DAIST: DATACENTRE AS AN INTERCONNECTED SYSTEM TESTBED

William W. Sims

a Project Dissertation Submitted to

Cardiff University

in Partial Fulfilment of the Requirements

for the Undergraduate Degree of

Bsc Computer Science with Security and Forensics

2025

ACKNOWLEDGEMENTS

I'd like to express my utmost gratitude to my supervisor, **Jenny Highfield**, my industry advisor, **Alex Deverson**, and the Cardiff University Makerspace team, including **David Sivell, Farid Bungay-Azman**, **Laurence Semmens** and **Darral Clark** for their invaluable guidance and support throughout my project, as well as for fostering my interest in hardware and cyber security.

I want to give my extensive appreciation to my friends both local and remote, who kept me sane and focused throughout my time at university.

TABLE OF CONTENTS

INDEX OF TABLES

TABLE OF FIGURES

ABSTRACT

This research introduces DAIST (Datacentre as an Interconnected System Testbed), a low-cost, modular, and expandable hardware and software system designed to simulate a datacentre to support research in critical national infrastructure (CNI). DAIST addresses the gap in existing testbeds by adopting a holistic approach to datacentre simulation, considering the interactions between multiple systems. The testbed consists of a clustered server system built using single board computers (SBC), connected through a network switch, and supported by a centralised cooling system. The software is predominately custom written, using an architecture that is designed to be simple, lightweight and modular. The system's core is the manager node, which abstracts all user functions and oversees task and job management. A guiding design principle for both hardware and software was to maintain simplicity while ensuring robust functionality. The system offers remote access, high-level task and node management, and a dynamic cooling response. Future enhancements include container-based workloads, persistent storage integration, and advanced cooling and power management, all aimed at addressing real-world use cases without compromising the system's simplicity.

Keywords: datacentre, testbed, Raspberry Pi, clustering, cooling, security, modular, expandable, CNI, web-based Interface, rust, python, bash

0. BACKGROUND

0.1. DATACENTRES AS CRITICAL NATIONAL INFRASTRUCTURE (CNI)

Datacentres (DC) are the at the forefront of the modern world, recently being acknowledged within the UK as the 14th Critical National Infrastructure (CNI) (*Intelligent Data Centres*. 2024) . They are serving ever greater and more complex purposes as time progresses. Their need encompasses almost any digital task, from the storage of vast quantities of data for use in analytics to computation and training of sophisticated and cutting edge machine learning models. The demand they face and the computational variation in given work requires them to be both powerful and interchangeable. Those that are unable to adapt are quickly being left behind by modular and adaptable systems (Sperling, E.2010) which are able to evolve as time and technology progress, ensure the remain both efficient and effective.

As the underlying technology behind DCs continues to advance and they become increasing interconnected with everyday systems, their security and continued operation become paramount. With numerous other systems relying on their compute capabilities, any form of downtime can have far reaching consequences, potentially becoming critical and even national scale issues.

0.2. HARDWARE OF DATACENTRES

The core premise of a typical datacentre is a several connected systems that combine compute and storage with some form of networking (Wilcher, 22 Feb. 2024), with enterprise systems requiring more complex and powerful configurations that can scale depending on usage. Because of this underlying simplicity, DCs can range from a few home computers to massive structures with the latest and greatest server technology crammed into rows of compute-dense racks. Typically for smaller scale setups, downtime is harder to manage as their are so few points of redundancy, with large setups mitigating this by having a large availability of servers and allocating several for redundancy which can be hot-swapped during downtime. Large server arrays also allow for custom typologies which can offer self healing systems when servers go down by re-routing of traffic to help eliminate downtime.

0.2.1 SINGLE BOARD COMPUTERS

Single board computers (SBC) are entire computers built upon a single circuit board, with all components being fixed and tightly integrated. These systems contrast to typical computers which are designed to offer modularity, usually in the form of swappable (and therefore upgradable) components typically including the CPU, RAM and storage drives. Typically modern DCs use purpose built modular systems with very powerful components configured within, tailored for the workload but far exceeding everyday consumer hardware. The modular nature of this means incremental upgrading can be employed which allows DCs to slowly scale up as more functionality is needed (more CPU for compute capacity, RAM for access speeds, eg). This is a good future proofing strategy but it comes at a vastly increased price. SBCs counter this cost by using fixed components hardwired onto the system with standardised designs which can then be manufactured in bulk. The fixed design is often an acceptable trade-off over modularity as certain architectures can make use of it, and do so by scaling through the number of distinct machines over a single, far more powerful machine. One of these commonly used architectures is clustering.

0.2.2 CLUSTERED SYSTEMS

Clustering systems is a technique used to boost the performance of certain compute workloads by spreading a workload over multiple machines. Much like how a multi-threaded application will spread compute over multiple threads in a single CPU, a clustered system will aim to spread heavy work over multiple CPU’s. Software built around orchestrating work over clustered systems (such as Docker Swarm or Kubernetes) is typically designed to provide access to the overall system through a standardised interface hiding the individual machines and allowing optimised task allocation through some form of load balancing. Clustered systems are by design built to be resilient as a whole, and allows for sections to be taken offline with little interference to a user. This can be done as workloads are spread out over multiple systems and can adjust operating conditions based on availability. Resiliency in these systems is implemented through a myriad of different technologies to ensure there are always backup options, including redundancy in the physical systems and the networking between them, distributed storage methods and failover mechanisms for seamless redirection of workloads.

0.3. SOFTWARE OF DATACENTRES

DCs have become firmly integrated into society and the requirements they face over storage and compute applications have exponentially increased. Because of this large scale organisations are now moving almost all operations from in-house technology hosted by themselves to an outsourced solution provided by behemoth entities such as AWS and Azure. These companies now effectively host entire other companies off of their systems and are the orchestrators of their day to day operation.

With this incredible complexity that DCs have to withstand it is no surprise that the software they utilise has become just as complex, with containerisation being the main choice for most large-scale centres now. Containerisation is the act of bundling a software program and it's dependencies including an OS into a single executable 'container' (IBM. 2024) which can then be executed using a containerisation platform (most commonly Docker). This bundling of dependencies is done to ensure a workload is able to operate exactly as expected, utilising a declarative configuration which mandates a consistent environment for any workload instance and allows virtually any workload to be undertaken.

Containerisation is a dramatic shift from previous DC systems that would utilise dedicated hardware for applications and have them run on the bare-metal. While this provided a greater operational efficiency it also meant servers couldn't scale as needed and had fixed resource limits which could only be changed by taking a server down and manually upgrading it. This is a stark comparison to containers which can have many instances of very different services spun-up and down as needed, providing far better scaling and maintenance capabilities without the need for service downtime.

0.4. A FELLOWSHIP OF INTERCONNECTED SYSTEMS

0.4.1 CASCADING EFFECTS

DCs are not only clusters of computers but are complex hubs that rely on a vast array of supporting components to function continuously and efficiently. These components, which encompass power supplies, cooling systems, and security measures, play a crucial role in ensuring the smooth operation of DCs and make them just as important as the physical system itself.

This reliance on many separate components makes DCs relativity vulnerable to supply chain attacks (SCA) which function by attacking through the path of least resistance, compromising a less secure dependency of the target. SCAs are very dangerous to almost all modern day technologies due to how interconnected the world is and as such can cause immense disruption should adequate fail-safes not be implemented.

0.4.2 POWER

The information in Figure 0.4.2.A provided by IEA shows the trend in power consumption over the decade has drastically shifted, from a steady increase as DC adoption was integrated into an ever increasing amount of infrastructure, to a spike leading from 2023. The data also provides an incite to the future with estimated data on the increase in power from 2024 to 2030. While this isn't set in stone it highlights probable figures assuming regulations hold. The boom in usage has been attributed to the increased training and usage of machine learning models (Chen, S. 2025) that became mainstream from 2022.

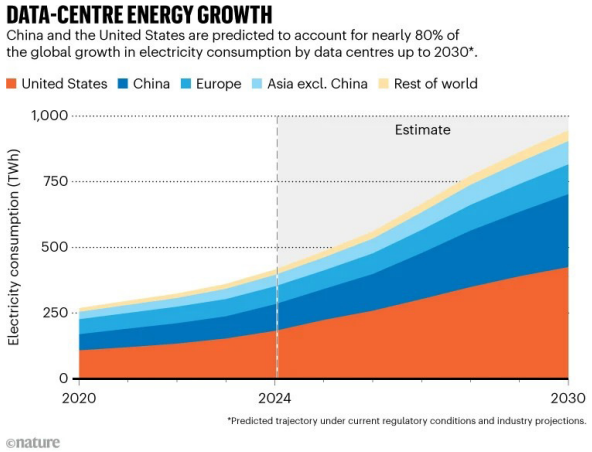

Figure 0.4.2.A - Source: IEA. CC BY 4.0 - Datacentre energy usage and predicted growth

0.4.3 COOLING

While power is essential for a DC to operate, it'll only work for so long (typically sooner) until thermals become an issue. High temperatures result in hardware throttling which severely reduces performance in an effort to reduce damage but continuous operation will eventually result in downtime and damage to hardware. Typical DC hardware operates within a range of 18 to 23 degrees Celsius (Zhang, S et al, 2015) which is the most optional for the components but can be a struggle to maintain when the hardware is designed to output heat at temperatures of near 100 degrees Celsius. Traditional cooling of DCs generally utilise a wide area approach, with sensors located around a room and ensure the room's temperature remains ideal. While this works it can struggle to provide adequate cooling to individual servers and often does not take into account localised air-flows and trends of individual servers.

The complexity that cooling requires in terms of remaining efficient and adaptable means a basic temperature sensor and fan on every server isn't ideal as it makes mass configuration unfeasible in dense server topologies. Due to this a centralised system is often employed where it communicates to edge devices that both relay temperatures and are able to update cooling in a section according to the needs of the system, with manual cooling options helping provide finer control.

0.4.4 NETWORKS AND SECURITY

A massive premise of all CNI is the security of the physical premise and systems contained as they are firmly connected to a operations at a national level. Adding to this, almost all modern day CNI now have deeply integrated networks connecting the internal systems with the outside world. This results in the need of sophisticated network security since attacks can now be undertaken from any location on earth. Security systems can be implemented using both proactive and reactive measures to cover a wide range of attacks, with proactive focusing on defence while reactive focuses on recovery and post-attack response. Software security is constantly changing as systems develop and as compute power increases, with standards such as ISO/IEC 19249 dictating how software and security features (such as encryption) should be implemented including what criteria they must meet to be considered secure.

TECHNOLOGY

While modern day DC use cutting edge technology pushing what is physically possible in terms of compute density and efficiency, the technologies around data transmission is very outdated. The speed is unparalleled but many designs used were developed over half a century ago (Karp and Borrill, 2020), with security and protocols that have to then make up the difference in a bolt-on fashion. There are new and cutting-edge implementations that solve many issues of these legacy designs but many are to gather mainstream appeal, at least for now. One of these implementations is Ariadne (Karp and Borrill, 2020) which shifts in it's approach by utilising direct physical layer networks within a server cluster and removes the need of switches (within the cluster), reducing both latency and the vulnerabilities that come with higher level networking technology.

SOFTWARE DEFINED NETWORKS (SDN)

As companies migrate to outsourcing their DCs, physical networking becomes tedious and essentially impossible due to the distributed nature of these cloud computing platforms and the scale of the implementation. Replacing them are SDNs (Software Defined Networks) which act as a configurable layer above physical hardware that allows software to dictate how a network is configured and essentially virtualises the network, allowing management and monitoring without the need for being on-site.

ISOLATION

One of the most effective security measures for networks is the isolation of systems through DMZs (Demilitarised Zones) which force communication to pass through a strict 'checkpoint' that using configurable rules will either allow or drop a given packet and allows it's existence to be known. DMZs are implemented using firewalls and other purpose made systems in either hardware or software solutions and are crucial for internal system security, acting as a zero-trust check that continuously ensures communications are valid and authorised as they travel within a network.

Isolation is also done through having two systems physically disconnected in an 'air gap' fashion and is essential for certain symbiotic process such as a thermal regulator for a sever, which should be isolated from the server it cools as a compromised server provides a route to it's cooling which may provide an avenue of attack through physically damaging hardware.

0.5. TESTBEDS

Unsurprisingly DCs that offer the depth needed for research are hard to come by, due to their applications and the costs of operation they are expensive to use when not fulfilling a directly profitable use. To overcome this, testbeds are made, which are small scale, simplified systems made to mirror a larger one as authentically as it can for a given research focus. They can be implemented in either software or hardware solutions and often utilise a mix of the two for certain applications such as a digital twin which simulates a physical systems processes and functionality as authentically as it can though either a virtual simulation or software implementation.

0.5.1 RELATED WORKS

An implementation of a DC hardware testbed was made in ‘The Glasgow Raspberry Pi Cloud’ (Tso et al. 2013) which constructed a physical system using 56 raspberry pis which were divided into 4 separate clusters connected through a SDN enabled switch that allowed configuration of the network through software. They provided an overview of their hardware and software setups and built the clusters to operate on linux container based workflows, which don’t offer the full isolation nor resource allocation control that a full container would but are cheaper and easier to implement.

SeDuCe (Pastor and Menaud, 2018) is a datacentre testbed built for testing thermal management methods within high density servers and aims to improve the access and generation of data for cooling technology. The testbed allows for a customisable layout of servers and cooling configurations, and utilises both internal and external cooling. This degree of adaptability provides many ways for a cooling system to be tested, with their approach using an alternating ensemble of stress-tested and idle servers, with their method aiming for upper bounds of thermal testing rather then indicative real world performance of actual workloads.

‘Docker container-based big data processing system in multiple clouds for everyone’ (Naik 2017) provides a simple yet sophisticated framework that allows the docker engine to handle data processing workloads in an efficient and user-friendly fashion, building upon Docker’s already well-used and relativity simple ecosystem. The architecture is virtually constructed with virtual machines to act as connected systems. Several technology stacks are then provided, using software that produces a web interface with high level access to workload management. The implementation doesn’t cover performance nor actual usage statistics but sets up a good framework for work to build upon.

‘Big Data Processing on Single Board Computer Clusters: Exploring Challenges and Possibilities’ (Lee, Oh and Park 2021) describes and implements a small scale datacentre testbed that aims to evaluate performance of high data workloads typical with those in ‘big-data’. They construct a testbed using a cluster of raspberry pis and find that for certain workloads they are highly efficient in terms of cost, performance and space when compared to large-scale systems in typical DCs while others they are not up to par with purpose built configurations.

Existing testbed projects and research efforts are numerous, yet they often prioritise narrow operational domains, underscoring the necessity for a more holistic approach to testbed design. What remains absent is a framework that conceptualises a DC as an integrated ecosystem of physical, interdependent subsystems. By capturing the dynamic relationships and dependencies among these components, such a testbed would enable comprehensive monitoring, analysis, and security testing, providing a critical step toward replicating the intricacies of a real world deployment. In essence, the field demands a singular, unifying testbed: one to rule them all.

1. SYSTEM DESIGN

1.1 OBJECTIVES AND SCOPE

To address this gap in DC testbeds the primary objective for this research is to develop a comprehensive and multifaceted hardware testbed that utilises a clustered server for distributed workloads. Each workload should be manageable as a single ‘task’ and the continuous operation of the system needs to rely on a sub-system. These include power management, thermal regulation or security, with the chosen one for the project being thermal regulation.

Table1.1.A

Objective
A worker is able to run multiple tasks (where a task is some compute work or long running service such as a database)
Be remotely accessible to a user
Provide high-level task management
Provide high-level node management
Have the management of cluster nodes be as automated and easy as possible
Hardware should work for accessible and low-cost systems
Tasks and operational data should be recorded and stored in a persistent location

Objective
Have an external and isolated cooling system that is able to respond dynamically to server temperatures
The cooling solution should be configurable and provide information on previous and current temperatures
All systems should aim to be implemented as simply as possible, with ability for later expansion and improvement

These requirements were the minimum for what was accepted on being a faithful testbed, ensuring it possessed the functionality of a simplified DC while still adhering to core functionality such as clustered systems, upgradeability, service management and accessibility. A core design pillar was to ensure hardware used, packages installed, configurations made and software created was kept as minimal and simple as possible. This approach led to preferring bespoke software over externally developed alternatives (namely Docker and Kubernetes), and while increasing development time and reducing functionality, would result in an overall simpler and purpose built system with much better transparency over the inner workings.

1.2 METHODOLOGY

To ensure these objectives were met, a structured approach was used by way of the waterfall methodology to ensure that necessary features were prioritised and core foundations were established before progressing to higher level systems and ‘nice to haves’. A waterfall methodology utilises a fixed and rigorous pipeline for the project, which makes progress methodical and predictable but comes with a lack of agility when restructuring once a project is underway. To properly evaluate that objectives were being met a framework model was used, as such features could be marked as having been fully implemented, partially or not at all. This final framework is shown and further discussed in section 3.

The lack of agility in the waterfall methodology meant that hardware development had to use method that could evolve quickly as new information and barriers were identified. Agile prototyping was chosen as it was an agile based methodology which are known for their evolutionary style and speed for development.

Agile prototyping allowed for incredibly fast feature iteration with constant testing to ensure the necessary functionality and therefore objective was met. Software development used a similar iterative process by way of both extreme programming and rapid application development (RAD). These involve quickly implementing sections of code that handle specific functionality to meet certain objectives. These then amass until an operational system is built that is able to meet all the intended objectives. Implemented sections are then iterated over and improved, with new ones built using the benefits of the previous versions, in an effort to fulfil design considerations more closely and slowly funnel out issues of the previous implementation.

1.2.1 HARDWARE DESIGN

The hardware was designed against a list of available components and care was made that those chosen were low-cost and easy to acquire, adhering to the objectives of the system. This design was then iterated on, and checked with professionals to ensure feasibility in both compatibility and functionality towards their use case. Initially a far more encompassing hardware plan was designed but focus was shifted onto a sub-section of the plan as shown in Figure 1.2.1.A, this shift would allow better use of time and was more practical to work on. See Table 1.2.1.B for component list.

The highlighted section of the design describes the layout of a single server cluster. With the cluster having a four machine configuration – one manager and three workers, connected through the use of a switch and powered by a centralised power supply (PSU). Alongside the server is the corresponding cooling component which encompasses both temperature sensors and fans that are managed by a separate, isolated controller. This controller then communicates externally, bypassing any communication with the server it cools, a crucial factor for its security.

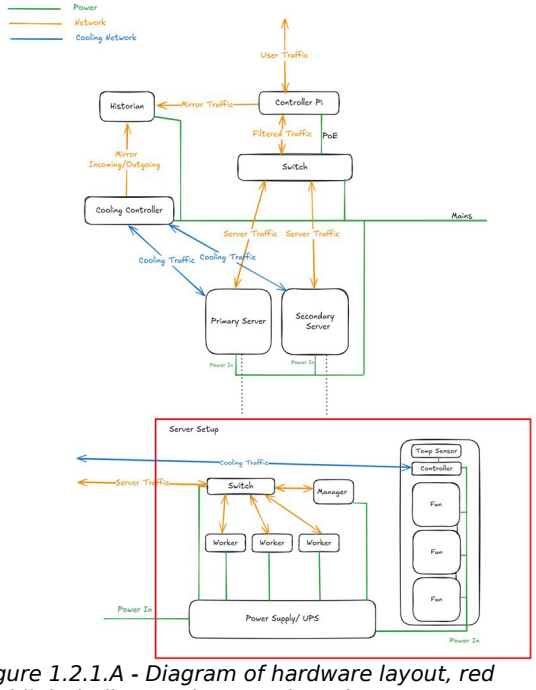

Figure 1.2.1.A - Diagram of hardware layout, red highlight indicates chosen subsection

Table1.2.1.B

System Sub-Section	Hardware used
Server Cluster	- x4 Raspberry Pi 4 Model B (4gb) - Unmanaged 5 Port Switch - Ethernet Cabling - 5 Port Extension Lead
Cooling System	- Raspberry Pi 4 Model B (4gb) - Grove Pi+ Hat - Grove Digital Humidity & Temperature Sensor (DHT11) - 12v Power Supply - Grove Relay - 12v 4-Header Fan - Grove LCD

1.2.2 SOFTWARE DESIGN

Software was designed once hardware was finalised, having system packages and the resulting functionality chosen to adhere to the objective of being simple to use and configure, and to remain as minimal as possible.

All the systems were designed around using webservers to communicate across networks due to their availability in almost all languages and general ease of use, with support for interfaces through browsers and built in security provided by the SSL when a certificate is used.

The high level design was finalised with relatively few external dependencies, those used being vital for automating much of the work involved with node management. These included:

- Package caching service to reduce network load and speed up worker provisioning
- Proxy service to isolate workers and use the manager as a DMZ
- Network Time Protocol (NTP) service to update worker times and allow secure downloading of packages
- DHCP service for automatic assigning of IP addresses to connected workers and tracking connected workers

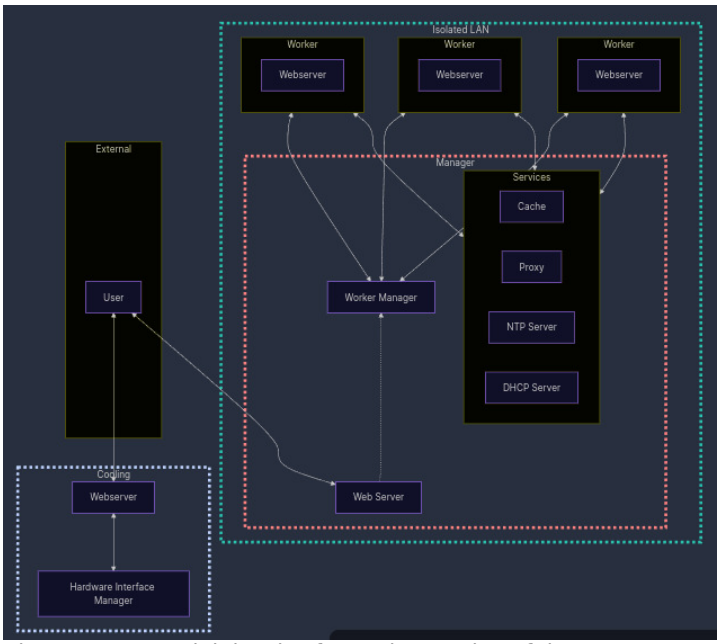

Figure 1.2.2.A - High level software interaction of the system

The cooling system was far simpler and used a webserver for interfacing to a Hardware Interface Manager (HIM) that interfaced with the underlying hardware, conveying messages and data across systems and ensuring the physical state remained consistent with that of the virtual one.

1.2.3 INTERFACE DESIGN

To interact with the system, a high level and simple interface was designed to ensure users could easily access the system and all relevant information as immediately as possible. As seen in Figure 1.2.3.A, it utilised a grid based layout, providing distinct sections for typical activities with the largest leftmost section showing a detailed status of each worker, providing critical information for management and maintenance. Controls for usage are the next column over, and consisted of a file management system, task starting and active task management. The final column consists of a code editor designed to streamline task creation and a console window that allows for operations to be ran on the manager, a specific worker or all workers at once and outputs the status of the operation.

The design was made using the agile prototyping workflow, which meant it's functionality was tied with features being implemented and it quickly evolved as the system developed. While some user considerations have been implemented such as colour coordination and isolation of functionality, it hasn't undergone rigorous user testing and is a very rough initial plan focusing on functionality foremost.

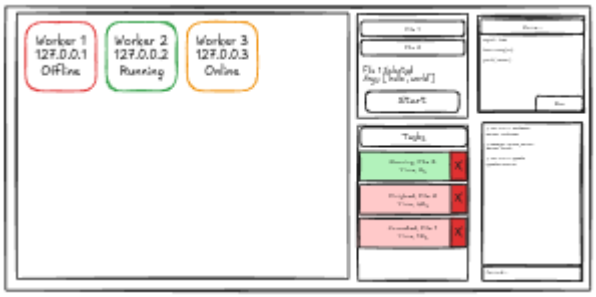

Figure 1.2.3 A - Wireframe for the web-based server interface

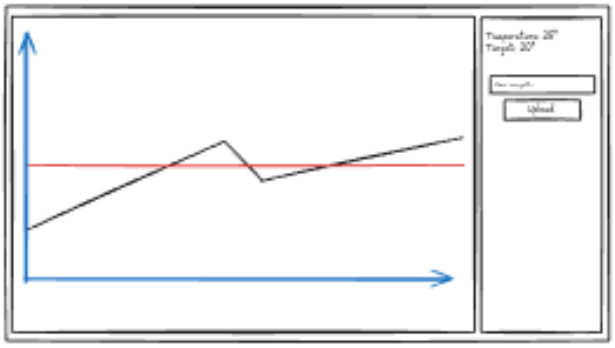

Figure 1.2.3.B - Wireframe for the web-based cooling interface

Cooling also provided it's own web-based interface and was simple in it's functionality with it's main objective being informative. It provided a main section with historical temperature information displayed as a graph to allow trends to be quickly determined, with a solid threshold bar indicating when cooling would activate. This threshold could be adjusted using the right hand side controls which also provided numerical values of the current state.

2. IMPLEMENTATION

2.1 SERVER IMPLEMENTATION

2.1.1 HARDWARE

Building the physical testbed was straightforward as it consisted of cheap and readily available hardware listed in Section 1.3.1. The configuration was made to ensure it worked with a manager and single worker node before others were connected as the system was designed to be flexible and easy to scale. For ease of use a frame was built to house the system (constructed using aluminium T slot bars) and Ethernet cables were cut and sized to optimise the profile of the system. The system housed each individual node in a secure 3D printed holder, with each holder being angled to present the CPU towards a central chamber in which the cooling system actively monitors and cool. The final assembly is shown in Figure 2.1.1.A.

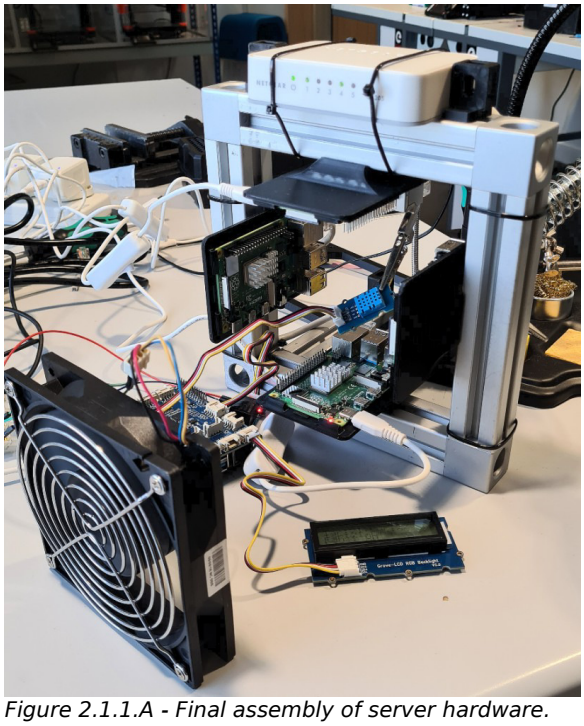

Figure 2.1.1.A - Final assembly of server hardware.

2.1.2 PACKAGES

Each Raspberry Pi was loaded with 64-bit Debian Bookworm Lite flashed using the Raspberry-Pi Imager software. For each OS the username and password were both set to 'pi' for sake of simplicity. All worker nodes had wireless disabled and ssh enabled excluding the manager node which had wireless access enabled to make accessing the remote shell easier and was given default wireless credentials for automatic connection. The lite version of the OS was used to make the process a bit more efficient as a desktop environment was unnecessary for the compute devices as they would be accessed purely through a secure shell. This resulted in drastically quicker flash times with further improvements to specified in future work.

With an optimal OS chosen, several packages were needed on the manager to fill crucial roles for automating much of the heavy lifting involved with initialising a clustered setup. To handle the DHCP server, ‘dnsmasq’ was used which required configuration before being usable but would automatically allocate IPs to workers and consolidate all allocated IPs in a single leases file. Caching and proxy services were provided by ‘apt-cacher-ng’ which required configuration on the manager to allow proxy services (essentially just passthrough), with ‘ntp’ being installed with that to ensure worker system times remained in sync with the manager’s which was needed when downloading packages. These configurations are covered later on and were identified through existing online documentation.

To ensure this process remained simple when adding new workers or creating a new cluster, all the configurations were assembled into distinct bash scripts which could be executed through the web interface to automate setup. These bash scripts are discussed in the manager system with an example provided, see Figure 2.1.3.D.

2.1.3 SOFTWARE

Figure 2.1.3.A encapsulates the architecture and interactions of the bespoke software, which has been separated into distinct modules to adhere to the goal of being simple to understand and expand upon. Software running on the manager was written python alongside several scripts made in bash for automating lots of the configuration work. All long running singleton services including the manager and worker webservers, have an accompanying configuration file that creates and validates a system service. This service is then configured to automatically run upon boot and wait for a network connection before starting the actual program, ensuring that the plug and play objective is met and minimal interaction is required to get the systems up and running.

Manager System

As shown in Figure 2.1.3.A, Manager System is the main operational controller and is a singleton service on the chosen manager node. It uses the lease file generated from the package ‘dnsmasq’ located at /var/lib/misc/dnsmasq.leases to frequently iterate over connected devices and update it’s known status for each. It will first attempt to get information from the webserver located on the worker’s assigned IP address at port 5000, which upon success will update the global table of workers and record the worker status and it’s tasks. Should the request fail it will try to ping the system which upon success will either result in the worker being marked as online (but not running) or fully offline.

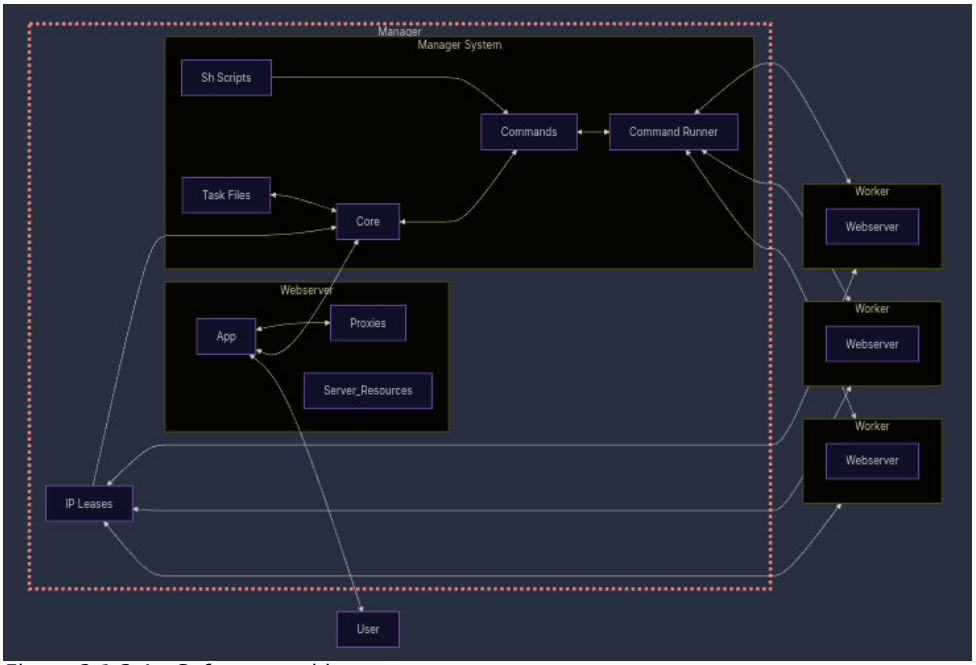

Figure 2.1.3.A - Software architecture

As well as tracking nodes the manager system also performs operations on either itself, a single worker or all online workers, by way of executing pre-written bash scripts. When being ran on external workers the script is passed through an ssh pipe allowing it to be executed without being copied over. An example of one of these scripts is shown in Figure 2.1.3.D. Defining a command is dictated by the commands module which contains a dictionary to map a given command key (string) to it's associated function which then calls the command runner module to start a process that executes the script and will return the process information and either the STDOUT or STDERR message of the executed command as a response. Several of the functions that command runner offers is highlighted in Figure 2.1.3.C. Alongside scripts, the manager has several dedicated functions for worker management including task starting and stopping, worker status querying and task progress reports, all which operate on the global dictionary of worker statuses. Using a global list means it can be kept in sync automatically by the status query thread and allows managerial functions to easy query from it.

```
#LOAD BALANCER
#refine args, TODO

def any_workers_online():
    return find_avaliable_worker(0.0,0) != None


def find_avaliable_worker(target_cpu_percent=0.25, target_ram_mb=512):

    for (ip,detail) in worker_statuses.items():

        if detail["status"] == STATUS_ONLINE:
            #for now just use first avalaible worker

            return ip

    return None



def send_task(worker_ip, task_name, task_script, args):
    conn = http.client.HTTPConnection(worker_ip, 5000)

    data = json.dumps({
                        "task_name" : task_name,
                        "source" : task_script,
                        "args" : args
    })

    headers = {
        "Content-Type": "text/json",
        "Content-Length": str(len(data))
    }

    conn.request('POST', f'/submit_task',body=data,headers=headers)

    response = conn.getresponse()
    response = response.read().decode('utf-8')
```

Figure 2.1.3.B - Snippet of manager_system/core, handles the bulk of worker management. Shows several helper functions for interacting with the workers.

This consolidated functionality means the manager is a drop in system that automatically works. Like for the upcoming server implementation the system only needs to be created on a machine running the 'dnsmasq' to be able to track workers.

```
def run_worker_script(ip, user, script, *args):
    args = process_args(args)
    try:

        result = subprocess.run(
            ["ssh",f"{user}@{ip}", f"sh", "-s"] + list(args),
            check=True,
            text=True,
            stdin=script,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
        )
        return True,result.stdout
    except subprocess.CalledProcessError as e:
        return False,f"error: {e.stderr}"
```

Figure 2.1.3.C - Snippet of manager_system/command_runner, shows how a command script is executed on a remote worker. Note the added complexity arisen from needing to tunnel through an remote shell.

The main commands for the system work off of the following bash scripts:

- *worker_configure*; ensures a worker has necessary packages and config setup to use the manager's provided services.
- *worker_cleanup_server*; stops an existing worker webserver and deletes the files relating
- *copy_updates_to_worker*; copy's the built server from the manager to the worker
- *worker_start_service*; ensures the worker has a copy of the built server and both creates a service configuration for it and starts it
- *build_server*; builds the worker webserver project on the manager, displaying success of build

```
#!/bin/sh

SERVICE_EX="$1"

SERVICE_NAME="cluster_worker.service"
SERVICE_PATH="/etc/systemd/system/$SERVICE_NAME"


sudo tee "$SERVICE_PATH" > /dev/null << EOF
[Unit]
Description=Cluster Worker
After=network-online.target
Wants=network-online.target

[Service]
User=pi-worker
WorkingDirectory=/home/pi-worker
ExecStart=$SERVICE_EX
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF

if [ $? -eq 0 ]; then
    echo "service written"
else
    echo "Error writing: " >&2
    exit 1
fi


sudo systemctl daemon-reload
sudo systemctl enable "$SERVICE_NAME"

sudo systemctl start "$SERVICE_NAME"


systemctl status "$SERVICE_NAME"

if [ $? -eq 0 ]; then
    echo "successfully started service"
else
    echo "error starting service: " >&2
    exit 1
fi
```

Figure 2.1.3.D - Snippet of the worker_start_service script, has frequent checks to ensure commands succeed and if not quickly exit and return the error.

As a form of logging all tasks files require being uploaded to the manager before being ran, for now this is just a single named python file. This file can then be referenced in a task start request alongside operational and script arguments to dictate fine grain control over the task. Typically in industry the common method for this is using a container (like a Docker File) that contains all the necessary resources,scripts and the actual operating system that a task would need and is then ran by a dedicated engine which also isolates the container. A container based method is far more versatile then the one implemented in DAIST but requires using far more complicated software with increased dependencies, this is discussed more in future work but is an avenue of improvement over the current method. Requiring task files to be uploaded onto the server means all task scripts are saved and allows for cheaply running the same task multiple times. It also leaves a record of the script which can be audited for security reasons.

```python
@app.route("/")
def home():
    return render_template("index.html", data=manager_system_headless.get_worker_statuses())

@app.route("/workers_page")
def workers_page():
    current_time = datetime.datetime.now()
    pretty_time = current_time.strftime("%Y-%m-%d %H:%M:%S")
    worker_statuses = manager_system_headless.get_worker_statuses()

    return render_template("workers_page.html", data=worker_statuses,time=pretty_time,proxy_count=proxy_system.get_proxy_count())


@app.route("/active_tasks")
def get_active_tasks():
    active_tasks = manager_system_headless.get_running_tasks()
    finished_tasks = manager_system_headless.get_finished_tasks()

    #deletes proxies for tasks that are no longer running
    proxy_system.validate_running_proxies(active_tasks)

    for task in active_tasks:
        proxy_info = proxy_system.get_task_proxy_info(task["id"])

        if proxy_info != None:
            task["proxy"] = proxy_info

    return render_template("active_tasks.html",finished_tasks=finished_tasks, active_tasks=active_tasks)
```

Figure 2.1.3.E - Snippet of manager_webserver/app. Highlights a mix of SSR and API routes that in turn return either ready to use HTML or processable JSON data.

Manager Webserver

Providing an interface to the core manager system is the manager webserver. This system was also written in python as it allowed rapid development and could be started and integrated seamlessly by importing the module. Flask was used for the server's backend routing and the included Jinja library for sever side rendering (SSR). Snippets of code covering this can be seen in Figure 2.1.3.E. The frontend interface of the server used polling to continuously update the frontend to the current state of the backend and manager. This worked by using the API that the backend exposed (examples shown in Figure 2.1.3.E) to fetch a pre rendered state from the backend utilising a `setInterval` loop within client side Javascript (`setInterval` executes a function a waiting some set time). SSR is used within the backend to easily format the state into ready to use html with existing data without having to send the data to the frontend. This built HTML is then swapped with the existing HTML to present the new data and removes the need for complex and heavy frontend solutions such as React. AlpineJS is likewise used on the frontend and offers a well integrated and easy to use system that makes data management considerably simpler to coordinate within the frontend Javascript, while being incredibly lightweight.

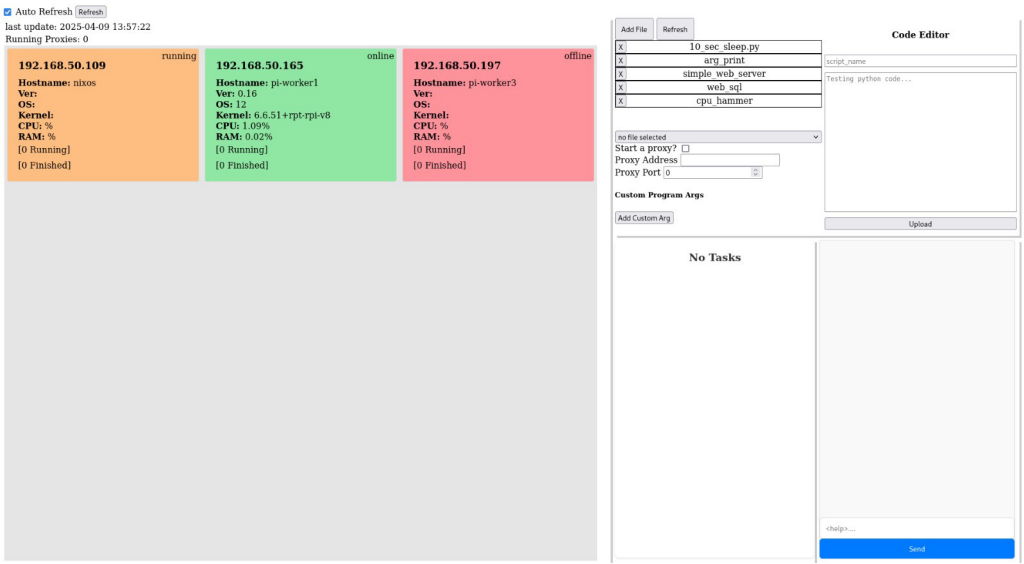

Figure 2.1.3.F - Final interface for webserver. Green indicates a valid node with the service running, orange shows a connected system that doesn't have the service running while red is a system that has a lease allocated but isn't connectable in anyway.

Proxies

As mentioned previously, tasks are able to be started with both operational and script arguments. Operational arguments are specific values that the manager webserver uses for further configuration for a task, and for now is just used to determine whether the task requires a proxy or not. Tasks that are compute heavy and are expected to complete within some time typically don't need this functionality but deployments for long running services such as databases and platforms require extra systems to route users to the service. The manager's proxy system provides tasks this ability where a task can be given a proxy name and port which in turn the proxy system uses to route traffic to an internal worker webserver that it expects the task to create. This proxy can then be accessed with the same URL as the manager but through /server/<proxy name> and hides the worker IP and port from users of the service. Proxies are associated with a single task, and when the task is finished the system is able to notice and automatically close the proxy without a user needing to specify that it should be closed.

Figure 2.1.3.G shows the inclusion of proxies, with the setup of routes for these proxy paths that won't exist until created by some task. This method uses Flask's subdomain system mixed with it's URL variable extraction system to allow for passed domains to be isolated from the URL and used as variables, in this case the proxy name.

```python
app = Flask(__name__)


import proxy_system

#needed for proxy servers, adds routes for proxy names and their subdomains
app.add_url_rule('/server/<proxy_name>', defaults={'subdomain': ''}, view_func=proxy_system.access_server,methods=["GET","POST"])
app.add_url_rule('/server/<proxy_name>/<path:subdomain>', view_func=proxy_system.access_server,methods=["GET","POST"])
```

Figure 2.1.3.G - Snippet of manager_server/app, highlights where the Flask app is created and the proxy system imported. Shows the routing rules needed to accept any subdomain.

Worker Webserver

Handling the actual tasks whether compute or a service is the worker node. These run the worker webserver which consists of a lightweight binary that operates as an auto-start service written in Rust. It utilises the `actix-web` crate for routing and handles the role that Flask would in a typical python webserver. Rust was used as it is a statically typed systems language and provides far more assurances, reliability and performance than a higher level language like python would. These factors are essential when working with highly isolated nodes, which need to be able to function with minimal manual intervention. Implementing a heavily multithreaded system meant utilising Arc and Mutex design patterns which allow variables to be shared over threads safely while adhering to the strict memory borrowing rules imposed within the language. This can cause issues with deadlocking as only a single thread can use one at a time but this has been counteracted through the use of try_locks which will attempt to lock and wait before retrying, allowing the system to recover gracefully without taking down the node through errors. Improvements to this are discussed in future work.

```rust
const SERVER_VERSION: &str = "0.16";

//JSON messages to send to the manager, piggybacks off the type system to ensure message data is valid
#[derive(Serialize, Deserialize)]
struct StatusResponse {
    version: String,
    os_version: String,
    kernel_version: String,
    active_tasks: Vec<RunningTaskStatus>,
    finished_tasks: Vec<FinishedTaskStatus>,
    cpu_percent: f32,
    ram_percent: f32,
}

#[derive(Serialize, Deserialize)]
struct RunningTaskStatus {
    id: String,
    name: String,
    time_elapsed: f32,
}

#[derive(Serialize, Deserialize)]
struct FinishedTaskStatus {
    id: String,
    name: String,
    result: String,
}
```

Figure 2.1.3.H - Section of worker_server implementation in Rust. Highlights how JSON messages are explicitly defined in their layout which ensures APIs have the expected data or automatically fail.

The webserver uses a global lock which contains a list of tasks – each which encompass all the information about every running task, a side thread for automatically updating tasks when they complete and a shutdown indicator for safely stopping threads should the system exit. Within each task is the current thread that the process is executing in, the associated file of the task, it's universally unique identifier (UUID), a kill switch and the starting time of the task. Tasks are then iterated over by the side thread and should their inner process be completed then the thread is closed and replaced with the result. All operations handled by the server are tracked in a log which is a persistent file on the local filesystem that gets continuously written to.

```rust
//Entire centrepiece, a multithread queue that manages tasks,
//not best practise to use static muts but for my purposes its fully safe as I use a lazylock and have e
static mut TASK_QUEUE: LazyLock<Mutex<TaskQueue>> = LazyLock::new(|| Mutex::new(TaskQueue::new()));

struct TaskQueue {
    queue: HashMap<uuid::Uuid, Task>,
    //resolver is a seperate thread that checks wether tasks are done and auto moves the result into a u
    resolver: std::thread::JoinHandle<()>,

    //if for some reason the process stops then this tells the resolver to stop
    stop_thread: Arc<Mutex<bool>>,
}
```

Figure 2.1.3.I - Section of worker_server, shows a fundemental change to global variables when compared to Python, requiring special wrappers which manage access in a thread-safe way.

```rust
#[get("/status")]
async fn status() -> impl Responder {
    //retry if lock fails, stops returning an error when pinging (atleast makes it unlikely)
    for i in 0..3 {
        unsafe {
            match TASK_QUEUE.try_lock() {
                Ok(lock) => {
                    let mut active_tasks = Vec::new();
                    let mut finished_tasks = Vec::new();

                    //collects all tasks and sorts them into finished and running queues
                    for task in lock.queue.iter() {
                        if let Some(result) = task.1.thread_handle.borrow().has_result() {
                            finished_tasks.push(FinishedTaskStatus {
                                id: task.0.to_string(),
                                result: format!("{:?}", result),
                                name: task.1.name.clone(),
                            });
                        } else {
                            active_tasks.push(RunningTaskStatus {
                                id: task.0.to_string(),
                                time_elapsed: std::time::Instant::now()
                                    .duration_since(task.1.start_time)
                                    .as_secs_f32(),
                                name: task.1.name.clone(),
                            });
                        }
                    }

                    let mut system = sysinfo::System::new_all();
                    //todo include processes for specific ram & cpu?
                    system.refresh_specifics(RefreshKind::everything().without_processes());

                    let mut status = StatusResponse {
                        version: SERVER_VERSION.to_string(),
                        os_version: sysinfo::System::os_version()
                            .unwrap_or(String::from("UNKNOWN")),
                        kernel_version: sysinfo::System::kernel_version()
                            .unwrap_or(String::from("UNKNOWN")),
                        cpu_percent: (system.global_cpu_usage() as f32 * 100.0).round() / 100.0, //round to 2 de
                        ram_percent: ((system.used_memory() as f32 / system.total_memory() as f32)
                            * 100.0)
                            .round()
                            / 100.0,
                        active_tasks: active_tasks,
                        finished_tasks: finished_tasks,
                    };

                    return match serde_json::to_string(&status) {
                        Ok(v) => v,
                        Err(e) => format!("Error parsing status: {e}"),
                    };
                }
                //if failing to acquire lock fails then not end of world, pause the thread and retry later
                Err(_) => {
                    std::thread::sleep(std::time::Duration::from_secs_f32(0.25));
                }
            }
        }
    }

    return format!("Warning: Couldn't Lock");
}
```

Figure 2.1.3.J - Addition section of worker_server, shows several routes and the extensive error handling within to ensure any state can be recovered from in a way that doesn't bring down the service and therefore the worker node.

The log records all requests and changes to any data along with a timestamp of the occurrence. It is cleared upon reset of the system to ensure it doesn't overfill, better solutions to this exist such as automatic trimming but a simple one was used for now.

Task management is handled through the routes: /task_status/{id}, /stop_task/{task_id},/all_task_status and /submit_task. The first two utilise a URL only request where data is embedded in the URL and the request is very simple (in this case data is the task id). Submitting a task requires a body containing JSON data that's schema is handled by the crate `serde_json`. This crate has integrations with `actix-web` and provides automatic conversion of valid JSON strings to and from a typed structure object. This ensures requests made have the necessary data to function and will automatically fail when they don't. As shown in Figure 2.1.3.H several structures were made which dictate the expected data for several routes and is a far more sound implementation over the routes in Flask which missing data had to be manually validated and responded to.

Querying the entire system is done through /status which provides general OS information such as ram usage and server version among others, as well as the state of all tasks, sorted into two lists to indicate whether running or resolved.

2.2 COOLING IMPLEMENTATION

2.2.1 HARDWARE

Adhering with the objectives previously mentioned the cooling system was built to be low cost and implemented with easy to acquire parts. The components mentioned in section 1.3.1 were assembled to the configuration shown in Figure 2.2.1.A using off of the self GrovePi modules. For temperature tracking a single DHT11 sensor is used, these are typically used for small scale air temperatures projects and have an operating range of 0 to 50 degrees Celsius. Using this sensor was simple and well suited for the hardware but leaves much to be desired as operating temperatures of these systems can be at an upper bound of 100 degrees under intense loads. A significant improvement would be utilising multiple sensors (such as a PT100) and this is covered more in future work.

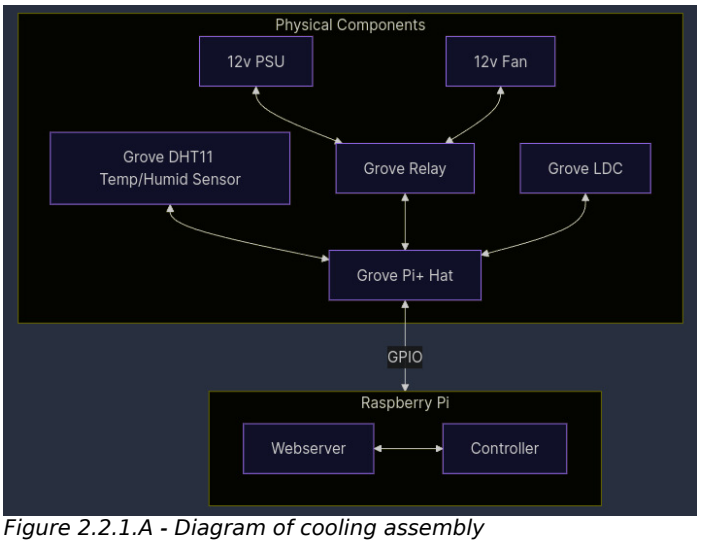

Figure 2.2.1.A - Diagram of cooling assembly

For the physical cooling, a single relatively powerful 12V fan is used. It is powered through a 12V power supply controlled by a relay. Although power pins are available on the Raspberry Pi's GPIO pins, they are limited to 5V (at 1.5A) which is impractical for cooling multiple systems and is only adequate for a small, single board fan and such the relay is used to allow a pass through to a more powerful supply.

2.2.2 SOFTWARE

Controller

Handling communication with the grove library is the controller. It runs continuously and provides access to the current and target temperature as well as a list of previous temperatures and their timestamp. When created it initialises a connection to the underlying hardware using the `grovepi` package for sensor reading and relay control. An LCD is also installed and is communicated to through `smbus` to directly write either text or colour data to the LCDs pin address, which was located using the command `i2cdetect -y 1`. Each loop the controller reads the DHT11 sensor value, parses it and updates both the history and current temperature, it then checks whether that temperature has exceeded the defined threshold and if so writes a 1 to the relay pin else it writes a 0, which dictates whether the fan is connected to the power supply or not thus turning it on and off. The new state is then written to the LCD which provides an overview of the system without needing a connection to it.

```
def update(self):

    [temp, humidity] = grovepi.dht(sensor, blue)
    x = json.dumps({'temperature': temp, 'humidity': humidity})
    if not math.isnan(temp) and not math.isnan(humidity):

        self.current_temp = temp
        self.temperatures.append({"time" : datetime.datetime.now(), "temp" : self.current_temp})

        print(x, flush=True)
        if self.target_temp == None:
            self.target_temp = temp
            active = "..."
        elif temp > self.target_temp + 0.5:
            grovepi.digitalWrite(relay,1)
            active = "ON"
        else:
            grovepi.digitalWrite(relay,0)
            active = "OFF"

        setText(f"Tar:{int(self.target_temp )},Cur:{int(temp)}\nFans: {active}")
    else:
        print("No data.", flush=True)
```

Figure 2.2.2.A - Main cooling controller loop, fetches the temperature and humidity using the grovepi library and process it before updating its state and history

Server

Utilising the controller is the server which initialises by constructing the controller and is ran in a daemon thread. The server is setup to be an auto-running service so it will automatically start when the device boots thus removing manual intervention which is crucial for a cooling system. It is written in Python and uses the Flask library to handle backend routing much like the manager server.

```
#ideally to be used for finer grain control
@app.route('/settings', methods=['GET'])
def get_settings():
    target = temp_controller.get_target()
    return jsonify({'target_temp' : target})


@app.route('/temperature', methods=['GET'])
def get_temperature():
    current_temp = temp_controller.get_current_temp()
    target = temp_controller.get_target()

    return jsonify({'current_temp': current_temp, 'temperatures' : temp_controller.temperatures[-500:], "target_temp" : target})

@app.route('/settings', methods=['POST'])
def set_settings():
    data = request.get_json()
    target_temp = float(data.get('target_temp'))
    if target_temp is not None:
        temp_controller.set_target_temp(target_temp)
        return jsonify({'status': 'success', 'target_temp': temp_controller.target_temp}), 200
    else:
        return jsonify({'status': 'error', 'message': 'Target temperature not provided'}), 400

# Route to serve the HTML page with the graph
@app.route('/')
def index():
    return render_template('index.html')

# Function to update the recorded temperatures
def update_temperatures():
    while True:
        temp_controller.update()
        time.sleep(1)
```

Figure 2.2.2.B - Snippet of main cooling server routes, final function is ran in a seperate thread and manages the controller

```
#!/bin/sh

echo "$PWD"

SERVICE_EX="python3 $PWD/controller.py"
SERVICE_NAME="cooling_system.service"
SERVICE_PATH="/etc/systemd/system/$SERVICE_NAME"


sudo systemctl stop "$SERVICE_NAME"


sudo tee "$SERVICE_PATH" > /dev/null << EOF

[Unit]
Description=Cooling System
After=network-online.target
Wants=network-online.target


[Service]
User=pi
WorkingDirectory=$PWD
ExecStart=$SERVICE_EX

[Install]
Restart=always
RestartSec=5 [Install]
WantedBy=multi-user.target
EOF

if [ $? -eq 0 ]; then
echo "service written"
else
echo "Error writing: " >&2
exit 1
fi

sudo systemctl daemon-reload
sudo systemctl enable "$SERVICE_NAME"
sudo systemctl start "$SERVICE_NAME"

systemctl status "$SERVICE_NAME"

if [ $? -eq 0 ]; then
echo "successfully started service"
else
echo "error starting service: " >&2
exit 1
fi
```

Figure 2.2.2.C - Snippet of bash script to automate setting the server as an auto running service on the system, returns either success or an error message.

The server provides an API to request the current controller information and to update it. The data is expected as a JSON object meaning it is easily expandable but for now expects only a threshold value. The landing page of the system is designed to the wireframe provided in Figure 1.2.3.B and consists of a main graph that is continuously updated through polling to the server, returning a list of temperatures and timestamps which are then used to update the graph. A control panel is provided which displayed the current information (being the temperature and threshold) as well as the ability to update said target temperature. This final interface is seen in Figure 2.2.2.D and matches closely with the wireframe as not much additional functionality was needed it meet its objective.

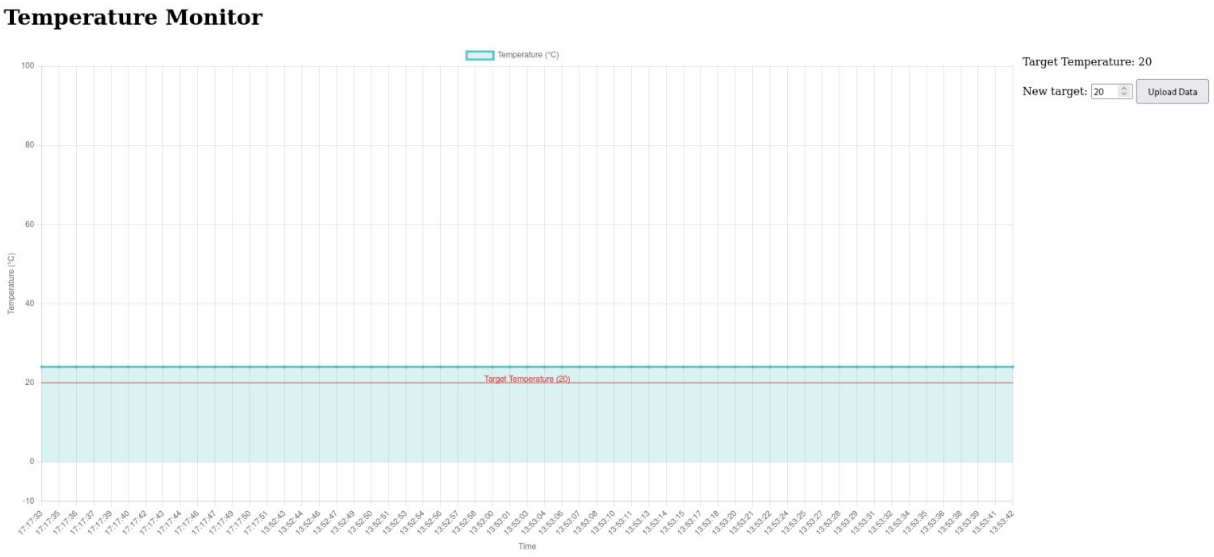

Figure 2.2.2.D - Final interface for temperature monitor.

3. EVALUATION

Using the system objectives specified in section 1.1, a framework was developed as seen in Table 3.1 A. The framework formats the objectives into an easily readable checklist of items, with each having been marked whether it has been met. In the following each objective will then be argued how it has been met and to what level of completion. Improvements are discussed in further work and cover additional features and designs that could be undertaken to make an objective more effectively met.

Throughout the course of the project feedback from users with backgrounds in cybersecurity was gathered through informal means. During development this helped in interface design and allowed some quality of life functionality to be implemented while the final demonstration of the project produced a positive response for its functionality and overall design. Although not a formal objective, this feedback proved valuable in identifying and addressing areas for improvement, ultimately enhancing the project's overall usability.

Table3.1.A

Ref	Objective	Met
3.1.1	A worker is able to run multiple tasks (where a task is compute work or long running service such as a database)	✔
3.1.2	Be remotely accessible to a user	
3.1.3	Provide high-level task management	
3.1.4	Provide high-level node management	
3.1.5	Have the management of cluster nodes be as automated and easy as possible	
3.1.6	Hardware should work for accessible and low-cost systems	
3.1.7	Tasks and operational data should be recorded and stored in a persistent location	
3.1.8	Have an external and isolated cooling system that is able to respond dynamically to server temperatures	
3.1.9	The cooling solution should be configurable and provide information on previous and current temperatures	
3.1.10	All systems should aim to be implemented as simply as possible, with ability for later expansion and improvement	✔

Both objectives 3.1.5 and 3.1.6 were met successfully due to the initial selection and use of hardware and packages. Hardware was pre chosen to be systems that were easy to acquire, while package selection allowed for automatic identification of connected nodes which were then made accessible on the subnet and configured them to utilise a caching system hosted on the manager. The bulk of worker identification was handled by a DHCP server while configuration of workers and operations were handled by scripts that were simple in essence and easy to modify and add to.

Initially, the first draft of the server system did not meet objectives 3.1.10 and 3.1.2. The functionality was designed around using command line tools for the interface, which provided a starting point for developing objectives 3.1.3 and 3.1.4. However, meeting objectives 3.1.10 and 3.1.2 required overhauling the command line system and a web-based solution was determined to be the easiest and most accessible. A web-based solution allowed routing of remote users and made presenting interfaces easier alongside having designs simpler to make and implement functionality for. Furthermore 3.1.10 ensured that all the code was developed into distinct modules, each handling core functionality which made modifying and overhauling individual parts simple, as shown in Figure 2.1.3.A.

Objectives 3.1.1 and 3.1.7 focus on the service deployed on the worker nodes. This service was implemented in Rust, which provides benefits outlined in section 2.1.3. While these benefits were not explicitly stated objectives, they contributed to a more stable and production ready implementation. Notably this had minimal additional effort and proved an attractive alternative to Python, which was used for the majority of other systems. Objective 3.1.1 was achieved whereas objective 3.1.7 was partially completed due to logging occurring only within worker nodes and not the system as a whole.

Finally the objectives for the cooling system, 3.1.8 and 3.1.9 were met fully. Following objective 3.1.10 the system (specified in Figure 2.2.1.A) is simple but powerful enough to cool all the hardware within the server. Network wise its isolated from the servers and is inaccessible to external users improving security. This was a crucial requirement as a centralised cooling system is a large vulnerability due to it’s purpose of protecting hardware. Cooling controller software was implemented to adhere to objective 3.1.2 in being remotely accessible, through use of a webserver. With objective 3.1.9 met through the frontend interface containing the current and previous system state presented through a graph.

While not specified as formal objectives the reliability and performance of DAIST are critical aspects for the real world. Stemming from earlier design principles, including automated identification of workers and auto-start configurations, the system demonstratives high reliability. Even in the event of sudden worker disconnections or complete power loss, the integrity and recovery mechanisms are effective, allowing it to reboot into a usable state with no human intervention. The only issues encountered were the need to restart tasks if they were running, which could be mitigated with persistent storage and more robust failover solutions.

However, the performance of the system was less optimal. The use of waiting loops to prevent network overburdening introduced a noticeable lag (up to a second) in actions requiring messages to travel to a worker node and back. This approach, while acceptable, is not suitable for high-throughput applications and led to unpredictable frontends when left open for long periods of time, as requests and error messages could accumulate when the manager was unreachable.

To improve performance, long polling is a simple change, which would allow requests to wait at the manager for a certain time before returning, maximising the chance of receiving new data. Alternatively, using sockets to maintain a constant connection could enable more efficient data streaming and reduce latency. Implementing these enhancements would significantly enhance the system's responsiveness and reliability, making it more suitable for demanding and potentially real-world applications.

4. FURTHER WORK

Having met all the proposed objectives to a reasonable degree, a set of objectives for further work were devised. These aim build upon the current system by adding more relevant functionality, securer systems and closer parity with existing modern workloads.

Table4.A

Ref	Objective
FOS.1	Rewrite the manager system within a statically typed language (Rust, Gleam)
FOS.2	Subscription based model for querying a task’s status
FOS.3	Containers as tasks, having workers run an engine that can execute them
FOS.4	Persistent storage for tasks, ideally networked across all workers

Ref	Objective
FOS.5	User authentication either built into the manager or as a separate service that users use to communicate with the manager
FOS.6	Usability testing of frontends
FOS.7	Increased plug and play for worker nodes: network boot, power over Ethernet (PoE)
FOS.8	Worker server pipeline improvements: automated building, deploying and versioning
FOS.9	Improved logging: integrating with manager and provide gated access to logs
FOC.1	Increased number of temperature sensors
FOC.2	Position temperature sensors as close to board chips as possible, ideally contact
FOC.3	Fine grain fan speed control, ideally through power curves
FOC.4	Use of other sensors for performance and system safety
FOC.5	More configuration controls
FOC.6	Logging of temperatures and changes to configurations
FO.1	Introduce a consumer for DAIST
FO.2	Introduce a power regulation subsystem

Further Objectives – Server

The majority of further work for the server is to do with the software it is running, as the hardware and packages were specifically designed to allow plug and play with any supported system and additional functionality is not particularly necessary.

FOS.1 requires the manager being rewritten in a type safe and system level language. During development of the worker server, it's codebase was incredibly effective once written and had no downtime due to the safety that such a language offers (in this case Rust), whereas the python system that the manager and cooling system were made in offered no such guarantees and as a result the system was frequently down from misrepresented types and data which was hard to fully eradicate. Gleam is another language option and while being fairly new, is built upon the powerhouse that is the Erlang VM (BEAM). Erlang is known for it use in telecommunications and massively scaling system, being designed at its core for fault tolerance, scalability and concurrency and would be a good use in this distributed system.

FOS.2, FOS.3, FOS.4 all link to general functionality of worker nodes and bolsters their usability. Containers will allow diverse and highly specific workloads to be underdone, ensuring the environment and resources a task is given remain consistent. (Li et al. 2024) discusses resiliency for container based systems especially when interacting with shared, persistent storage. Persistent storage is a necessity for users as the current system allows for services but within memory only, meaning data has to be routed out externally to be saved. Implementing task accessible persistent storage would allow users to reuse data across tasks and makes large scale data processing feasible. This in turn increases the security required of the system as limits need to be in place for tasks that can read and write to storage sites of other tasks.

Automating worker provisioning has been a large part of the system and FOS.7 and FOS.8 aim to further refine the implementation by reducing how much manual intervention is needed to add a worker node to a cluster. Network boot as mentioned previously allows the operating system of a device to be installed automatically when connected to a network instead of requiring a drive to be manually flashed. While PoE is a smaller optimisation, designed to optimise the physical server by having low power workers connected only via Ethernet cabling which would handle both network and powering requirements, cutting down physical server bulk.

For system resiliency and further latency optimisation a novel topology could used such as 'Ariadne' (Karp and Borrill, 2020) which removes the need for a top of rack switch by having machines wired directly to each other and then producing a node graph using simple protocols defined as 'Liveliness' and 'Common Knowledge'. An implementation of this would want to build a physical system using their virtual one as a basis and would aim to capture the same speed improvements and resiliency as their virtual one was able to gain.

Frontends were heavily used to abstract the available information and functionality into a user friendly interface. FOS.6 builds upon that idea and has these frontends properly tested with users to ensure it meets their needs as the current designs haven't been extensively tested and instead cater more towards administrative use and access to raw functionality. Increasing usability would also require implementing objectives FOS.5 and FOS.9 which aim to improving logging capabilities and introducing security measures to limit user actions to only what they have permission to do. Improved logging would allow for all actions

and their effects to be tracked and recorded which allows for system monitoring and forensics should it be needed.

Further Objectives – Cooling

A crude implementation was used for the cooling system as it was designed to ‘just work’, be responsive and remain very simple. FOC.1, FOC.2 and FOC.4 detail on improvements to the hardware, with FOC.1 and FOC.2 aiming on sensor capabilities by increasing the number of them and how they are used. This includes positioning them closer to the actual heat sources (in this case CPUs of the systems) which would provide more immediate readings of the temperature rather then the average air temperature within the enclosure. FOC.4 dictates utilising more efficent sensors such as the PT100 which is a resistance based contact sensor designed to operate (with varying accuracy) over ranges from -200 to 650 degrees at the cost of increased complexity to use.

Fan control of the system was limited to a boolean on and off, with no speed control. FOC.3 and FOC.5 aim to bolster this by having speed be controllable and determined by more sophisticated rule sets that can take zoning, power curves and multiple sensors readings into account. This would allow for more optimal cooling and better customisation for given hardware and user preferences.

Much like the server cluster, FOC.6 covers the same role as FOS.9 with improving logging capabilities. This again allows for better record keeping and tracking of system and user actions which are useful for intrusion and error detection, with the downside that extra considerations are needed such as file culling, authentication and backups to ensure it remains secure.

Further Objectives – Overall

One weakness with the current system is it’s lack of testable elements. While DAIST can be used generally, quantitative ways to test and monitor the usability and performance are not implemented. Objective FO.1 aims to remedy this by implementing an extensive testing and monitoring system that is able to allocate automated ‘users’ that can start and perform work on the system, ideally utilising both compute and long running tasks. Alongside allocation it would posses monitoring capabilities to track the time taken and how the system(s) responds to specified actions under specified conditions, allowing protocols to be made and tested.

To further align with real world DCs that DAIST aims to model, a power subsystem could be introduced. This, much like the the cooling system would be an essential component within the testbed and control power delivery, usage and fail-safe behaviour (including power loss or noisy power). This would allow for an increasing number of testing configurations in which statistics could be monitored and measured, providing a nice integration with FO.1.

5. CONCLUSION

Overall DAIST was successful and met it’s objectives of producing a simple yet comprehensive datacentre testbed. To achieve this it used a collection of relatively simple subsystems which combined to form an overall complex system. Specifically it is a server comprised of many smaller systems clustered together in a star topology alongside a corresponding thermal monitor and regulator. It has been built to be easily replicable with low cost hardware and it’s simple architecture allowed for seamless expansion to accommodate the need to scale.

The ability to be built upon and evolve was paramount as not all desired avenues were traversed and extra capabilities are both wanted and planned. Many of these features were previously discussed over what they would change about the system, add to it and fundamentally improve it, with focus on real-world performance and use cases being the main drive.

Although the project shifted from it’s initial plan which was focused on a pure security testbed, it has resulted in a functional and expandable system that offers the potential for more depth. This pivot occurred as existing research in security based systems was considerable and as such provided an opportunity to focus on a more authentic testbed that utilised multiple interconnected dependencies, in short It posses the capabilities of the initial plan’s system but offers much more.

REFLECTION

Over the course of this project I've been pushed far outside my comfort zone both in developing physical hardware systems and in writing the software that operates upon them. While I have existing knowledge in software development, having to consider the hardware it runs on and optimising for it has been a new area. The wide spread of work has been challenging but has given me an appreciation in the complexity of these systems and has greatly improved my time and allocation skills. Determining when enough functionality had been implemented to move on was a crucial part of getting the project done. My initial plan used a Gantt chart to track general times for certain milestones to have been met, being made quite conservatively as a best case scenario. I made sure that delays were covered by allocating fallback time, which was the three week Easter break. Having this fallback reduced the stress considerably as delays in hardware acquisition and development didn't impede the plan and it could continue. Figure A shows the initial plan's time, with Figure B showing the simplified final timing.

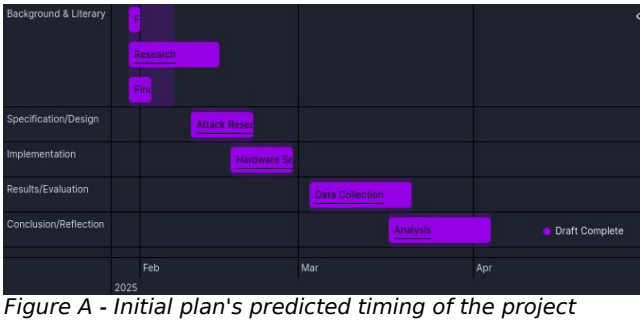

Figure A - Initial plan's predicted timing of the project

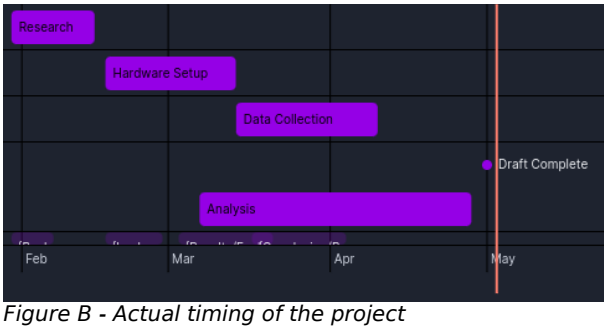

Figure B - Actual timing of the project

The delay in hardware acquisition also resulted in me deciding to trim the expanse of the hardware implementation, in using a more realistic subsection of the plan; notably a single server which was the core premise to the system and was an area I'm glad to have been able to focus on in greater detail. Picking my 'battles' effectively was a common occurrence during development, in that certain features and designs were made to be the most effective use of time in a hope to bring about a working implementation with as little effort as possible (while still adhering to the objectives, notably a simple and lightweight design). On this point it could be made that using industry level tools and software would have saved time and that is possible, but initial testing used these systems and I found them cumbersome and bloated for my purposes and eventually decided a better use of time was to work on my own bespoke software, which would allow me to learn the actual underlying implementation and design goals instead of relying on prebuilt tooling.

This project gave me a refreshing mix in both hands-on hardware work and the development of software. Throughout the hardware development I made extensive use of Cardiff University's Makerspace team who provided expert help and advice over the hardware aspects of my project and actually provided the hardware for me to use and test. They offered invaluable assistance in the support of 3D printing and assembling parts. The knowledge I was able to acquire in electrical and mechanical engineering that I otherwise would have not been able to experience was an immense help to both the project and my personal development.

Throughout the project I utilised a highly iterative approach to developing and designing both my software and hardware. I knew roughly what I needed to get the functionality I desired but the underlying implementation of this (including actual packages, services and physical components) was new to me. Having the ability to quickly implement and test something new, see if it worked and update my approach accordingly was a powerful and effective methodology for rapid learning and development. This approach came with the downside that progress could be made almost too quickly, and I found myself sometimes racing ahead instead of refining what I had. This has lead to several design issues, particularly within the codebase that while functional, do leave me wanting to ameliorate.

Having completed my objectives and finalised the implemented system, I was left wanting to develop more and continue pushing the capabilities of the system to further heights. The objective of designing it to be expandable and fairly open ended resulted in a constant flux of potential improvements and changes being generated throughout development. These would aim to boost the usability and functionality of the system and are evident within the expansive further work section of the report. My most immediate improvement to the system would be supporting container based workloads as it would exponentially increase the usability and generalisation of the system, ideally making it far more attractive for a typical user who would no longer be bound to using single python files for a task. Containers were initially avoiding due to the need of prebuilt tooling, specifically a container engine to execute these tasks. I tested some well known ones during initial development as mentioned within the report but I found them complex and time consuming to learn and such I opted for a simpler, custom made solution.

APPENDIX

BIBLIOGRAPHY

1. Mccrady, S.G. 2013. *Designing SCADA application software : a practical approach*. Elsevier. Available at: https://doi.org/10.1016/C2012-0-06034-6.
2. Alves, T., Das, R., Werth, A. And Morris, T. 2018. Virtualization of SCADA testbeds for cybersecurity research: A modular approach. *Computers & Security* 77, pp. 531–546. Available at: https://linkinghub.elsevier.com/retrieve/pii/S0167404818304905
3. Wilcher, Dr Don. “Turn a Raspberry Pi into a PLC Using OpenPLC.” *Control.com*, 22 Feb. 2024, control.com/technical-articles/turn-a-raspberry-pi-into-a-plc-using-openplc/.
4. Designing and Developing Virtual and Real Testbeds for Industrial Control Systems Education and Training – ProQuest. (2025, February 06). Retrieved from https://www.proquest.com/docview/2428115721?pq-origsite=primo&sourcetype=Dissertations%20&%20Theses
5. Low-Cost Industrial Controller Based on the Raspberry Pi Platform – ProQuest. (2025, February 06). Retrieved from https://www.proquest.com/docview/2748386763?pq-origsite=primo&sourcetype=Dissertations%20&%20Theses
6. *What is a data center?* (2024). https://www.cisco.com/c/en/us/solutions/data-center-virtualization/what-is-a-data-center.html.
7. Khalil, Y. And Elmaghraby, A.S. 2008. Data Center Resilience Evaluation Test-bed: Design and Implementation. In: *2008 IEEE International Symposium on Signal Processing and Information Technology*. pp. 369–374. Available at: https://ieeexplore.ieee.org/document/4775667/
8. Karp, A.H. and Borrill, P.L. 2020. A Design for Resilient Datacenter Networks. *IT Professional* 22(6), pp. 33–38. Available at: https://ieeexplore.ieee.org/document/9250323
9. How to Run a Raspberry Pi Cluster with Docker Swarm – Howchoo. (2025, February 07). Retrieved from https://howchoo.com/pi/how-to-run-a-raspberry-pi-cluster-with-docker-swarm
10. Naik, N. 2017. Docker container-based big data processing system in multiple clouds for everyone. In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. Vienna, Austria: IEEE, pp. 1–7. Available at: http://ieeexplore.ieee.org/document/8088294/
11. *What CNI classification means for UK data centre providers – Intelligent Data Centres*. 2024. Available at: https://www.intelligentdatacentres.com/2024/09/24/what-cni-classification-means-for-uk-data-centre-providers/
12. Sperling, E.2010. *Next-Generation Data Centers*. Available at: https://www.forbes.com/2010/03/12/cloud-computing-ibm-technology-cio-network-data-centers.html
13. Getting Started With Docker On Raspberry Pi (Full Guide). (2025, February 04). Retrieved from https://raspberrytips.com/docker-on-raspberry-pi
14. Kirkland, M. 2019. *Designing and Developing Virtual and Real Testbeds for Industrial Control Systems Education and Training*. Available at: https://www.lib.uidaho.edu/digital/etd/items/kirkland_idaho_0089n_11768.html
15. Zhang, S., Liu, X., Ahuja, N., Han, Y., Liu, L., Liu, S. And Shen, Y. 2015. On demand cooling with real time thermal information. In: *2015 31st Thermal Measurement, Modeling & Management Symposium (SEMI-THERM)*. San Jose, CA, USA: IEEE, pp. 138–146. Available at: http://ieeexplore.ieee.org/document/7100152/
16. Cuorvo, R., d’Ambrosio, N., Iorio, D., Perrone, G. And Romano, S.P. 2024. Securing Industrial Systems: A Testbed for Cyber-Defense Evaluation and Data Collection. In: *2024 20th International Conference on Network and Service Management (CNSM)*. Prague, Czech Republic: IEEE, pp. 1–7. Available at: https://ieeexplore.ieee.org/document/10814599/
17. Pastor, J. And Menaud, J.M. 2018. SeDuCe: a Testbed for Research on Thermal and Power Management in Datacenters. In: *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. pp. 1–6. Available at: https://ieeexplore.ieee.org/document/8555773
18. F. P. Tso, D. R. White, S. Jouet, J. Singer and D. P. Pezaros, “The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures,” *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, Philadelphia, PA, USA, 2013, pp. 108-112, doi: 10.1109/ICDCSW.2013.25.
19. Hawthorne, D., Kapralos, M., Blaine, R.W. and Matthews, S.J. 2020. Evaluating Cryptographic Performance of

Raspberry Pi Clusters. In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, pp. 1–9. Available at: https://ieeexplore.ieee.org/document/9286247/

20. *How can data centres balance operational resilience with the demand for sustainability, and what key innovations will drive this? – Intelligent Data Centres*. 2024. Available at: https://www.intelligentdatacentres.com/2024/11/11/how-can-data-centres-balance-operational-resilience-with-the-demand-for-sustainability-and-what-key-innovations-will-drive-this/
21. *What Is Containerization? | IBM*. 2024. Available at: https://www.ibm.com/think/topics/containerization
22. *Data Centres and Data Transmission Networks* | IEA. 2023. Available at: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks
23. Chen, S. 2025. Data centres will use twice as much energy by 2030 — driven by AI. *Nature*. Available at: https://www.nature.com/articles/d41586-025-01113-z
24. Lee, E., Oh, H. And Park, D. 2021. Big Data Processing on Single Board Computer Clusters: Exploring Challenges and Possibilities. *IEEE Access* 9, pp. 142551–142565. Available at: https://ieeexplore.ieee.org/document/9576750/
25. Li, Z., Saldías-Vallejos, N., Seco, D., Rodríguez, M.A. and Ranjan, R. 2024. Long Live the Image: On Enabling Resilient Production Database Containers for Microservice Applications. *IEEE Transactions on Software Engineering* 50(9), pp. 2363–2378. Available at: https://ieeexplore.ieee.org/document/10620003