Consiglio Nazionale delle Ricerche

Istituto di Calcolo e Reti ad Alte Prestazioni

Custom JupyterHub Deployment on H2IOSC Cluster

Luigi Barbato, Francesco Gargiulo

Technical Report: **RT-ICAR-NA-2026-02** ***Data:*** **January 2026**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Napoli, Via P. Castellino 111, I-80131 Napoli, Tel: +39-0816139508, Fax: +39-0816139531, e-mail: napoli@icar.cnr.it, URL: *www.na.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Custom JupyterHub Deployment on H2IOSC Cluster

Luigi Barbato, Francesco Gargiulo

Technical Report: **RT-ICAR-NA-2026-02** ***Data:*** **January 2026**

Custom JupyterHub Deployment on H2IOSC Cluster

Luigi Barbato (1,2) e Francesco Gargiulo (1)
luigi.barbato@icar.cnr.it, francesco.gargiulo@icar.cnr.it

(1) **Istituto di Calcolo e Reti ad Alte Prestazioni del Consiglio Nazionale delle Ricerche (ICAR-CNR)**
Via Pietro Castellino, 111 – 80131 Napoli, Italia

(2) Università Campus Bio-Medico di Roma

Via Alvaro del Portillo, 21 — 00128 Roma

1. Abstract

This technical report details the implementation of a high-performance computing environment utilizing a Kubernetes-orchestrated JupyterHub infrastructure. The platform constitutes a state-of-the-art research solution engineered to accommodate a broad spectrum of computational workloads, including quantum computing simulations and advanced artificial intelligence research.

Deployed on a robust six-node cluster with specialized hardware procured by CNR-ICAR under the PNRR project "Humanities and Cultural Heritage Italian Open Science Cloud" (H2IOSC) this infrastructure supports Italy's strategic initiative to develop and integrate national Research Infrastructures (IR). The system seamlessly unifies traditional Jupyter notebooks with fully-featured web development environments, offering a cohesive platform for both scientific computing and software development.

Its architecture strikes a deliberate balance among performance optimization, rigorous security enforcement, and user accessibility, making it equally suitable for computational researchers and developers working on resource-intensive applications.

The document is organized as follows: Section 2 presents a comprehensive overview of the hardware specifications; Section 3 outlines the software stack and the implementation of

security policies; Section 4 describes the platform's operational modes, focusing on JupyterHub access and resource management; and Section 5 provides concluding remarks.

2. Infrastructure Hardware Specifications

The hardware foundation of this research computing platform has been carefully architected to provide both exceptional raw computational power and the sophisticated management capabilities necessary to make that power accessible and efficient. The infrastructure follows a classic Kubernetes design pattern, separating control plane management from computational workload execution, but implements this pattern with enterprise-grade hardware that pushes the boundaries of what's possible in research computing environments.

2.1 Control Plane Infrastructure (Master Nodes ×3)

The control plane represents the nervous system of the entire infrastructure, responsible for orchestrating the complex ballet of containers, scheduling workloads, maintaining cluster state, and ensuring high availability. This critical function is distributed across three dedicated master nodes, each representing a substantial investment in reliability and computational capability.

Each master node is built around an impressive dual-processor architecture, featuring two processors with 24 cores each, yielding a total of 48 physical cores per node. This substantial processing capacity ensures that the control plane never becomes a bottleneck, even under the most demanding operational conditions when managing hundreds of containers and complex scheduling decisions.

Hardware Specifications per Master Node:

- Processing Capacity: 48 total cores per node (dual processors of 24 cores each)
- Memory Architecture: 512GB DDR4 ECC RAM
- Storage: SSD array optimized for ETCD high availability

The processors are complemented by massive ECC (Error-Correcting Code) memory, providing not only the capacity needed for extensive cluster state management but also the reliability guarantees essential for mission-critical infrastructure. The ECC memory actively detects and corrects memory errors, preventing the data corruption that could cascade into cluster-wide failures. Storage architecture on the master nodes centers around SSD arrays specifically optimized for the demanding I/O patterns of ETCD, Kubernetes' distributed key-value store. ETCD serves as the single source of truth for all cluster state, and its performance directly impacts the responsiveness of the entire infrastructure.

Cluster Configuration:

- Nodes: umaster01, umaster02, umaster03
- Internal IP Addresses:
  - umaster01: 192.168.101.21
  - umaster02: 192.168.101.22
  - umaster03: 192.168.101.23
- Functions: ETCD cluster hosting, Kubernetes API Server, Controller Manager, Scheduler

The three master nodes form a highly available ETCD cluster, implementing a Raft consensus algorithm that ensures data consistency even in the face of node failures. Beyond hosting the ETCD cluster, these nodes run the essential Kubernetes control plane components: the API Server, which serves as the front-end to the Kubernetes control plane; the Controller Manager, which runs the control loops that regulate cluster state; and the Scheduler, which makes decisions about pod placement across the worker nodes.

2.2 Worker Infrastructure (Worker Nodes ×3)

While the control plane provides the intelligence and orchestration capabilities of the cluster, the computational infrastructure is where the real work happens. The three worker nodes represent an extraordinary concentration of computational power, specifically optimized for GPU-intensive workloads and modern machine learning applications. Each worker node is a computational powerhouse in its own right, but together they form an

integrated system capable of tackling problems that would be impractical or impossible on traditional computing platforms.

Hardware Specifications per Worker Node:

- Processing Capacity: 256 total cores per node
- Memory Configuration: 2TB DDR4 ECC RAM (~2,113GB per node)
  - Total cluster memory: 6TB RAM available for user workloads
- Graphics Processing Array:
  - 8× NVIDIA A100-SXM4-80GB per node
  - Ampere architecture
  - 80GB HBM2 memory per GPU
  - Compute capability 8.0
  - Total cluster: 24× NVIDIA A100 GPUs
  - Aggregate computational power: ~120 PFLOPS (FP16 precision)

The processing architecture features massive parallel processing capabilities for CPU-bound tasks. This core count enables sophisticated workload parallelization, allowing researchers to efficiently utilize all available processing power for tasks ranging from data preprocessing to model training and validation. The CPU architecture works in concert with the GPU resources, handling tasks like data loading, preprocessing, and orchestration while the GPUs focus on the computationally intensive matrix operations central to modern machine learning.

The ECC protection ensures data integrity during long-running computational jobs, where even a single bit flip could invalidate hours or days of computation. The crown jewel of the computational infrastructure is undoubtedly the GPU array. These aren't consumer-grade GPUs repurposed for research; they're datacenter-class accelerators specifically engineered for the demanding requirements of scientific computing and artificial intelligence research. The SXM4 form factor enables higher power delivery and better thermal management than PCIe alternatives, critical for sustained computational loads.

Each A100 GPU features 80GB of HBM2 (High Bandwidth Memory) memory, providing not just capacity but also the memory bandwidth essential for feeding the GPU's computational units. The A100 architecture supports advanced features like tensor cores for accelerated mixed-precision training and multi-instance GPU (MIG) technology for GPU virtualization. With this configuration, the cluster delivers computational capability that places this infrastructure among the most powerful research computing platforms available.

Worker Cluster Configuration:

- Nodes: uworker01, uworker02, uworker03
- Internal IP Addresses:
  - uworker01: 192.168.101.24
  - uworker02: 192.168.101.25
  - uworker03: 192.168.101.26

2.3 Network Architecture and Connectivity

The network architecture represents the invisible infrastructure that transforms six independent servers into a cohesive, high-performance computing cluster. Every aspect of the networking design has been carefully considered to balance performance, security, and operational simplicity, creating an environment where containers can communicate efficiently while maintaining proper isolation and security boundaries.

At the foundation of the internal cluster networking sits the Container Network Interface (CNI) implementation. The infrastructure employs Canal CNI, which represents an elegant fusion of Calico and Flannel technologies. Calico provides sophisticated network policy enforcement capabilities, enabling fine-grained control over which pods can communicate with each other, while Flannel handles the overlay networking that allows pods to communicate across physical hosts.

Internal Cluster Network Fabric:

- Container Network Interface: Canal CNI (Calico + Flannel integration)
- IP Address Management:

    - Pod CIDR: 10.42.0.0/16 (65,536 addresses available for pods)
    - Service CIDR: 10.43.0.0/16 (65,536 addresses for Kubernetes services)
- Cluster DNS: RKE2 CoreDNS
    - Internal domain: cluster.local
    - Automatic service resolution across the entire cluster

This combination delivers both the security features necessary for a multi-tenant environment and the networking simplicity that makes the system manageable. The substantial address space ensures that even with hundreds of concurrent user sessions and supporting services, the cluster will never exhaust its pool of addresses. Name resolution works transparently across the entire cluster, so a container on uworker01 can seamlessly communicate with a service running on uworker03 using simple DNS names, without any knowledge of the underlying network topology.

External Access and Service Exposure:

- Load Balancing: MetalLB (Layer 2 mode)
    - IP Pool: 192.168.101.101–192.168.101.150
    - Static IP allocation for LoadBalancer services
- Ingress Management: NGINX Ingress Controller
    - Hostname-based routing
    - TLS termination and certificate management
    - Virtual hosting support
- DNS Infrastructure:
    - h2iosc.icar.cnr.it/jupyterhub → 192.168.101.108 (JupyterHub service)
    - h2iosc.icar.cnr.it/sso→ 192.168.101.102 (Keycloak SSO service)
- TLS Certificates:
    - Management via Kubernetes Secrets
    - jupyterhub-tls certificate for HTTPS connections
    - Server validation disabled for internal Keycloak

External access presents unique challenges in on-premises Kubernetes deployments, where cloud-provider integration isn't available. MetalLB addresses this through Layer 2 mode, using ARP (Address Resolution Protocol) to announce service IP addresses on the physical network. This approach enables the cluster to expose services with static IP addresses, providing the same experience users expect from cloud-based Kubernetes platforms. The NGINX Ingress Controller implements sophisticated reverse proxy capabilities, enabling multiple services to share the same external IP address while being distinguished by their DNS names, handling TLS certificate management and SSL termination automatically.

3. Software Architecture and Security Implementation

3.1 Foundation Layer Specifications

The software stack begins with the host operating system, forming the foundation upon which all higher-level services are built. The infrastructure standardizes on Ubuntu Server 22.04.5 LTS (Long Term Support), a deliberate choice that balances cutting-edge features with enterprise stability. The LTS designation ensures that security updates and bug fixes will be available for years, providing a stable foundation for the research platform.

Host Operating System:

- Distribution: Ubuntu Server 22.04.5 LTS
- Kernel Versions:
  - Master nodes: Linux 5.15.0-161-generic
  - Worker01: Linux 5.15.0-125-generic
  - Worker02–03: Linux 5.15.0-161-generic

The kernel versions vary slightly across the cluster, reflecting the practical realities of a living infrastructure that has evolved over time. These kernel versions all belong to the 5.15 series, maintaining compatibility while allowing for incremental updates and hardware-specific optimizations. The kernel is responsible for managing hardware resources, scheduling processes, and enforcing security policies at the lowest level of the software stack.

Container Orchestration and Runtime:

- Kubernetes Platform: Kubernetes v1.30.6+rke2r1
- Distribution: RKE2 (Rancher Kubernetes Engine 2)
- Container Runtime: containerd v1.7.22-k3s1
- Package Management: Helm v3.19.0

Container orchestration is provided through RKE2, SUSE's hardened, enterprise-ready Kubernetes distribution designed with security and compliance requirements in mind from the ground up. Unlike some Kubernetes distributions that treat security as an add-on, RKE2 embeds security best practices throughout the stack, from CIS benchmark compliance to automated certificate rotation and security policy enforcement. The container runtime layer uses containerd, which handles the actual creation and management of containers, interfacing directly with the kernel's container primitives. Package management for Kubernetes applications is handled by Helm, often described as the package manager for Kubernetes, which allows complex applications to be packaged as 'charts' that encapsulate all the resources needed to run an application.

Additional Cluster Features:

- GPU Operator: Complete NVIDIA GPU Operator
  - DCGM (Data Center GPU Manager) for monitoring
  - Device plugin for GPU scheduling
  - Automatic feature discovery
  - Driver version compatible with CUDA 11.8+
- Cluster Age: 363 days (operational for approximately 1 year)
- Average Uptime: >99.5% (with planned maintenance windows)

The GPU capabilities that make this infrastructure suitable for AI and scientific computing are enabled through the complete NVIDIA GPU Operator. This sophisticated piece of software automates the deployment and management of all the components necessary for GPU acceleration in Kubernetes. The cluster has been operational for 363 days at the time

of this documentation, approaching one full year of continuous service. Throughout this period, the infrastructure has maintained an impressive average uptime exceeding 99.5%, demonstrating that the system can serve as a dependable foundation for long-running research projects.

3.2 JupyterHub Configuration and Architecture

JupyterHub serves as the primary user interface to the computational infrastructure, transforming the raw power of the Kubernetes cluster into an accessible, web-based research environment. The JupyterHub deployment has been carefully configured to provide a robust, scalable multi-user experience while maintaining enterprise-grade authentication through Keycloak integration.

Core Components:

- Hub Pod: Main JupyterHub deployment
  - Image: quay.io/jupyterhub/k8s-hub:4.3.1
  - Manages authentication, user spawning, and proxy routing
  - Database: SQLite with persistent volume (1Gi)
  - Strategy: Recreate (ensures database consistency)
- Proxy Pod: ConfigurableHTTPProxy (CHP)
  - Image: quay.io/jupyterhub/configurable-http-proxy:5.1.0
  - Dynamic routing to user pods
  - API on port 8001 for Hub communication
  - Strategy: Recreate (avoids routing inconsistencies)
- User Scheduler: Custom Kubernetes scheduler
  - Image: registry.k8s.io/kube-scheduler:v1.30.14
  - Replicas: 2 (high availability)
  - MostAllocated strategy for pod consolidation
  - Log level: 4 (detailed operational logs)

The core of JupyterHub runs in a dedicated hub pod, responsible for managing the entire JupyterHub ecosystem, handling user authentication, spawning and tracking user notebook servers, and coordinating with the proxy for request routing. The hub maintains its state in a SQLite database backed by a persistent volume claim, ensuring that user server assignments and authentication state survive pod restarts. The deployment strategy is configured as 'Recreate' rather than the default 'RollingUpdate', a deliberate choice that ensures database consistency given SQLite's single-writer constraint.

Traffic routing to user servers is managed by a separate proxy pod running ConfigurableHTTPProxy (CHP), a node.js-based proxy that implements dynamic routing. The proxy maintains an API server specifically for communication with the hub, allowing the hub to programmatically update routing rules as user servers start and stop. Like the hub, the proxy uses a Recreate deployment strategy to avoid routing inconsistencies.

A particularly sophisticated element of the deployment is the custom user scheduler, which represents a fork of Kubernetes' own kube-scheduler configured specifically for JupyterHub's workload patterns. The user scheduler runs with two replicas for high availability and implements a 'MostAllocated' scoring strategy, a counterintuitive but highly effective approach for batch workloads. Rather than spreading pods across all available nodes, MostAllocated concentrates pods onto fewer nodes, leaving some nodes completely idle and enabling more efficient scale-down and power management.

Authentication is handled through a GenericOAuthenticator configured to integrate with the Keycloak instance at sso.icarna.lan. The OAuth flow uses client credentials (h2iosc as the client ID) to enable single sign-on, meaning users authenticate once with Keycloak and gain access to JupyterHub without additional prompts. The OAuth callback is configured to handle the return journey from Keycloak after authentication at https://h2iosc.icar.cnr.it/jupyterhub/hub/oauth_callback. The system is configured with 'allow_all: true', meaning any user successfully authenticated by Keycloak can access JupyterHub, though two users (francesco.gargiulo and luigi.barbato) are designated as administrators with enhanced privileges.

The JupyterHub service configuration addresses specific operational requirements of the environment. The base URL is set to '/jupyterhub', allowing JupyterHub to run behind a reverse proxy at a non-root path. Cookie handling is configured with 'SameSite: None' and 'Secure: True' to enable proper session management in the cross-origin context created by the OAuth authentication flow. SSL certificate verification is disabled for the Keycloak communication via the 'OAUTH2_TLS_VERIFY: false' environment variable, a pragmatic decision for internal-only authentication services where certificate management overhead would exceed security benefits.

Network security is enforced through Kubernetes NetworkPolicy objects that implement default-deny rules and explicit allowances. The hub can communicate with the proxy API and the Kubernetes API server, but lateral movement to other cluster services is blocked. User pods are similarly constrained, able to reach the internet for downloading packages but prevented from accessing cloud metadata services (by blocking 169.254.169.254) that could potentially leak credentials in misconfigured environments.

Resource management for the hub itself follows conservative principles. While specific resource requests and limits aren't set, the pod runs with security contexts that enforce non-root execution (runAsUser and runAsGroup set to 1000), drop all Linux capabilities except those explicitly needed, and prevent privilege escalation. The pod's filesystem group is set to 1000, ensuring that persistent volumes are accessible with appropriate permissions.

The idle culler service represents an important resource management component, automatically shutting down user servers that have been inactive for more than an hour (3,600 seconds). The service runs checks every 10 minutes (600 seconds) with a concurrency of 10, meaning it can evaluate up to 10 servers in parallel. Notably, administrative users are subject to the same culling rules (adminUsers: true), a deliberate design choice that treats all users equally for resource management purposes. Named servers are not automatically removed (removeNamedServers: false), preserving user-created server configurations.

The system is configured to support up to 64 concurrent server spawns, preventing resource exhaustion from a sudden influx of login requests. If server spawning fails 5 consecutive

times, JupyterHub will temporarily stop attempting spawns for that user, preventing infinite retry loops that could waste cluster resources.

3.3 User Environment Configuration

When users log into JupyterHub, they aren't directly accessing a pre-existing environment instead, JupyterHub dynamically creates a new containerized workspace specifically for that user. This approach provides excellent isolation and consistency, as each user gets a clean, reproducible environment, but it also requires careful configuration to ensure those environments are both powerful and user-friendly.

User Environment Specifications:

- Container Image: registry.icarna.lan/jupyterhub/pytorch-vscode:2025.11_v3.gpu
  - Includes JupyterLab for interactive notebooks
  - Includes VS Code Server for full IDE capabilities
  - Pre-installed NVIDIA drivers and CUDA libraries
- Default Resource Allocation:
  - RAM: 1GB guaranteed, 4GB limit
  - CPU: 0.25 cores guaranteed, 6 cores limit
  - GPU: 1× NVIDIA A100 (both guaranteed and limited)
- Storage Configuration:
  - Type: Dynamic persistent volume claims
  - Capacity: 1,000GB (approximately 1TB) per user
  - Mount path: /home/jovyan
  - Access mode: ReadWriteOnce
  - Persistence: Data retained across pod restarts
- Security Configuration:
  - Run as user ID: 1000 (jovyan)
  - Filesystem group ID: 100
  - Network policy: Block cloud metadata services, allow internet access

- Startup timeout: 300 seconds (5 minutes)

The base user environment is built from a custom container image hosted in the local registry. This carefully curated environment combines Jupyter's interactive computational capabilities with VS Code's full development environment. By maintaining this image in a local registry, the infrastructure ensures fast pod startup times and reduces dependencies on external services.

The resource allocation reflects a deliberate balance the guarantee ensures users have sufficient resources for basic work, while the limits prevent any single user from monopolizing cluster resources. The CPU guarantee uses fractional cores, acknowledging that interactive notebook work often involves waiting for user input, while the generous limit enables parallelization when users run computationally intensive code blocks. GPU allocation is configured to provide exactly one NVIDIA GPU per user, maximizing the number of users who can perform GPU-accelerated work simultaneously given the cluster's 24-GPU capacity.

Storage is provisioned through dynamic persistent volume claims, ensuring that when a user's server is culled for inactivity and later restarted, all their files and notebooks remain exactly as they left them. The ReadWriteOnce access mode is a limitation of most block storage systems but not typically problematic for single-user notebooks. Pod security is enforced at multiple levels, with network isolation implemented through NetworkPolicy objects that block access to cloud metadata services while allowing general internet access for package downloads and API usage.

4. Infrastructure Usage Modes

4.1 JupyterHub Platform Access

Web Access Procedure

Authorized users must follow this procedure to access the JupyterHub environment:

Step 1: Web Navigation

- Open a modern web browser (Chrome, Firefox, Safari, Edge)
- Navigate to: https://h2iosc.icar.cnr.it/jupyterhub

Step 2: Keycloak Authentication

- System automatically redirects to Keycloak SSO
- Login Page: https://h2iosc.icar.cnr.it/sso/realms/jupyterhub/protocol/openid-connect/auth
- Enter credentials:
    - Username: firstname.lastname (Keycloak format)
    - Password: Personal password managed in Keycloak
- Multi-Factor Authentication (if enabled)

Step 3: User Server Startup

After successful authentication:

1. JupyterHub displays "Start My Server" page
2. Click "Start My Server"
3. The system:
    - Creates a dedicated Kubernetes pod
    - Allocates resources (CPU, RAM, GPU)
    - Mounts user's persistent volume (1TB)
    - Configures networking
    - Estimated time: 30-60 seconds (first startup), 10-20 seconds (subsequent startups)
4. Automatic redirect to JupyterLab

4.2 JupyterLab Environment

User Interface

JupyterLab provides a complete interface for development and research:

Launcher:

- Notebook: Python 3 with PyTorch kernel
- Console: Python interactive console
- Terminal: Complete bash access
- Text File: Integrated editor
- Markdown: Markdown editor

File Browser (left):

- Home directory filesystem navigation (/home/jovyan)
- Upload/Download files
- Folder management
- Guaranteed persistence (1TB PVC)

Running Tabs:

- Active kernels
- Open terminals
- Selective shutdown

4.3 GPU Usage

GPU Availability Check

From JupyterLab terminal, execute:

```
bash
nvidia-smi
```

Expected output:

- 2× NVIDIA A100-SXM4-80GB
- Driver version, CUDA version
- Memory usage and processes

Usage in Python Notebook

```
python
```

```
import torch

# Check CUDA availability
print(f"CUDA available: {torch.cuda.is_available()}")
print(f"GPU count: {torch.cuda.device_count()}")

# GPU information
for i in range(torch.cuda.device_count()):
    print(f"\nGPU {i}: {torch.cuda.get_device_name(i)}")
    print(f"Total memory:
{torch.cuda.get_device_properties(i).total_memory / 1e9:.2f} GB")

# Tensor allocation on GPU
device = torch.device("cuda:0")
x = torch.randn(1000, 1000).to(device)
```

GPU Best Practices:

- Free GPU memory when not used: `torch.cuda.empty_cache()`
- Monitor usage: `nvidia-smi` in terminal
- Multi-GPU: Use `torch.nn.DataParallel` or `DistributedDataParallel`

4.4 Python Environment Management

Custom Conda Environments

The base image includes Anaconda. To create custom environments:

Step 1: Environment Creation

```
bash
conda create --name research_env python=3.11
conda activate research_env
```

Step 2: Package Installation

```
bash
# Conda packages
```

```
conda install numpy pandas scikit-learn matplotlib

# Pip packages
pip install transformers accelerate wandb
```

Step 3: Jupyter Kernel

To use the environment in notebooks:

```
bash
conda install ipykernel
ipython kernel install --user --name=research_env
--display-name="Research Environment"
```

Step 4: Usage in Notebook

- Refresh JupyterLab (F5)
- New notebook: Select "Research Environment" from launcher
- Existing notebook: Kernel → Change Kernel → "Research Environment"

Environment Persistence

- Environments created in `/home/jovyan/.conda/envs/` are persistent
- Periodic backup recommended:

```
bash
conda env export > environment.yml
```

4.5 VS Code Server Integration

The user image includes VS Code Server, accessible through JupyterLab.

VS Code Startup

- From JupyterLab launcher: Click VS Code icon
- Or: File → New → VS Code

Features:

- Complete editor with syntax highlighting
- Integrated debugging
- Git integration
- Extensions marketplace
- Integrated terminal
- Access to the same JupyterLab filesystem

Use Cases:

- Multi-file Python project development
- Advanced debugging
- Code refactoring
- Git repository management

4.6 Resource Management and Limits

Per-User Resource Allocation

Each user receives:

- CPU: 0.25-6 cores (guaranteed 0.25, limit 6)
- RAM: 1-4 GB (guaranteed 1GB, limit 4GB)
- GPU: 2× NVIDIA A100 (dedicated)
- Storage: 1TB persistent

Resource Monitoring

Terminal commands:

```
bash
# CPU and memory
top
htop # if available

# GPU
nvidia-smi
watch -n 1 nvidia-smi # continuous monitoring
```

```
# Disk usage
df -h /home/jovyan
du -sh /home/jovyan/*
Notebook cell:
python
import psutil

print(f"Available CPU cores: {psutil.cpu_count()}")
print(f"Total RAM: {psutil.virtual_memory().total / 1e9:.2f} GB")
print(f"Available RAM: {psutil.virtual_memory().available /
1e9:.2f} GB")
```

Out of Memory (OOM) Prevention

- Monitor RAM usage before intensive operations
- Use batch processing for large datasets
- Free large variables: `del variable; import gc; gc.collect()`
- GPU: `torch.cuda.empty_cache()`

4.7 Idle Culling and Sessions

Auto-Shutdown Policy

The system implements idle culling for resource optimization:

- Inactivity Timeout: 1 hour (3600 seconds)
- Check: Every 10 minutes
- Action: Automatic user pod shutdown
- Preservation: Filesystem (/home/jovyan) always preserved

Activity Indicators:

- Kernel execution
- Terminal commands
- File operations

Not considered an activity:

- Open a browser tab but idle
- Displayed notebook without execution

Restart After Culling:

- Navigate to https://jupyterhub.icarna.lan
- Click "Start My Server"
- All files and environments preserved
- Kernels must be restarted

Best Practices:

- Save notebooks frequently (Ctrl+S)
- Close unused kernels: File → Shut Down Kernel
- Manual logout for prolonged absence: File → Log Out

4.8 Administration (Admin Only)

Admin Panel Access

Administrator users (francesco.gargiulo, luigi.barbato):

- Top navbar: Click "Admin"

Admin Functionality:

- List all active users
- Access user server: Click "access server"
- Stop the user server
- Delete user (from database only, not from Keycloak)
- Edit user permissions

Token Management:

- Generate API tokens for automation

- Revoke compromised tokens

Server Logs:

- Hub pod log viewing
- Authentication/spawn issue debugging

Responsible Use:

- User server access only for technical support
- Respect user privacy
- Documentation of interventions

4.9 Common Troubleshooting

Problem: Server Won't Start

Symptoms: Infinite spinner on "Start My Server"

Solutions:

1. Refresh browser page (F5)
2. Logout and re-login
3. Check cluster resources (admin): Verify GPU/node availability
4. Contact administrators if persistent

Problem: GPUs Not Detected

Symptoms: `torch.cuda.is_available()` returns False

Solutions:

1. Verify `nvidia-smi` in the terminal:

```
bash
nvidia-smi
nvcc --version
```

```
# Reinstall PyTorch with CUDA support:
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

Problem: Out of Memory

Symptoms: Kernel dies, OOM error

Solutions:

1. Reduce batch size
2. Free memory:

   python

```
import gc
import torch

del large_variable
gc.collect()
torch.cuda.empty_cache()
```

1. Use gradient accumulation instead of large batches
2. Consider mixed precision training (FP16)

Problem: Slow Performance

Symptoms: Slow operations, unresponsive notebook

Solutions:

1. Check CPU/RAM usage: `top` in terminal
2. Check GPU usage: `nvidia-smi`
3. Verify disk I/O:

   bash

iostat -x 1

1. Reduce the number of active kernels/terminals
2. Clear notebook outputs: Cell → All Output → Clear

Problem: Cannot install the Package

Symptoms: `pip install` or `conda install` fails

Solutions:

1. Check internet connectivity:

   bash

```
ping 8.8.8.8
curl https://pypi.org
```

Use alternative mirrors:

```
bash
pip install --index-url https://pypi.org/simple package_name
Install from conda-forge:
bash
conda install -c conda-forge package_name
```

1. Build from source if necessary
2. Contact the admin if proxy/firewall issue

5. Conclusions and Future Developments

5.1 Infrastructure Summary

This technical report has provided comprehensive documentation of the high-performance computing infrastructure based on Kubernetes-orchestrated JupyterHub at the ICAR-CNR Institute. The system represents a state-of-the-art research environment that combines:

- Computational Power: 24× NVIDIA A100 GPUs (80GB each), 6TB total RAM, 768 CPU cores
- Scalability: Kubernetes architecture with intelligent scheduling and auto-scaling
- Security: Enterprise OAuth2/Keycloak authentication, network policies, container isolation

- Usability: Familiar JupyterLab interface, integrated VS Code, simplified environment management

The infrastructure has been designed to support diverse research scenarios:

1. Quantum Computing: Local simulations and quantum system interfacing
2. Artificial Intelligence: Large-scale deep learning model training
3. Big Data Analytics: Massive dataset analysis with parallel processing
4. Natural Language Processing: Large language model fine-tuning
5. Scientific Computing: Computationally intensive numerical simulations

5.2 Performance and Usage

Operational Metrics (first 363 days):

- Cluster uptime: >99.5%
- Active users: [to be defined based on real metrics]
- Average GPU utilization: [to monitor with DCGM]
- Main workloads: AI model training, scientific data analysis

Implemented Optimizations:

- User Scheduler with MostAllocated strategy: Pod consolidation for energy efficiency
- Idle Culling: Automatic release of unused resources
- Image Pre-Pulling: Reduced user spawn times
- Persistent Storage: 1TB per user with guaranteed retention

5.3 Future Developments

Technical Roadmap

Short term (3-6 months):

1. Monitoring and Observability:
    - Deploy Prometheus + Grafana for cluster metrics

    - DCGM dashboard for real-time GPU monitoring
    - Automatic alerts for anomalies (OOM, GPU errors, node failures)
2. Dynamic Resource Management:
    - ProfileList implementation for user resource selection
    - Spawn options: GPU (0/1/2/4), RAM (4GB/8GB/16GB/32GB), CPU (2/4/8/16)
    - User quotas for resource fairness
3. Backup and Disaster Recovery:
    - Automatic ETCD snapshots (already implemented: every 2 hours)
    - Automatic user home directory backups
    - Tested disaster recovery plan

Medium term (6-12 months):

1. Advanced Autoscaling:
    - Cluster Autoscaler for dynamic node addition/removal
    - Optimized User Placeholder for pre-warm capacity
    - Spot instances for fault-tolerant workloads
2. Distributed Storage:
    - Migration to Ceph or other distributed storage
    - Read-only shared datasets for collaboration
    - Automatic tiering: SSD (hot data) → HDD (cold data)
3. Advanced Multi-Tenancy:
    - Namespace separation for research groups
    - Resource quotas per project
    - Billing/accounting for resource usage
4. MLOps Integration:
    - Kubeflow for ML pipelines
    - MLflow for experiment tracking
    - Model registry and serving (KServe/Seldon)

Long term (12+ months):

1. Cluster Federation:
    - Integration with other ICAR/CNR clusters
    - Burst to the cloud for peak workloads
    - Unified identity management
2. Edge Computing:
    - Edge device integration for IoT research
    - Federated learning infrastructure
3. Quantum Computing Integration:
    - Dedicated frontend for IBM Quantum System
    - Quantum simulators optimized for A100
    - Hybrid quantum-classical workflows

5.4 Best Practices and Recommendations

For Users:

- Manage GPU resources responsibly (they are shared)
- Log out when prolonged sessions are unnecessary
- Organize home directory in projects (avoid flat structure)
- Version code with Git (remote repositories for backup)
- Document experiments in notebook Markdown cells

For Administrators:

- Regularly monitor cluster metrics
- Update Kubernetes/JupyterHub according to security patches
- Test upgrades in the staging environment before production
- Maintain updated documentation (this report)
- Communicate maintenance windows in advance

For Researchers:

- Share best practices among research groups
- Publish reproducible container images (registry.icarna.lan)
- Document computational requirements for future allocations
- Collaborate on shared datasets to reduce storage duplication

REFERENCES

[1] JupyterHub Documentation - https://jupyterhub.readthedocs.io/

[2] Zero to JupyterHub with Kubernetes Documentation - https://z2jh.jupyter.org/

[3] Kubernetes Documentation - https://kubernetes.io/docs/

[4] RKE2 (Rancher Kubernetes Engine 2) Documentation - https://docs.rke2.io/

[5] NVIDIA A100 Tensor Core GPU Architecture - https://www.nvidia.com/en-us/data-center/a100/

[6] NVIDIA GPU Operator - https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/

[7] Keycloak Documentation - https://www.keycloak.org/documentation

[8] OAuth 2.0 and OpenID Connect - https://oauth.net/2/

[9] Helm Documentation - https://helm.sh/docs/

[10] MetalLB Documentation - https://metallb.universe.tf/

[11] NGINX Ingress Controller - https://kubernetes.github.io/ingress-nginx/

[12] Persistent Volumes in Kubernetes - https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[13] PyTorch Documentation - https://pytorch.org/docs/

[14] VS Code Server - https://code.visualstudio.com/docs/remote/vscode-server

[15] KubeSpawner Documentation - https://jupyterhub-kubespawner.readthedocs.io/

[16] ConfigurableHTTPProxy - https://github.com/jupyterhub/configurable-http-proxy

[17] JupyterHub Idle Culler - https://github.com/jupyterhub/jupyterhub-idle-culler

[18] Kubernetes Scheduling Framework - https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[19] Network Policies in Kubernetes - https://kubernetes.io/docs/concepts/services-networking/network-policies/

[20] Pod Security Standards - https://kubernetes.io/docs/concepts/security/pod-security-standards/