and increases the learning curve. To improve usability, it was necessary to propose a clearer and more consistent user interface that better reflects typical workflows and makes important information easier to find.

At the same time, the current users are already familiar with the existing interface and rely on established interaction patterns. A radical change to the layout or navigation could therefore have a negative impact on productivity and would likely be met with resistance. The GUI design of the new solution therefore aims for an evolutionary rather than revolutionary change: it introduces a more intuitive structure and visual hierarchy, but preserves similar navigation concepts and overall information density so that experienced users can adapt quickly.

Another important requirement was to keep the visual identity of the MetaCentrum computing grid[1]. The brand colour is a distinctive shade of orange, which is already used in existing tools and documentation and therefore forms an important part of the user experience. However, large orange areas on the screen can be visually tiring and, in extreme cases, reduce readability. For this reason, the new interface is based on a dark grey theme that serves as a neutral background, while orange is used as an accent colour for interactive elements such as primary buttons, active navigation items and important status indicators. This approach preserves the recognisable MetaCentrum brand while improving visual comfort during long-term use.

For the design process, the Figma tool was used to create the visual specification of the interface. In Figma, a set of basic building blocks was first defined as reusable components, ensuring consistency across the entire application and simplifying future modifications. On top of these components, the main layout was designed, including a sidebar menu for navigation between the key parts of the application and a content area for displaying overviews and detail views. The resulting

1. `https://www.metacentrum.cz/en/`

design of the new personal view is illustrated in Figure 2.1 and serves both as a communication tool with stakeholders and as a reference for the implementation of the frontend.

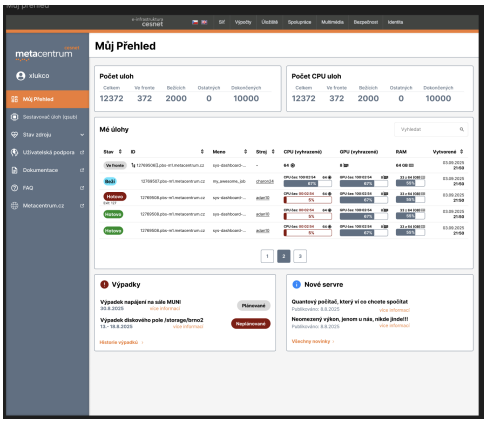

Figure 2.1: Figma design of the new personal view

2.3 Architecture Design

The new solution is designed as a two-layer web application consisting of a backend API and a separate frontend. The backend exposes the domain data and operations through a JSON-based HTTP API, while the frontend is responsible for rendering the user interface and handling user interaction in the browser. This separation follows a common pattern used in modern web applications and reflects the requirements for flexibility, maintainability and potential reuse of the data layer.

Both layers are developed in a single monorepository (monorepo). The monorepo contains the API and frontend as separate packages (or subprojects) that may share common code where appropriate (for example domain models or API client definitions), while still preserving a clear logical separation between the two layers.

2.3.1 Overview of the Two-Layer Architecture

The backend API layer integrates data from PBS, Perun and OpenStack, transforms them into a unified domain model and enforces the access

control rules based on the *user* and *admin* roles. It is implemented as a stateless web service that returns all responses in JSON format. Its only responsibility is to serve data and perform server-side computations such as aggregation, filtering and anonymisation.

The frontend layer is implemented as a separate web application that runs in the user's browser. It communicates with the API using HTTP requests, consumes the JSON responses and renders them into the graphical user interface described in Section 2.2. The frontend handles navigation between views, local state management (e.g. filters and sorting) and presents the data in tables, charts and detail pages.

2.3.2 API Layer

The API layer acts as an integration and abstraction layer over the underlying systems:

- It collects data from PBS, Perun and OpenStack and periodically fetches the necessary entities (jobs, nodes, queues, reservations, users, infrastructure descriptions, projects, virtual machines, ...).
- It normalises and combines the collected data into a consistent model tailored to the needs of the monitoring application.
- It implements role-based access control and anonymisation rules as described in the functional requirements, including the distinction between regular users and administrators.
- It exposes the data through a set of endpoints that return JSON, covering overviews (e.g. lists of jobs or projects) as well as detail views for individual entities.

By concentrating all domain logic and access control in the API, the system ensures that the same rules are applied consistently regardless of the client and that sensitive operations cannot be bypassed by directly accessing the underlying systems.

2.3.3 Frontend Layer

The frontend layer consumes the API and transforms the JSON data into an interactive web interface:

- It provides the personal view for users, including their own statistics, jobs and OpenStack projects.
- It renders overviews of jobs, queues, infrastructure and reservations, with client-side filtering and sorting where appropriate.
- It presents anonymised or full data depending on the current role, based on information obtained from the API.
- It implements the QSUB assembler, where the user selects parameters and the frontend assembles a command that can be copied to a terminal.

Because the frontend is decoupled from the backend at the API boundary, it can be developed and iterated independently at the application level, even though both parts live in the same monorepo. The visual design can evolve without changes to the data layer as long as the API contract remains stable.

2.3.4 Rationale, Advantages and Disadvantages

The choice of a two-layer architecture with a JSON-based API and a separate frontend, implemented together in a monorepo, was motivated by several considerations.

Among the main advantages are:

- **Separation of concerns** – the API focuses on data integration, business logic and security, while the frontend focuses on user experience and presentation. This separation simplifies reasoning about each part and reduces coupling.
- **Reuse and extensibility** – the API can later be reused by other clients (for example command-line tools, scripts or additional internal applications) without changes to the core logic. New frontend features can be added as long as they build on the existing endpoints.
- **Monorepo code sharing** – keeping the API and frontend in a single monorepo enables straightforward sharing of common

code, such as type definitions, domain models or API client libraries, which reduces duplication and the risk of inconsistencies between layers.

- **Coordinated development** – the monorepo makes it easy to evolve the API and frontend together in a single change set, while the architectural separation still allows developers to work on backend and frontend parts with clear boundaries.
- **Potential for independent deployment** – even though the code is stored in a single repository, the API and frontend can be built and deployed as separate artefacts if needed (for example scaling the API independently), which is important for handling peaks in monitoring queries.
- **Security** – all access control and anonymisation logic is centralised in the API layer, so there is a single place where role-based rules are enforced and audited.

The chosen architecture also has disadvantages that need to be acknowledged:

- **Increased operational complexity compared to a monolithic server-rendered application** – there are two logical components (API and frontend) that must be configured to communicate correctly, including handling of authentication and possible cross-origin restrictions, even though they are stored in one repository.
- **Network overhead** – every interaction between the user and the data layer requires HTTP requests from the browser to the API, which may introduce additional latency compared to server-side rendering directly coupled to the data sources.
- **Duplication of validation and error handling** – some validation and error reporting needs to be implemented both in the API (for correctness and security) and in the frontend (for user-friendly messages), which slightly increases implementation effort.
- **Monorepo maintenance** – while a monorepo simplifies code sharing, it also requires consistent tooling and conventions to keep the structure manageable as the project grows.

Despite these drawbacks, the benefits of a clear separation between the data layer and the user interface, combined with the practical advantages of a monorepo for this relatively small project, were considered more important. In particular, the need to integrate multiple heterogeneous data sources, enforce non-trivial access rules and support future extensions favours an API-centric architecture. The resulting design provides a flexible and maintainable foundation for the new PBSmon replacement.

2.4 Selected technologies

As mentioned earlier, the original solution was implemented as a web application. The new solution will also take the form of a web application, but it will be built on a different technology stack. When selecting the technologies, it was important to take into account current trends in web development, as well as future maintainability and extensibility.

As described in Section 2.3, the new solution should be conceptually divided into two parts: an API and a frontend. To simplify development and maintenance, JavaScript was chosen as the primary programming language, enabling both parts of the system to be implemented in the same language. Specifically, NestJS is used for the API layer and React for the frontend. These technologies are described in more detail in the following subsections.

2.4.1 NestJS

NestJS is a progressive Node.js framework designed for building efficient, reliable and scalable server-side applications. It leverages TypeScript as its primary language, while still supporting plain JavaScript. The framework is heavily inspired by architectural patterns known from enterprise environments, such as the Model–View–Controller (MVC) pattern and layered architectures, and it emphasises concepts like modules, controllers and providers. [4]

One of the main advantages of NestJS is its modular architecture. The application is structured into self-contained modules, which encapsulate related functionality. This organisation improves the read-

ability of the codebase and simplifies long-term maintenance and extensibility. NestJS also provides built-in support for dependency injection, which encourages loose coupling between components and facilitates testing. [4]

According to articles [5, 6], NestJS is well suited for backend development and has accumulated more than 60,000 stars on GitHub, placing it among the world's most popular backend frameworks. It is a modern, flexible and powerful framework that is gaining popularity thanks to its strong TypeScript support, modular architecture and mature ecosystem.

For these reasons, NestJS was chosen as the framework for implementing the API layer of the new solution.

2.4.2 React

React is a JavaScript library for building user interfaces using a declarative, component-based model. It allows composing complex views from smaller reusable components and integrates well with modern JavaScript and TypeScript tooling. [7]

In this project, React is used as a purely client-side rendered frontend. Rendering of the user interface is thus offloaded from the server to the client, which reduces the load on the backend API and simplifies the server-side implementation. The main trade-off of client-side rendering is weaker support for search engine optimisation (SEO) compared to server-side rendering or static site generation. However, SEO is not required in this system, as it targets authenticated users and is not intended for public indexing. For this reason, a purely client-side React application is sufficient and avoids additional complexity. [7]

React was chosen mainly due to its wide adoption, mature ecosystem and good TypeScript support, which make it a suitable choice for the frontend part of the solution [7].

Tailwind CSS

Tailwind CSS is a utility-first CSS framework that provides a set of predefined classes for styling HTML elements. It is designed to be used in conjunction with React, allowing for a more efficient and consistent

styling approach. Tailwind CSS is a popular choice for modern web development due to its ease of use, flexibility and performance. [8]

I have chosen Tailwind CSS due to its popularity and ease of use. It is a popular choice for modern web development and is a good fit for the new solution.

2.4.3 Nginx

Nginx ("engine x") is a high-performance HTTP web server and reverse proxy that can also act as a load balancer and content cache. It is designed to handle a large number of concurrent connections with low resource usage, which makes it suitable as an entry point for web applications. [9]

In this project, Nginx is used primarily as a reverse proxy in front of the backend API and as a static file server for the React frontend. Requests to the API are forwarded from Nginx to the NestJS application, while requests for frontend assets are served directly from Nginx. This setup enables centralised configuration of TLS termination, request routing and basic security headers, while the application servers focus on business logic. [9]

As mentioned in the section 2.3, the new solution consists of separate modules — the API and the frontend — Nginx is used to provide a single unified interface to the outside world. Both parts of the system are exposed under one domain and port, and the internal structure is hidden from clients, which simplifies deployment and future maintenance.

2.4.4 Docker

Docker is a platform for building, distributing and running applications in lightweight containers. A container packages the application together with its runtime dependencies and configuration, which makes the resulting environment reproducible across different machines. [10]

In this project, Docker is used to containerise the backend API, the frontend and supporting services (such as Nginx). Each service runs in its own container, while a shared configuration (e.g. using `docker-compose`) defines how the containers are connected and started.

This approach simplifies local development, testing and deployment, as the entire system can be started or stopped with a single command and behaves consistently across environments. [10]

3 Implementation

This chapter describes the implementation of the new PBSmon solution. The main objective was not only to migrate the legacy monolithic system to a modern technology stack, but also to deliver a maintainable and extensible codebase that supports the current scope of MetaCentrum infrastructure. In particular, the implementation covers monitoring data obtained from the PBS scheduling environment and extends the original functionality by integrating OpenStack as an additional data source.

The resulting system is implemented as a separated backend and frontend that communicate via a JSON API. This separation enables clearer responsibility boundaries, simplifies testing, and allows both parts to evolve independently. At the same time, the implementation keeps the core purpose of PBSmon intact: regularly collecting infrastructure metadata, normalizing it into a consistent representation, and presenting it to users in a readable form.

3.1 Authorization and Authentication

TBD - using Perun's e-infra AAI, using OIDC token

3.2 System Architecture

This chapter describes the implementation of PBSmon 2.0 and explains how the individual parts of the system were realized in code. The project is implemented as a monorepo and consists of multiple runtime components that together provide data collection, a JSON API, and a web-based user interface. The repository is structured into the following main components:

- `api/` – backend service exposing a JSON API consumed by the frontend. It aggregates and normalizes monitoring data from the supported infrastructure sources (PBS, Perun and OpenStack).
- `web/` – frontend web application responsible for rendering page views and visualizing the collected monitoring data.

- `pbs-collector/` – a dedicated micro-service for periodic data acquisition from PBS environment. Described in more detail in section 3.3.1.
- `nginx/` – reverse proxy configuration used as the entry point to route requests to the web application and the API.
- `docker-compose.prod.yml` – production deployment definition describing services, networking, and environment configuration.
- `deploy.sh` – deployment helper script that automates the deployment process.

At runtime, user requests are handled through the reverse proxy, which serves the frontend and forwards API calls to the backend. The frontend then communicates with the backend exclusively through the JSON API, while the collector runs independently to keep the monitoring data up to date.

3.3 API

The backend API forms the integration layer between the monitored infrastructure and the web user interface. Its primary purpose is to provide a JSON API for the frontend while hiding the complexity of individual data sources. In PBSmon 2.0, the API aggregates monitoring data from the PBS environment, Perun and OpenStack, normalizes them into a consistent representation, and exposes them through JSON endpoints.

API code structure

The API source code is organized into the following top-level directories:

- `src/` – source code of the API.
  - `common/` – shared utilities and cross-cutting concerns.
  - `config/` – application and environment configuration. Eq. QSUB assembler configuration, pbs server list, etc.
  - `modules/` – feature-oriented modules. Each module represent a domain entity and its related data. Each module consists of controllers and services. And therefore each module have its own routes and endpoints. There are list of modules:
    - `accounting/` – module for collecting accounting data from the accounting database.
    - `app/`
    - `data-collection/` – module responsible for collecting data from the infrastructure sources.
    - `groups/` – module for retrieving user groups from the PBS server.
    - `infrastructure/` – module for the infrastructure overview.
    - `jobs/` – module for the jobs overview.
    - `projects/` – module for the OpenStack projects overview.
    - `qsub/` – module for the QSUB assembler.
    - `queues/` – module for the queues overview.
    - `status/` – module for the status overview.
    - `storage-spaces/` – module for the storage spaces overview.
    - `users/` – module for the users overview.
- `data/` – runtime collected data from the infrastructure sources.

3.3.1 PBS data collection

During the implementation, an important architectural decision concerned the placement of the PBS data collection logic: whether it

should be part of the main API service or separated into an independent microservice (`pbs-collector`). The final design uses a dedicated collector because the PBS integration introduces specific and operationally demanding requirements (installation of the `libopenpbs` IFL library, a Debian-based runtime, and a properly configured Kerberos environment). In addition, prior experience with the legacy solution showed stability issues caused by memory leaks in the OpenPBS library, which could lead to process instability and, in the worst case, affect the entire server. By isolating the collector in a Docker container with explicitly defined resource limits, such failures are contained: once the limits are exceeded or the process becomes unstable, the container can be automatically restarted without impacting the API and the user interface.

The `pbs-collector` microservice is therefore responsible for periodic retrieval of PBS data and exporting it into filesystem for later processing by the API. It collects core PBS entities (e.g., jobs, nodes, queues, and related objects) using the IFL interface of `libopenpbs` (as described in Section 1.2.2), and it also gathers fairshare values for the users via the `list_cache` command (as described in Section 1.2.1). This separation keeps the API focused on data delivery and presentation concerns, while the collector encapsulates the low-level PBS-specific integration and its operational constraints.

3.3.2 Perun data collection

Perun is not collected by PBSmon in the same way as PBS or OpenStack, because the required data are provided to PBSmon via a push mechanism external to the application. For this reason, PBSmon does not implement an active Perun client; instead, it only consumes prepared Perun exports.

Within the API repository, Perun-related inputs are stored under `/api/data/perun/`. This directory contains static JSON files:

- `pbsmon_users.json` – users data from Perun.
- `pbsmon_machines.json` – machines data from Perun.

These files were described in more detail in section 1.3.1. During development, these files were mocked to enable frontend and API

development without relying on the external Perun export pipeline. In production, the application follows the same approach as the legacy PBSmon and loads these JSON files directly from the filesystem at runtime.

3.3.3 OpenStack data collection

TBD - using prometheus program - that was describe in some non existing section - these data stores only into memory and are not collected to the filesystem.

3.3.4 Documentation

The API contract is described using an OpenAPI specification, which defines the available endpoints, their parameters, and response schemas. In addition to the static specification, the implementation exposes a Swagger UI that renders the OpenAPI document and provides an interactive interface for exploring and testing the API. This allows developers and administrators to call individual endpoints directly from the browser, inspect responses, and verify the expected behaviour without requiring external tools. The OpenAPI specification is included within the thesis archive.

3.4 Frontend - page views

This section describes the implemented frontend page views and how they present monitoring information to end users. Each view corresponds to a specific domain area of the system (e.g., jobs, nodes, queues, outages, news, or OpenStack-related data) and is implemented as a composition of reusable UI components such as tables, filters, and detail panels. The pages obtain all data exclusively through the backend JSON API, which keeps the frontend independent of the underlying infrastructure protocols and allows the UI to evolve without changing data-collection logic.

Layout TBD - describe layout and navigation, the lists of page views

3.4.1 Personal view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

3.4.2 QSUB assembler

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

3.4.3 Machines

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

Node detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

3.4.4 Storage spaces

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

3.4.5 Cloud Projects

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

Project detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

3.4.6 Jobs Queues

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

Queue detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

3.4.7 Jobs

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image of the page

Job detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

3.4.8 Users

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

User detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases
- Image??

3.4.9 Users groups

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases

Group detail view

TBD

- purpose
- displayed data + user interactions
- API endpoints
- covered use cases

3.5 Comparison with the legacy solution

TBD - compare the new solution with the legacy solution in terms of features, performance, scalability, maintainability, etc.

4 Future improvements

TBD -

- Kubernetes extensions - showing state of kubernetes infrastructure
- Export to CSV - for analytical purposes - for admin users
- Showing personal disk quotas within user
- Option to submit jobs using the API - for users

Conclusion

TBD - simply state conclusion

A OpenStack Response Example

This appendix contains an example of the JSON response structure for the OpenStack cluster named "glados" that is collected by the PBSmon system.

Listing A.1: Example response for OpenStack cluster "glados"


```
[
    {
        "CPUs": "40",
        "Hypervisor": "ics-gladosag-007-ostack.
           priv.cloud.muni.cz",
        "VMs": [
            {
                "CPUs": "38",
                "created": "2020-10-09T09:32:42Z
                   ",
                "instance_state": "ACTIVE",
                "name": "RationAI-node-2",
                "user_id": "1633180
                   b677608f61e96784ee5cbc608c0f4b62d@einfra
                   .cesnet.cz"
            }
        ]
    },
    {
        "CPUs": "40",
        "Hypervisor": "ics-gladosag-006-ostack.
           priv.cloud.muni.cz",
        "VMs": []
    },
    {
        "CPUs": "40",
        "Hypervisor": "ics-gladosag-003-ostack.
           priv.cloud.muni.cz",
        "VMs": [
            {
```

```
                "CPUs": "38",
                "created": "2020-01-17T10:29:22Z
                   ",
                "instance_state": "ACTIVE",
                "name": "RationAI-node-1",
                "user_id": "1633180
                   b677608f61e96784ee5cbc608c0f4b62d@einfra
                   .cesnet.cz"
            }
        ]
    },
    {
        "CPUs": "40",
        "Hypervisor": "ics-gladosag-004-ostack.
           priv.cloud.muni.cz",
        "VMs": []
    },
    {
        "CPUs": "40",
        "Hypervisor": "ics-gladosag-001-ostack.
           priv.cloud.muni.cz",
        "VMs": [
            {
                "CPUs": "16",
                "created": "2022-08-17T13:07:15Z
                   ",
                "instance_state": "ACTIVE",
                "name": "front-82e95edc-1e2d-11
                   ed-9687-0ee20d64cb6e",
                "user_id": "9
                   cd61d48508661633e261f711634b749fdc5d9fcc20769
                   .eu"
            }
        ]
    }
]
```

Bibliography

1. *Altair PBS Professional 2022.1 - Big Book* [online]. Troy (MI): Altair Engineering, Inc., 2022-07-16 [visited on 2022-07-16]. Available from: https://help.altair.com/2022.1.0/PBS%20Professional/PBS2022.1.pdf.
2. *Perun AAI: The solution for identity and access management* [online]. Perun AAI, 2025 [visited on 2025-01-27]. Available from: https://perun-aai.org/.
3. *OpenStack: The Most Widely Deployed Open Source Cloud Software in the World* [online]. OpenInfra Foundation, 2025 [visited on 2025-01-27]. Available from: https://www.openstack.org/.
4. *NestJS - A progressive Node.js framework*. [N.d.]. Available also from: https://nestjs.com/. Accessed: 2025-12-09.
5. *Why NestJS is the new gold standard for Node.js backend development*. Available also from: https://dev.to/rayenmabrouk/why-nestjs-is-the-new-gold-standard-for-node-backend-development-lm. Accessed: 2025-12-09.
6. *2025 and NestJS: A Match Made for Modern Backend Needs*. LeapCell. Available also from: https://leapcell.medium.com/2025-and-nestjs-a-match-made-for-modern-backend-needs-5d257d4061be. Accessed: 2025-12-09.
7. *React Documentation*. [N.d.]. Available also from: https://react.dev/. Accessed: 2025-12-09.
8. *Tailwind CSS*. Tailwind Labs, [n.d.]. Available also from: https://tailwindcss.com/. Accessed: 2025-12-09.
9. *nginx Documentation*. [N.d.]. Available also from: https://nginx.org/en/docs/. Accessed: 2025-12-09.
10. *Docker Documentation* [https://docs.docker.com/]. [N.d.]. Accessed: 2025-12-09.