School of Electronic Engineering
and Computer Science

Report

Programme of study:
BSc Computer Science with Industrial Experience

Project Title: Home Computing Network: A Model Sustainable Computer System

Supervisor:
Raymond Hu

Student Name:
Delane Brandy

Final Year
Undergraduate Project 2023/24

Queen Mary
University of London

Date: 6th May 2025

Abstract

Environmental sustainability is an existential threat, with the ever-increasing demand for computing exacerbating the problem. Embodied emissions represent the foremost challenge for sustainable computing. Continual progress in the operational efficiency of computing has put carbon emissions produced during manufacturing into focus. Apple reported that hardware manufacturing accounted for 74% of its overall footprint, while operational use accounted for 19%. To tackle this, a fundamental rethinking of the design, implementation, and lifecycle of computer systems and networks is required. This project aims to design a sustainable computer system that reduces the need for new hardware by leveraging distributed computing concepts within a local environment – fully utilising existing computing power within the home. Decentralised and sustainable computing models can enable a future where technologies' efficiency, scalability, and environmental sustainability concerns are addressed. Leverage underutilised resources, whether old or idle, and integrate them into a Kubernetes-based Home Computing Network (HCN) to extract performance while reducing consumption and reliance on centralised infrastructure. By integrating resource-sharing and dynamic telemetry-aware scheduling, this system promotes sustainability through the reuse and extended lifecycle of hardware, minimising e-waste and carbon emissions. A shift towards more flexible, cost-effective, and environmentally conscious computing enables a more sustainable growth of technology.

The project combines research into existing distributed computing approaches, prototype development, and performance evaluation. Key metrics such as latency, usability, and energy efficiency will be analysed to assess the system’s feasibility and environmental impact and evaluated synthetically using standard Kubernetes frameworks, Sonobuoy and kube-burner and under real-world constraints using AI media upscaling and distributed code compilation. By addressing the challenge of embodied emissions and prioritising the sustainable design principles of reduce, reuse, and recycle, this project seeks to contribute to a more environmentally conscious future for computing technologies.

Contents

Chapter 1: **Introduction**

The ever-increasing demand for computer hardware within the home and for supporting infrastructure solutions presents two key problems: ever-increasing amounts of waste, with poor recycling capabilities, and the unaddressed total of embodied emissions trapped during manufacturing. This report explores a sustainable computing solution to reduce the need for new hardware by leveraging decentralised distributed computing principles. By reusing and maximising the capabilities of existing devices within local environments, the project seeks to extend the life of devices, decrease hardware redundancy, and minimise manufacturing emissions without sacrificing usability.

1.1 Background

Environmental sustainability is an existential threat that requires all areas of society to work to mitigate its worst effects. Sustainable design creates long-term solutions and helps societies ensure the well-being of their people and harmony with the environment for generations (Interaction Design Foundation - IxDF, n.d.). The dramatic rise in computing demand incurs high energy and environmental demand worldwide, with projections of emissions caused by computing technology being 8% of worldwide emissions (Edler, 2015). These negative externalities are being supercharged by the ‘AI Revolution’, with global data centre energy needs increasing by 100% by 2030 alone (Barclays Research, 2024). Existing efforts target operational emissions, with technology areas seeing a 3- 8x improvement in operational efficiency (Eric Masanet, 2020), leading to the dominant source of emissions from manufacturing - embodied emissions (Gupta et al., 2021). Apple 2019 reported that hardware manufacturing accounted for 74% of its overall footprint, while operational use accounted for 19% (Apple, 2019). “Tackling ICT’s carbon footprint from hardware manufacturing requires us to fundamentally rethink the design and implementation of computer systems” (Gupta, et al., 2022). The core tenets of environmentally sustainable design – Reduce, Reuse, Recycle – focus the solution on these three core areas. The contemporary focus is typically on recycling – recycling alone is inadequate to address the increasing environmental impact of technology (Huang, 2009). Rapid industrial growth in developing nations further increases demands on natural resources as well as increases the production of industrial and municipal waste (Harris, 2005). Thus, there is a gap in the other two core tenets – reduce and reuse - focusing on reducing the number of manufactured processors and aiming to unlock the power of underutilised computing. Ideas on work done in distributed systems, particularly in the data centre space with cluster computing, and ideas in the IoT Edge, where they are exploring mixed use of local and cloud computing (Shang, et al., 2020).

1.2 Problem Statement

The overwhelming need to act and design sustainably across all sectors requires a radical rethinking of how we design, use, and extend the life of computing products. The underutilisation, centralisation of computing, excess waste, and overconsumption are unsustainable and wasteful. The foremost sustainability focus predominantly concerns operational efficiency and recycling, leaving room for significant gains in reducing and recycling computer technology. Decentralisation and ownership of computing and data also allow increased user privacy and control. Existing tools for developing and maintaining distributed, cluster computing systems are far too complex to set up and maintain for end users and lack the flexibility to easily span a range of computing platforms and types. Solutions are built for large-scale enterprise server solutions. Devices are locally limited to whatever computing capacity is available on a device, even if a more powerful idle device can be used on a network. Other than remote access of said device, few solutions are available, and an optimised resource-sharing network, HCN, presents a solution. The current solution to on-device performance limitation is that applications are increasingly moving to the cloud, which incurs a latency cost and limits the scope of applications, as well as gaining data privacy and usability costs. This solution enables greater on-device power efficiency and the total utilisation of computing performance across the network, negating the need for greater per-device performance.

1.3 Aim

The project aims to design a computer system, defined as a cohesive combination of hardware, software, and networking elements configured to execute tasks, with the aim of this system operating in a more sustainable and environmentally conscious way by utilising distributed computing concepts - reducing the need for new hardware and maximising the use of existing hardware. Adapting pre-existing hardware with new software to generalise and decentralise resources across a local network. The system is based on ideas surrounding a Home Computing Network (HCN); a decentralised or interconnected system of computing devices within a home environment designed to share resources, process tasks collaboratively, and optimise functionality through local and remote computation. This system leverages distributed computing concepts, edge computing, and cloud services to achieve efficient performance, resource utilisation, and intelligent automation for smart home applications (R&S, 2024). Levering existing cluster computing technologies, such as Kubernetes in tandem with Docker, connect computers or servers over a network to form a larger 'computer'. These cluster computing concepts can be combined with a suite of other distributed architectures to improve and extend computer hardware performance within the home. These solutions aim to create a unified home computing network that utilises more of the computing performance across the network with the aim of reducing the performance requirements per device. This system acts to minimise redundancy and extend the life of devices, as the computer can continue to be utilised, reducing the need to manufacture more processors and reuse devices. Evaluating if this approach is effective at its stated aims of environmental impact and its use-case feasibility, measured on metrics such as latency, performance, end-user complexity, security, and others.

1.4 Objectives

The primary objective of this dissertation is to design and evaluate a sustainable computing system that reduces hardware redundancy and embodied emissions by leveraging distributed computing principles. The following objectives support this aim:

1. **Develop a Framework and Prototype for a Home Computing Network (HCN)**
   - Design a system for resource sharing and distributed task execution across devices in a local environment, enabling a decentralised computing network.
   - Build a prototype based on the most suitable approaches to demonstrate the system's feasibility.
   - Perform a comparative analysis of the prototype against existing models regarding efficiency, scalability, usability, and energy efficiency.
2. **Select Optimal Technology Stack and Develop Solutions**
   - Identify and choose the proposed system's most suitable hardware, software, and networking components.

   - Develop new applications, extend existing software libraries, and implement necessary modifications to operating systems or pre-application layers to enhance compatibility, usability, and performance.
3. **Assess Feasibility and Adaptation**
   - Evaluate the practicality of leveraging existing hardware and software to implement a decentralised computing system.
   - Investigate and combine multiple approaches to create a robust solution adaptable for both home and general industrial applications.
4. **Evaluate Performance and Environmental Impact**
   - Conduct rigorous testing to assess the system's performance, focusing on key metrics such as latency, resource utilisation, usability, and energy efficiency.
   - Compare the system's efficiency and effectiveness with traditional centralised and cloud-based computing models.
   - Analyse the system's environmental impact, emphasising reductions in embodied emissions, extended device lifespans, and minimised hardware redundancy.
5. **Demonstrate Practical Use Cases**
   - Explore and implement real-world applications, such as AI-based functionalities (e.g., AI upscaling in televisions), to illustrate the system's practicality and user benefits.
6. **Conclude Findings and Provide Recommendations**
   - Summarise the strengths and weaknesses of the proposed solution.
   - Offer actionable recommendations for future development, emphasising areas for improvement and potential avenues for further research.

1.5 Research Questions

To guide the research and achieve the stated objectives, the dissertation will address the following key questions:

- How can distributed computing principles be applied to home environments to optimise the utilisation of existing hardware?
- What are the main technical and usability challenges in implementing an HCN, and how can they be mitigated?
- How does the performance of an HCN compare with existing centralised or cloud-based solutions in terms of latency, throughput, and energy efficiency?
- What metrics can be used to evaluate the environmental impact of distributed computing in home settings, and how effective is the proposed system in achieving sustainability goals?
- What are the implications of this system for user privacy, data security, and system complexity?
- How can this system be applied to practical use cases, such as AI-driven applications, to demonstrate its feasibility and advantages?

1.6 Report Structure

The report is structured as follows:

- **Introduction**: Provides background, problem statement, aims, objectives, research questions, and an overview of the report structure.

- **Literature Review**: Reviews existing work in distributed computing, cluster computing, supercomputing, and IoT Edge systems, identifying gaps and relevant technologies.
- **Design**: Conceptual Design and implementation of HCN, focused on the prototype HCN. It offers a model for what the HCN is and how it can be extended.
- **Evaluation**: Presents the system’s performance and environmental impact analysis, including comparisons with existing models.
- **Conclusion**: Summarises findings, discusses limitations, and suggests future research directions.

Chapter 2: **Literature Review**

2.1 Sustainable Computing

The impact of climate change, with the resultant focus on sustainable design methods, will shape the future of system design. (Lindsay, Gill, & Smirnova, 2021). The burgeoning field of Sustainable computing, or green computing, aims to shape this. The focus of where the sustainability gains will come from is, however, split. The rise of Machine Learning, Big Data, and the Internet of Things is increasingly responsible for the world's energy consumption and, as such, a major contributor to environmental pollution. Existing efforts target these operational emissions. A plethora of efficiency optimisations have been proposed and adopted to counter environmental overheads, leading to data centres seeing between 3- 8x energy efficiency improvements. Even when considering renewable energy sources, a Markov Decision Process (MDP)-based framework is proposed to optimise renewable energy use in cloud data centres, addressing the variability of solar power and workload demands. Dynamically managing microservices, prioritising essential work while de-prioritising non-mandatory services during energy shortages. Integrating solar power and battery storage, the framework reduces reliance on brown energy, enhancing sustainability, and demonstrating up to a 30% improvement in energy efficiency. (Xu, Toosi, Bahrani, Razzagh, & Singh, 2019). When factoring in ARM-based mobile and Iot solutions, we see similar gains. Operational Improvements paired with increased renewable energy sources shift the dominating emissions sources towards manufacturing.

The Architectural Carbon Modelling Tool (ACT) proposes a detailed framework to model and optimise carbon footprints during hardware design. ACT incorporates three tenets of sustainable design: Reduce (designing leaner systems), Reuse (emphasising general-purpose hardware over application-specific designs), and Recycle (extending hardware life cycles). ACT includes models that embody operational carbon emissions across components to enable designers to minimise environmental impact without sacrificing performance or efficiency. ACT optimises carbon yields and provides insights into the environmental trade-offs of using general-purpose versus application-specific hardware and strategies for designing leaner or modular systems. Case studies demonstrate ACT's potential to guide sustainable trade-offs in hardware configurations, balancing performance and efficiency with reduced environmental impact. This marks a step toward integrating sustainability as a primary metric alongside performance, power, and area in computing system design. (Gupta, Elgamal, Hills, Wei, & Hsien-Hsin S. Lee, 2022).

Local distributed computing represents a radical rethinking of computer systems, shifting from centralised computing models toward a decentralised approach that leverages idle computational resources within households. This departure from the norm shows promise in reducing and reusing technology. Reducing the need for new hardware by maximising the utilisation of existing computational resources within households, lowering the embodied carbon emissions associated with manufacturing, while having the added effect of reducing reliance and need for energy-intensive data centres. Reusing older or underutilised devices, such as outdated computers, and extracting computing performance by integrating them within a distributed network. Reducing the need to recycle by extending the functional lifespan of hardware prevents premature disposal. Lowering the embodied carbon emissions associated with manufacturing. HCN also enables efficient resource sharing and monetisation of surplus capacity, providing a novel framework for generating passive income (R & S, 2024).

2.2 Local Distributed Computing

Solving the problem of full utilisation of resources in a network raises multiple implementation challenges. Transferring data away from centralised cloud infrastructure to the home – leveraging local underutilised computing performance through the adoption of distributed principles is being approached in a wide range of ways – Distributed In-Network Computing (DINC), is one such way. DINC presents a fundamental rejection of existing network service chains regarding architecture, resource constraints, communication models, and performance requirements. DINC enables the distribution of computational tasks across multiple programmable network devices. Traditional in-network computing is per-device resource-constrained, such as memory and processing stages, which restricts the deployment of resource-intensive tasks. DINC overcomes these challenges by partitioning programs into segments and deploying them across multiple devices while maintaining correctness and functionality. DINC employs a multi-objective optimisation approach to minimise resource usage, latency, and segment duplication, ensuring efficient use of network resources. It integrates a planner, which optimises the placement of program segments; a slicer, which partitions the code; and a generator, which deploys the segments on programmable devices. Applications of DINC include telemetry, load balancing, and machine learning inference, where tasks are distributed across network paths to achieve high performance and resource efficiency. Strategy. DINC is a scalable, flexible, and easy-to-deploy modular framework (ZHENG, et al., 2023).

The HCN integrates advanced technologies to create a decentralised computing ecosystem, utilising idle resources from household devices. It employs a combination of containerisation, orchestration, and cloud computing paradigms to manage diverse devices and enable distributed task execution seamlessly. Central to HCN is the use of containerisation through Docker, which packages applications and their dependencies into lightweight, platform-agnostic, portable containers. This ensures seamless execution across heterogeneous devices, including modern PCs, older hardware, and embedded systems such as Raspberry Pi. Containerisation simplifies application deployment, scalability, and management while enabling devices with varying capabilities to collaborate efficiently. For container management, HCN employs Kubernetes, a robust platform for automating deployment, scaling, and task allocation. Kubernetes dynamically manages resources, distributes workloads across nodes, and ensures high availability through features such as self-healing and failover. HCN supports multiple cloud paradigms to achieve Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) offerings, scalable and cost-effective ways to build these decentralised home networks. HCN uses task distribution mechanisms to allocate workloads efficiently across devices. These tasks may range from web hosting and file storage to machine learning inference and data processing. Kubernetes orchestrates these tasks, optimising resource allocation and ensuring that workloads are balanced dynamically across the network. HCN incorporates technologies to monitor and monetise idle resources. Users can contribute unused computing capacity to the network and earn passive income based on resource utilisation. Decentralisation brings new challenges in managing data security, networking and computing infrastructure. In its current form, it adds significant overhead to the end user. (R & S, 2024).

Leveraging underutilised computing processing is an area of interest in Edge Computing. MCC-Edge is a system that integrates Mobile Crowd Computing (MCC) with edge computing to enable a sustainable, decentralised framework for real-time IoT data processing. This is demonstrated by a smart HVAC system where public-owned Smart Mobile Devices (SMDs) are opportunistically leveraged as computing nodes. Using idle processing power reduces the need for centralised servers and minimises energy consumption, costs, and environmental impact. At the core of the MCC-Edge system is its crowdsourced edge computing architecture, which transforms idle SMDs into "crowdworkers" that execute tasks opportunistically. This eliminates the need for dedicated edge hardware, making the system cost-effective and scalable. The hierarchical and layered architecture of MCC-Edge includes a Local Coordinator (LC), implemented on lightweight devices like a Raspberry Pi, which manages task distribution and resource coordination among connected SMDs. An MCC Coordinator (MC) oversees multiple LCs, redistributes tasks when resources are insufficient, and interfaces with the cloud as a fallback for additional computing power. The system uses real-time analytics to optimise HVAC operations dynamically. Local network communication ensures low-latency task execution, with a middleware solution that handles task scheduling, fault tolerance, and result aggregation. Reduction in the use of cloud processing while fully utilising the processing power of existing computing devices reduces the need for new hardware, while improving the efficiency of the HVAC system (Pramanik, Pal, & Mukhopadhyay, 2024).

2.3 Quantifying Success in Computing

Quantifying success in computing involves evaluating systems across diverse dimensions such as efficiency, sustainability, performance, and scalability. The ACT framework emphasises the importance of sustainability in computer systems by modelling and optimising carbon emissions across hardware lifecycles. Success is quantified through metrics such as the carbon-delay and carbon-energy products, which enable trade-offs between performance, power efficiency, and environmental impact. Similarly, MCC-Edge integrates mobile crowd computing (MCC) with edge computing to process HVAC data locally, reducing reliance on cloud infrastructure. The success of this approach is measured through its ability to decrease energy consumption, minimise latency, and mitigate environmental impacts, showing a marked improvement over traditional cloud-based systems. HCN demonstrated success through the leveraging and quantifying of passive income generated by the system through the adoption of selling computing performance as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) solutions. DINC measures success through latency reduction, resource optimisation, and scalability in data centre and wide-area network scenarios. Multi-objective optimisation ensures effective deployment strategies, with evaluations showing efficient functionality across large-scale networks.

2.4 Conclusion

This literature review highlights the evolution of computing technologies towards more sustainable, decentralised, and efficient models. The rise of Sustainable Computing reflects an urgent response to the ICT industry's environmental challenges. From the ACT framework, which addresses carbon emissions across hardware lifecycles, to dynamic frameworks like Markov Decision Process (MDP)-based systems optimising renewable energy in cloud data centres, sustainability is becoming an integral metric in system design. These approaches demonstrate significant improvements in energy efficiency, reduced reliance on brown energy, and minimised environmental impacts.

Exploring Local Distributed Computing reveals innovative solutions such as DINC, HCN, and MCC-Edge, which leverage underutilised resources to decentralise computation. These systems reduce energy consumption and carbon footprints and improve scalability, flexibility, and cost-efficiency. DINC addresses the scalability challenges of in-network computing by distributing tasks across programmable devices. At the same time, HCN enables households to contribute idle computational power to decentralised networks, generating passive income and reducing e-waste. Similarly, MCC-Edge showcases the potential of crowdsourced edge computing to transform existing mobile devices into dynamic, ad-hoc processing nodes, reducing the reliance on centralised servers.

The ability to quantify success in computing emerges as a critical theme across these studies. Metrics such as the carbon-delay product, resource utilisation, latency reduction, and passive income generation provide concrete ways to evaluate the performance and sustainability of these systems. Collectively, these frameworks reflect a shift toward more efficient computing and are aligned with broader environmental and economic goals.

In conclusion, the future of computing lies in integrating sustainability, decentralisation, and performance optimisation into the core of system design. By leveraging idle resources, optimising for environmental impact, and adopting decentralised architectures, these technologies pave the way for a greener, more efficient, and economically viable computing landscape. Further research should focus on addressing challenges such as security, interoperability, and scalability to ensure the widespread adoption of these transformative approaches.

Chapter 3: **Design**

A new model for home computing, with sustainability as its core, must be secure, extensible, and functional. Extensible in practice means that computational hardware can be turned off, replaced, or upgraded with the system knowing and adjusting accordingly. The key principle behind this HCN is the utilisation of idling compute; where hardware sits unused – long and short term – we use it to perform tasks on the network. The HCN aims to reduce the redundancy of computing resources, extend the life span of existing compute resources, and, as a result, reduce waste. This HCN offers advantages over a conventional approach in users' ability to tap into additional sources of compute on the network. To achieve this, extension of Kubernetes is required; conventionally, it is used in a cluster of near-identical compute resources. Resulting in the scheduler taking very little consideration of real performance. With the HCN, each node radically differs in architecture, node capabilities, and long and short-term limits. The dynamic and static node labelling system is the extension that informs the scheduler about true network resources and enables easy job design that considers this.

3.1 Design Considerations & Constraints

Focusing on preventing the degradation of hardware components and addressing the limitations in software support is key to extending the working lifecycle of computers. The performance and endurance of NAND flash memory, SD cards, and USB drives have limited write endurance, typically tolerating between 10,000 and 100,000 write/erase cycles per block (Aras et al., 2020). Flash storage is subject to repeated small writes for logs, caches, and job state tracking when used as persistent local disks in distributed systems. These operations can rapidly degrade the media without mitigation, resulting in silent data corruption or total failure. This directly undermines the sustainability objective by shortening the hardware lifecycle. Several mitigation strategies can be employed to extend flash storage's lifespan. Wear-levelling algorithms can distribute writes more evenly across the storage medium (Aras et al., 2020). In-memory buffering of logs and job state, combined with delayed or batched write operations, can significantly reduce wear without sacrificing performance. Additionally, centralising SSD-based network-mounted storage and prioritising nodes with SSDs for write-heavy workloads can help offload stress from less durable devices. These strategies are critical when nodes are expected to participate in persistent, stateful workloads such as media caching.

Beyond storage constraints, the HCN must also account for hardware heterogeneity. Devices may vary significantly in architecture (`amd64`, `arm64`), memory, compute capability, and GPU support. This presents a challenge for application compatibility and performance consistency. Modern containerisation tools like Docker provide a solution by packaging applications and their dependencies into cross-platform and architecture images, ensuring consistent execution across diverse systems. When orchestrated with Kubernetes, container workloads can be scheduled intelligently using dynamic node labels that reflect device capabilities and constraints (R & S, 2024).

At the centre of the design is the use of Windows Subsystem for Linux 2 (WSL2), enabling Windows devices to run Linux-based workloads without needing dual-booting or external hardware. Simplifying system development and powering the underlying technology for Docker on Windows presents a robust solution for unifying software platforms.

WSL2 runs a virtualised Linux kernel in a Hyper-V container; this architecture introduces a compatibility layer that abstracts hardware access, impacting the performance and availability of certain low-level features. GPU frameworks, like OpenCL, rely on direct access to hardware in native Linux via /dev interfaces and kernel modules. Microsoft provides vGPU and WSLg, but this approach depends heavily on hardware vendors offering compatible drivers. As a result, GPU support in WSL2 is inconsistent. For instance, while both Nvidia and Intel offer support for OpenGL and Vulkan, Intel extends support for OpenCL – an important cross-platform framework, with Nvidia enabling CUDA. Due to the virtualisation layer, a small performance hit of 7% is seen when compared to a native installation on the CPU[1]. GPU performance paints a very different picture where performance is highly dependent on hardware platform and graphics framework, due to Intel's Vulkan support being heavily reliant on Dozen, a driver that implements Vulkan over Direct3D 12, a performance delta upwards of 50% is seen. Given this variability, the system scheduler must be acutely aware of each node's GPU capabilities in terms of hardware presence, API-level support, and runtime stability. Dynamic node labelling based on real-world benchmarks is essential to ensure workloads are scheduled for compatible and performant devices.

Shared storage presents another design challenge. Specific use cases—such as distributed AI inference or compilation—may require access to common datasets or intermediate results across nodes.

Security remains paramount in any distributed system, particularly one that leverages semi-trusted devices on a local network. In this design, SSH key-based authentication secures communications between nodes, avoiding password-based logins altogether. Each node must register securely with the control node using a join script or signed token. The system can be further hardened by isolating Kubernetes APIS, enabling role-based access control (RBAC), and encrypting sensitive data in transit using TLS.

3.2 HCN System Architecture

The HCN is a Kubernetes-based system that orchestrates workloads across idle, heterogeneous devices within a local environment. Using real-time and recorded performance metrics, these devices—from user laptops to home servers—are dynamically discovered, labelled, and scheduled through Kubernetes. The system supports a range of practical, distributed workloads, including AI media processing, code compilation. At the core of HCN is a lightweight Kubernetes cluster that abstracts and unifies a diverse set of nodes under a single control plane, enabling intelligent and efficient use of idle compute capacity. All orchestration, scheduling, and workload placement logic leverages Kubernetes primitives, extended by real-time observability through on-node systemd services.

3.2.1 Control Plane

At the centre of the HCN is the control plane, a dedicated Linux host that serves as the central orchestrator for all cluster operations. Responsible for managing the lifecycle of participating nodes, scheduling workloads, collecting metrics, and ensuring overall consistency across the system. To fulfil these roles reliably, the control plane must:

- Be a reliable, always-available device
- Have at least 2 CPU cores

[1] Based on 7-Zip compression benchmarks conducted using the Phoronix Test Suite. See Appendix A for full results.

- Have at least 2 GB of RAM
- Have a full-duplex network connection
- Run a native Linux installation – to support low-level tools and monitoring services without the limitations imposed by virtualisation.

Built using Kubernetes kubeadm, the control plane manages the highly dynamic and heterogeneous set of devices. Including:

- A mix of architectures: **`amd64`** and **`arm64`**
- Varying levels of GPU driver support
- Differing and dynamic availability patterns

The system uses Kubernetes-native features such as node labels, taints, tolerations, and node affinity to inform scheduling decisions. These mechanisms are extended with custom scripts and monitoring tools that allow the control plane to label nodes based on their performance and idleness, enabling a responsive and sustainable compute environment.

Control Plane setup must proceed everything else. In addition to orchestration and scheduling, the control plane performs essential infrastructure provisioning. **`init-control.sh`** is responsible for:

- Installing dependencies – docker, docker-cri, kubeadm, etc
- Establishing an SSH server – the first entry point for control-node communication.
- Generating SSH keys to enable passwordless control-node communication
- Set up the HCN kubeadm cluster
- Set up cluster networking using Calico
- Generate the kubeadm join command
- Set up storage if it's available.

3.2.1.1 Storage

The control plane can optionally serve as the host for network-attached storage (NAS); a huge enhancement to the functionality of the HCN, it offers advantages in AI inference or parallel code compilation, where common datasets, models, or intermediate files must be accessible across multiple nodes. To minimise wear on NAND flash devices, the system encourages offloading write-heavy tasks to SSD-backed nodes or centralised NAS. Flash-based media such as USB drives and SD cards have limited write endurance and are susceptible to degradation under sustained write conditions. Using network-mounted volumes for persistent data ensures that short-lived or frequently written files (e.g., logs, caches, build artefacts) do not accelerate hardware failure on less durable devices.

The shared storage is provisioned using a statically configured NFS server hosted on the control plane. The **`setup-storage.sh`** script prompts the user for a block device, mounts it at **`/mnt/shared_drive`**, and exports it to the entire subnet using **`nfs-kernel-server`**. The export is automatically detected and patched into the nfs-pv.yaml manifest, which defines a 100 GiB **`PersistentVolume`** with **`ReadWriteMany`** access mode. This volume is made available to the cluster and bound via a corresponding **`PersistentVolumeClaim`**, allowing Pods to read and write to the storage across multiple devices concurrently. The **`PersistentVolume`** is configured with a Retain reclaim policy, ensuring data persistence even if a PVC is deleted or redeployed.

To reduce dependency on external networks and improve deployment speed, the control plane hosts a private Docker registry accessible across the cluster. Deployed via a single-replica Kubernetes **`Deployment`** and exposed on port 30000 using a **`NodePort`** service, the registry stores image data on the control plane's NFS-backed shared volume. This is achieved by binding the registry container's /**`var/lib/registry`** path to a **`PersistentVolumeClaim`** backed by the 100 GiB **`ReadWriteMany`** NFS share. This setup allows all nodes to push and pull images locally, enabling faster deployments and supporting offline operation in constrained environments. The registry is initialised automatically as part of the storage setup process.

3.2.2 Scheduling

The control plane delivers core orchestration functionality through a dynamic, label-based scheduling system. It interprets two main categories of node information: static labels reflecting persistent hardware characteristics, and dynamic labels representing real-time node availability. This allows jobs to be scheduled intelligently based on performance capabilities and real-time status.

Some static tables are generated by **`setup-drivers.sh`**, these inform the node platform support and dictate what jobs can be scheduled:

- cuda: true
- vulkan: true
- opengl: true
- opencl: true

When platform support is detected, the appropriate labels are applied; in the case of NVIDIA GPU support on WSL2:

```
kubectl label node "$NODE_NAME" cuda=true vulkan=true opengl=true –
overwrite
```

Based on the platform and benchmarking results support static labels are generated by **`static_labelling.py:`**

- cpu: [low, mid, high]
- has-battery: [true, false]
- vulkan-perf: [low, mid, high]
- opengl-perf: [low, mid, high]
- cuda-perf: [low, mid, high]
- opencl-perf: [low, mid, high]

Job specifications define what platform requirements they want and what performance levels are required. **`real-esrgan-deployment`** has the following **`spec`** requirements:

```
nodeSelector:
  cuda-perf: high
  cuda: "true"
```

This ensures the workload is only assigned to nodes with matching labels, avoiding misplacement and optimising throughput, compatibility, and system longevity. The control plane leverages these labels to support differentiated workload types—from high-intensity AI processing to lightweight, distributed compilation—by matching resource demands to node capability.

3.2.2.1 Idle-Aware Scheduling

The HCN evaluates real-time metrics to determine whether a node can take on work. **`dynamic_labelling.py`** runs as a systemd service that periodically collects CPU utilisation and battery level data to determine if the node is idle.

If a node falls within the predefined ranges:

- CPU utilisation is below 30%
- Battery level is above 70%
- Check if the load comes from containerised compute (Kubernetes), do not taint.

The node is dynamically labelled as **`idle=true`**. However, if it falls out of these ranges, it is labelled as **`idle=false`** and tainted with **`idle=false:NoExecute`**. This taint means that Kubernetes evicts all pods on the node and prevents new ones from being scheduled. When the node falls back into the expected range, it's untainted. When the system detects if user load accounts for 20% of all loads, jobs are suspended.

```
if idle and power_ok and not is_user_load_present():
          label_node(node, "idle", "true")
          if has_taint(node, "idle", "false"):
              untaint_node(node, "idle", "false")
      else:
          taint_node(node, "idle", "false")
```

On WSL2 nodes where Linux doesn’t accurately report system utilisation, this flow is done slightly differently.

`monitor_status.py` is run on Windows via PowerShell.exe. It collects unitisation metrics, outputting **`idle=true/false power_ok=true/false vmem_cpu=x.xx`**. This data is then piped to wsl_dynamic_labelling.py, which applies idle labels and taints. Both are initialised as the systemd **`dynamic-labelling.service`**.

These dynamic labels ensure that workloads are only dispatched to otherwise used nodes. This two-tiered labelling model—combining static hardware profiles with dynamic availability—provides a lightweight yet flexible mechanism for scheduling jobs across a highly variable pool of user-owned devices.

3.2.2.2 Scheduling Workflow

When a user submits a job to the HCN, it is encapsulated in a Kubernetes Pod or Job specification that defines required scheduling constraints. The control plane applies a hybrid scheduling model: it first filters nodes based on node availability, where untainted nodes are selected, then static labels (e.g., **`cpu=high`**, **`gpu=high`**, **`cuda=true`**) are checked to narrow the sections to match the pod spec with node capabilities.

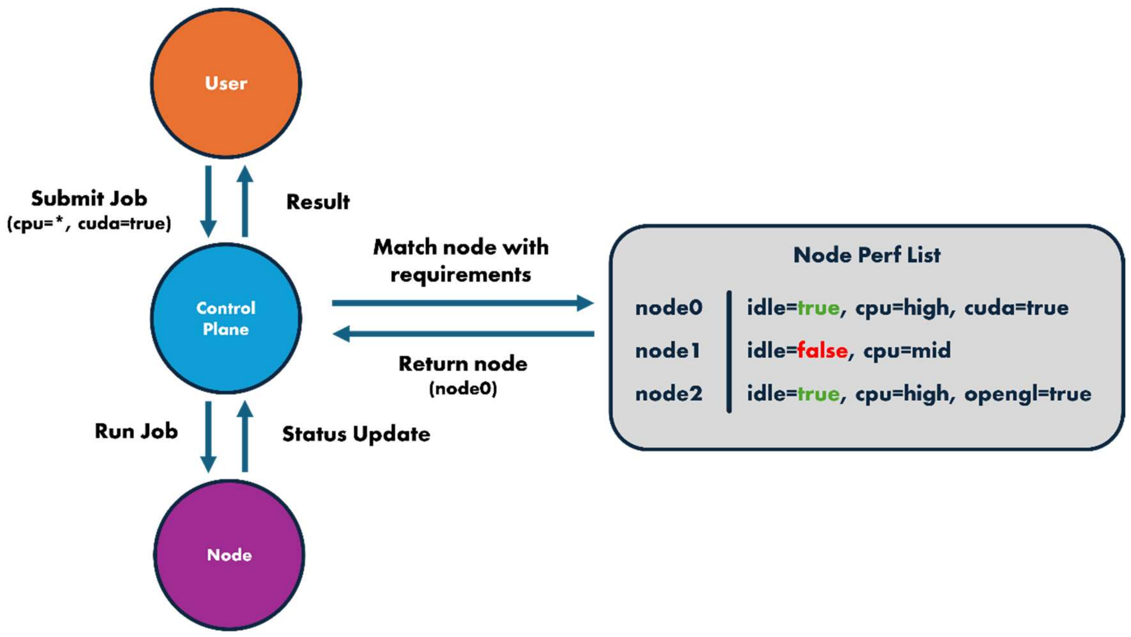

Figure 1. Scheduler Workflow

Only nodes that meet performance and availability requirements are eligible for scheduling. This prevents jobs from interrupting active users or running inefficiently on underpowered hardware. The result is a responsive and sustainable system that optimises performance and energy efficiency without requiring constant user intervention.

Node availability is managed exclusively through taints rather than labels. When a node is deemed unavailable, it is tainted with **`idle=false:NoExecute`**, causing any running pods to be evicted and preventing new workloads from being scheduled. Kubernetes controllers like **`Deployments`**, **`Jobs`**, and **`DaemonSets`** are used to ensure that pods are rescheduled when the node becomes available again. They continually search for Ready and untainted nodes that match the pod spec and schedule pods onto them. Alternatively, pods may include a toleration to allow them to remain scheduled on tainted nodes or to tolerate the taint temporarily before eviction.

```
tolerations:
  - key: "idle"
    operator: "Equal"
    value: "false"
    effect: "NoExecute"
```

Such toleration is only necessary if pods are intended to land on currently tainted nodes or to remain running temporarily after a taint is applied. This system deliberately omits tolerations to ensure that workloads are scheduled strictly on idle (untainted) nodes.

3.2.3 Nodes

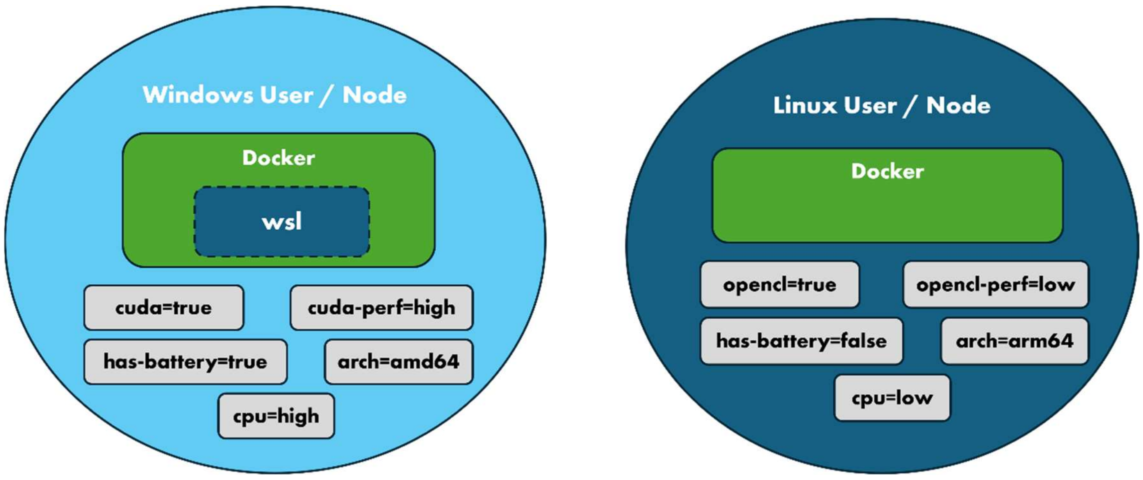

Figure 2. Visual Representation of Nodes with Labels

Nodes are the execution environments where jobs are run and code is processed. They represent the backbone of the system—distributed, user-owned devices that contribute computing power when idle. Some nodes are powerful desktop machines with dedicated GPUS and SSDS, while others are lightweight, low-power devices like Raspberry Pis using Microsd cards. Architecturally, **`arm64`** and **`amd64`** devices are supported, enabling a broad hardware range—from Raspberry Pi's to high-performance home servers.

This hardware diversity is expected and embraced. The system doesn't rely on static specs or user-defined capabilities to handle it. Instead, benchmarking is performed using the Phoronix Test Suite when a node first joins. Testing logic is defined by the platforms supported, detected in setup-drivers.sh, which on native platforms runs tests to check for driver availability and assigns related labels.

```
  if command -v nvidia-smi &>/dev/null; then
    info "CUDA supported"
    kubectl label node "$NODE_NAME" cuda=true --overwrite
  fi
```

Tests are automatically selected based on the detected CPU architecture and GPU vendor support:

```
phoronix-test-suite batch-benchmark build-linux-kernel <<< 1
...
if has_label cuda; then
  phoronix-test-suite batch-benchmark octanebench
fi
```

Benchmarks have been selected to give an accurate representation of platform-specific performance. **`OctaneBench`**, for example, is benchmarking software that focuses only on NVIDIA devices and, as a result, computes using only CUDA. Additionally, a specific focus on software that works to access the real performance of GPUS across native and WSL2 Linux. Phoronix Test Suite is used because it focuses on unattended, open-source, and transparent benchmarks, offering a range of platform-specific benchmarking suites that **`OpenBenchmarking`** provides.

Benchmark	Platform	Justification
`build-linux-kernel`	**`CPU`**	Cross architecture and industry standard with a particular focus on complex dependency resolution and multithreaded compilation.
`octanebench`	**`Cuda`**	Specifically designed for GPU rendering performance using NVIDIA's CUDA API
`vkmark`	**`Vulkan`**	Tests graphics performance using Vulkan API, measuring real-time rendering capabilities
`juliagpu`	**`OpenCl`**	Runs compute kernels in Julia via OpenCL
`unigine-heaven`	**`OpenGl`**	Industry standard graphics-intensive benchmark using OpenGL to stress-test rendering pipelines and GPU capabilities.

Benchmark results are parsed by **`static-labeling.py`**, which maps performance to tiers (**`high`**, **`mid`**, **`low`**) and applies Kubernetes labels via:

```
subprocess.run([“kubectl”, “label”, “node”, node, f” {key}={value}”,
“--overwrite”], check=True)
```

For instance, a device compiling the Linux kernel with a score of 150 would be labelled **`cpu=high`.** GPU scores are treated similarly based on thresholds (e.g., Vulkan FPS > 60 = **`vulkan-perf=high`**). This approach makes scheduling performance-based and hardware-agnostic. When a user updates computer hardware, the benchmarking and reporting suite can be updated by retriggering node-perf with the **`--update`** argument.

Labels aren't limited to performance. Each node is also tagged with metadata like **`arch=arm64`** (provided natively by Kubernetes), **`has-battery=true`**, and **`storage=sdcard|ssd|hdd|virtual|unknown`** inferred during onboarding by inspecting the device and its connected components:

```
def has_battery():
    power_devices = subprocess.check_output(["upower", "-
e"]).decode().splitlines()
    return any("battery" in device for device in power_devices)
```

This distinction is critical. Tasks with large write volumes are steered toward SSD-backed machines, avoiding micro SD devices prone to failure under sustained load. GPU-bound workloads are assigned only to nodes with the required APIS (CUDA, Vulkan, etc.), confirmed through real benchmarks—not guessed.

Nodes in the HCN can also have different roles. Some are passive compute-only devices, while others are user-facing laptops where tasks are submitted but rarely executed locally. Many devices fall into both categories depending on context. To avoid interfering with active users, nodes only become schedulable when idle, determined by telemetry data such as CPU load and battery level. Battery-powered nodes are only labelled **`idle=true`** when CPU usage drops below 20% and charge exceeds 80%, determined using data from **`dynamic_labelling.py`** and applied dynamically:

```
kubectl label node node1 idle=true
```

Nodes are not expected to be permanently online. Laptops may disappear when unplugged or closed, while Raspberry Pis and desktops might run 24/7. The control plane is designed for this volatility—Kubernetes automatically tracks node availability, and no single node, other than the control plane, is essential for overall system stability.

Joining the cluster is secure by design. Each node uses SSH key-based authentication to fetch a join command from the control node:

```
ssh-keygen   -t   rsa   -b   4096   -f   "$KEY_FILE"   -N   ""   -C
"$SSH_UNAME@$IP_ADDRESS"
...
sshpass -p "$SSH_PASSWORD" scp -o StrictHostKeyChecking=no I am running
a few minutes late; my previous meeting is running over.
    "$PUB_KEY_FILE" "$SSH_UNAME@$IP_ADDRESS:~/.ssh/temp_key.pub"
...
scp    -i    $KEY_FILE    "$SSH_UNAME@$IP_ADDRESS:~/join-command.sh"
/tmp/join.sh
./tmp/join.sh
```

This approach avoids password-based logins and ensures only authorised devices can enter the network.

3.2.3.1 Windows Nodes

Windows support is essential for the scalability and inclusiveness of the HCN. Most consumer devices run Windows, and without a way to integrate them, the network would exclude a large pool of potentially idle compute. The system uses WSL2 (Windows Subsystem for Linux) to address this, which provides a full Linux kernel virtualised inside Windows. This allows Windows machines to run Ubuntu-based tooling—including Docker, Kubernetes (**`kubelet`**, **`kubectl`**), benchmarking, and telemetry tools—without dual-booting or deep OS changes.

However, WSL2 introduces technical challenges that require platform-specific handling. Networking in WSL2 can be unreliable, especially when resolving the control node's IP. To fix this, a helper script (**`hostname.py`**) is run from PowerShell inside **`hcn.sh`**:

```
IP_ADDRESS=$(powershell.exe python hostname.py raspberrypi | tr -d '\r')
```

WSL2 also lacks **`systemd`** by default, which is essential for running **`kubelet`**. The **`init.sh`** script detects WSL2 and automatically enables **`systemd`** by writing to the correct config:

```
echo -e "\n[boot]\nsystemd=true" >> /etc/wsl.conf
```

Users are prompted to run **`wsl --shutdown`** to apply the change, after which **`systemd`** services can run normally.

GPU support in WSL2 is available but fragmented. NVIDIA cards support CUDA and Vulkan through passthrough drivers. Intel GPUs support OpenCL and Vulkan via Dozen (a translation layer built over Direct3D 12). GPU platform detection and driver installation are handled conditionally in **`setup-drivers.sh`**:

```
# For NVIDIA
    apt-get  -yqq  install  build-essential  software-properties-common
freeglut3-dev \
      mesa-vulkan-drivers   mesa-utils   vulkan-tools   libgl1-mesa-glx
libglu1-mesa-dev  \
      nvidia-container-toolkit libvulkan1 pocl-opencl-icd mesa-common-
dev
```

Even with drivers installed, performance varies. CPU tasks in WSL2 typically run ~8% slower than native Linux. Depending on API and backend implementation, GPU performance drops can be much steeper—sometimes over 50%. This makes benchmarking especially important on WSL2: a node may support Vulkan on paper, but real FPS results might force a **`gpu=low`** label, e.g., from 15 FPS in **`vkmark`**.

Dynamic labelling in WSL2 is implemented via a hybrid pipeline. The script setup-wsl-labelling.sh installs two components:

- **`monitor_status.py`** – a PowerShell-compatible script that monitors CPU load and battery state from Windows.
- **`wsl_dynamic_labelling.py`** – a Linux-side parser that reads telemetry from standard input and applies Kubernetes labels or taints accordingly.

These are connected by a wrapper script (dynamic-labelling.sh) which bridges PowerShell and Linux:

```
powershell.exe py monitor_status.py | python3 wsl_dynamic_labelling.py
```

Despite these limitations, WSL2 significantly broadens the reach of the HCN. It enables everyday users to contribute compute with minimal friction, tapping into high-performance hardware that would otherwise sit idle. Bridging the Windows–Linux divide makes the system accessible to a broader audience, reflecting real-world device ownership patterns.

3.2.4 HCN System Orchestration

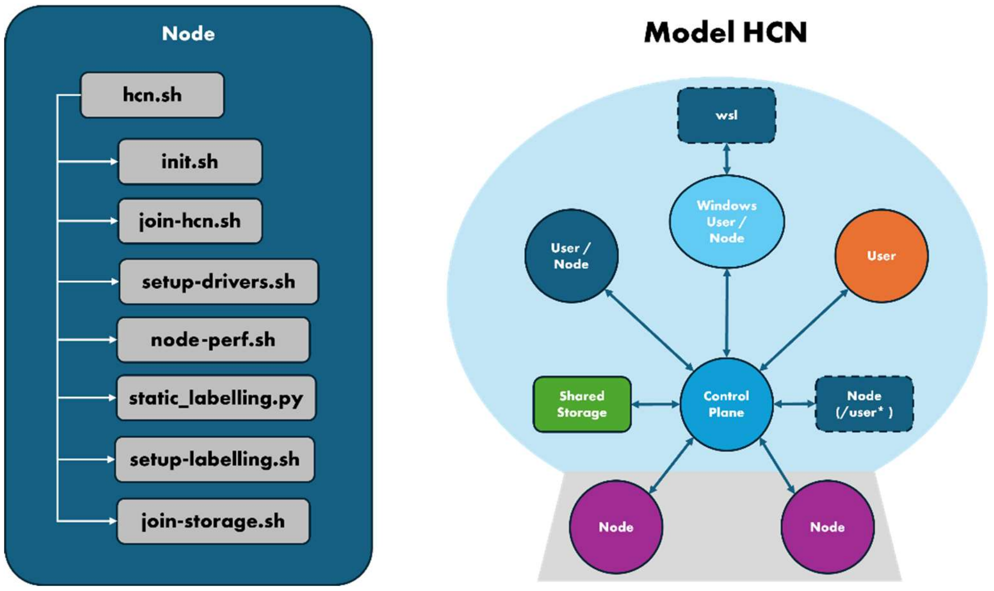

Figure 3. Overview of HCN System Initiation

Figure 4 outlines the orchestration flow of the HCN, which spans two tiers: the central Control Plane and multiple distributed Nodes. The system is bootstrapped through the unified hcn.sh wrapper script, which handles environmental detection, credential exchange, system setup, and performance profiling. See Appendix 2 for a more detailed textual overview.

3.3 Use Case Design

Use cases are core to defining, validating, and guiding the implementation of systems against real-world requirements. To demonstrate the value and use of the HCN, the system's use case design targets two distinct hardware platforms: CPU and GPU. Distributed code compilation targets CPU resources, while distributed AI computation targets GPU, with a general Kubernetes benchmarking system supporting broader system evaluation.

Developing two distinct types of Kubernetes jobs, daemons and deployments, that target different areas allows the evaluation of the system's scalability, scheduling accuracy, and performance, quantifying effectiveness. The design demonstrates how minimal modifications to existing workloads can enable seamless integration with the HCN platform. By simply specifying scheduling constraints such as nodeSelector: **`gpu=high`**, **`cuda=true`**, existing Kubernetes jobs can be redirected to appropriate nodes, highlighting the ease of transition and the practical flexibility of the system architecture.

3.3.1 Distributed Code Compilation

Distributed code compilation demonstrates the system's ability to parallelise and accelerate software builds by distributing compilation tasks across idle nodes. Distcc, a self-contained distributed compilation framework, is at the core of the architecture. In its default behaviour, Distcc orchestrates the scheduling of distributed compiler jobs by referencing a list of available servers—this server list is the core of our extension for use within the HCN.

3.3.1.1 Static and Dynamic Profiling

Limited by the inability to use Kubernetes scheduling, integration of distcc into the HCN focuses on refining and dynamically updating the list of available hosts using dynamic and static labelling logic. On clients that opt into dynamic code compilation, the **`setup-client-distcc.sh`** is run. The script first installs and configures ccache to enable local compilation caching and intercept compilation calls. Distcc is installed and set as the default compiler by manipulating environment variables in Linux user profiles, allowing standard compilation commands to route through distcc automatically without developer intervention. Lastly, we establish the **`update-distcc-hosts`** systemd service that runs periodically to maintain an accurate and up-to-date list of active compilation servers. By calling the **`client-refresh-hosts`** script, we query the control plane for available distcc pods that meet our client's platform architecture requirements and amend the host list distcc uses to schedule compilation jobs, gathering a list of available arm64-native pods and amd64-cross compiler pods on arm nodes and amd64-native pods for x86 nodes.

```
kubectl get pods -n "$NAMESPACE" -l "$LABEL_SELECTOR" -o
jsonpath='{range .items[*]}{.status.podIP}{"\n"}{end}' \
  | sort -u \
  | awk -v conns="$CONNECTIONS_PER_NODE" '{ print $1 "/" conns }' \
  > "$HOSTS_FILE"
```

3.3.1.2 Daemon and Docker Architecture

Achieving platform heterogeneity in a scenario where software and hardware platforms are at odds requires specificity. Necessitating precise, architecture-aware system design that demands DaemonSets are targeted and compatible with both the Docker images they deploy and the node architectures on which they are scheduled. Resulting in the development of three DaemonSets:

- **distccd-arm64** – Deployed on **`arch=arm64`** nodes, and made available to arm distcc clients for native compilation
- **distccd-amd64** – Deployed on **`arch=amd64`** nodes, and made available to distcc x86 clients for native compilation
- **distccd-cross** – Deployed on **`arch=amd64`** nodes labelled **`cpu=high`**; with a cross compiler installed and used. This pod is made available to arm distcc clients. It extends the network's compilation capacity by leveraging powerful hardware for cross-architecture builds.

Each DaemonSet applies Kubernetes node selectors and affinity constraints to schedule pods appropriately. Docker images for each role are built using multi-architecture support with Docker Buildx and pushed to a private local registry, allowing nodes to pull platform-specific images securely and efficiently.

3.3.2 Distributed AI Computation

The ever-increasing integration of AI within the home, processed in the cloud, presents many data ownership, privacy, and latency concerns. Leveraging existing compute found in the home to bring AI computation to the edge addresses these challenges with the added environmental benefit. Distributed AI Computation demonstrates the system's ability to offload AI-intensive workloads onto idle GPU-equipped nodes.

3.3.2.1 Upscaling Orchestration

Structured around a client-server model, the AI image upscaling model leverages the Real-ESRGAN model to enhance pictures and video. **`server.py`**, a containerised FastAPI application that manages the full model lifecycle and frame processing pipeline. The server exposes two HTTP endpoints: **`/root`** for health checks and **`/upscale POST`** for frame submission. When it receives an uploaded frame, the server performs several operations: it saves the image temporarily, loads it into memory using OpenCV, applies the Real-ESRGAN model for upscaling, and saves the upscaled output. An asynchronous lock, **`model_lock,`** protects the model during inference. Once upscaling is complete, the server streams the upscaled frame back to the client and schedules background cleanup.

```
try:
    async with model_lock:
        img = cv2.imread(input_path, cv2.IMREAD_UNCHANGED)
        output, _ = tiler.enhance(img, outscale=4)
        cv2.imwrite(output_path, output)
except Exception as e:
    return JSONResponse(status_code=500, content={"error": str(e)})
```

`video_upscale_client.py` is responsible for managing the upscaling pipeline. **`ffmpeg`** first breaks down video into individual frames, preserving sequence integrity and frame rate. Each frame is sent to the server via **`/upscale, and the`** server output is saved locally before being reassembled using **`ffmpeg`** again.

```
def upscale_frame(frame_path, output_path, server_url):
    with open(frame_path, 'rb') as f:
        files = {'file': f}
        response = requests.post(f'{server_url}/upscale', files=files)
        If response.status_code == 200:
            with open(output_path, 'wb') as out_file:
                out_file.write(response.content)
        else:
            raise    Exception(f"Failed    to    upscale    {frame_path}:
{response.text}")
```

3.3.2.2 Daemon and Docker Architecture

Scalable deployment of AI workloads within the HCN requires precise, architecture-aware system design. The Real-ESRGAN AI server is containerised and deployed as a Kubernetes-managed service to ensure platform compatibility, resilience, and controlled resource allocation across heterogeneous nodes.

The container image, built using the **`run-ai.sh`**, is based on **`nvidia/cuda:12.2.0-cudnn8-runtime-ubuntu22.04`**, providing a lightweight CUDA 12.2 and cuDNN 8 environment optimised for deep learning inference. Key packages installed include:

- PyTorch with CUDA support
- OpenCV for image processing
- FastAPI and Uvicorn for serving the model
- Real-ESRGAN libraries and dependencies

Deployment is defined in the **`real-esrgan-deployment.yaml`**:

- **Node Selection** – Pods are scheduled onto nodes labelled **`gpu=high`** and **`cuda=true`**, ensuring access to capable GPUS.
- **Resource Requests** – Each pod requests one full GPU (**`nvidia.com/gpu:1`**) with CPU and memory limits to maintain predictable performance.
- **Internal Service Exposure** – Services are exposed internally via **`ClusterIP`**, allowing secure access to the AI server without direct pod exposure.

During deployment, image registry addresses are patched to reflect the current network environment, enabling flexibility in dynamic home network setups.

Chapter 4: **Evaluation**

Evaluation of the HCN focuses on three primary aspects: system functionality, cluster performance, and behavioural analysis. The system was deployed across heterogeneous home devices and tested under real-world conditions to assess whether it met its design objectives.

4.1 Objectives

Evaluate whether the HCN meets its functional, performance, and sustainability goals when deployed under real-world conditions. The evaluation tests how effectively the system distributes workloads across heterogeneous home devices, adapts to changing resource availability, and achieves measurable benefits in resource efficiency and responsiveness. The key objectives to evaluate the HCN design conformance are:

- **Validate System Functionality:**
  Confirm and analyse if the HCN reliably orchestrates distributed workloads, including correctly applying dynamic and static node labels and taints.
- **Measure Performance Metrics:**
  Quantify performance metrics like latency and throughput in representative workloads and assess how the system responds to idle conditions, resource churn, and scheduling decisions.
- **Assess System Behaviour and Stability:**
  Observe pod startup latency, job completion times, and fault tolerance mechanisms during typical and stressed conditions using both real-world use cases and stress testing tools like kube-burner.
- **Verify Kubernetes Conformance:**
  Use Sonobuoy to validate that the cluster behaves as a conformant Kubernetes environment, despite modifications such as dynamic labelling and heterogeneous hardware.
- **Evaluate Environmental Impact:**
  Determine how much the HCN reduces reliance on dedicated compute infrastructure and contributes to energy-efficient task execution.

4.2 Methodology

4.2.1 Hardware and Software Environment

The HCN was deployed across five heterogeneous devices representing a realistic mix of architectures, performance levels, and network conditions in a typical home environment. Reflecting the goal of leveraging existing, idle computing hardware, some devices represent new systems in frequent use, some older and only in the HCN with 24/7 uptime.

Device Type	Architecture	Hardware	Network Interface
delane-pc High Performance PC	WSL2: amd64, cuda, Vulkan, OpenGL	Nvidia RTX 3070, Intel i5-12600K	Ethernet
delane-laptop Modern Laptop	WSL2: amd64, OpenCL, Vulkan, OpenGL	Intel Core Ultra 5 125U	Wi-Fi 6
dev1 Older Laptop	amd64, OpenCL, Vulkan, OpenGL	AMD Ryzen 5 4500U	Wi-Fi 4
control-node Raspberry Pi 4	arm64	Quad-core SBC	Ethernet
dev2 Raspberry Pi 3	arm64	Low-memory SBC	Wi-Fi (b/g/n)

Devices ran `Ubuntu 24.04 LTS` to ensure platform consistency, and Kubernetes v1.32.1-5 was deployed via kubeadm, with Docker as the container runtime. The control node in this deployment is equipped with a 250 GB SSD mounted over USB, powering the local registry and cache. Static and dynamic labelling/tainting worked as expected, with the system tainting nodes corrected when the computer is in use.

- Several monitoring and benchmarking tools were integrated into the system:
- Prometheus (custom configuration) was used to scrape node-level metrics such as CPU utilisation, memory availability, and pod lifecycle events.
- Sonobuoy was used to run Kubernetes conformance tests and ensure baseline cluster compliance.
- kube-burner generated synthetic workloads to evaluate system responsiveness and throughput under high scheduling pressure.

4.2.2 Benchmarks and Metrics

Three categories of metrics guided evaluation: use case performance, system-level behaviour, and sustainability impact. These were selected to align directly with the evaluation objectives described in Section 4.1.

4.2.2.1 Use Case Performance

Evaluation metrics for the two use cases are as follows:

AI Upscaling:	**Distributed Compilation:**
Frame Latency (seconds per frame)	Total Build Time
Total Processing Time for batch jobs	Parallelism Efficiency (comparison with local-only compilation)
Throughput (frames per second	Architecture-Aware Scheduling performance

Inference latency and job duration were recorded using Prometheus metrics and timestamps embedded in client-side scripts. Throughput was derived from the number of successfully processed frames or compiled files per unit time.

4.2.2.2 System-Level Metrics

To assess responsiveness, scalability, and robustness, the following system behaviours were measured:

- Pod Startup Time: Time from pod creation to running state
- Scheduling Delay: Time between job submission and node assignment
- Container Start Time: Time taken to pull and start containers
- Failure Recovery: Time to reschedule jobs after node loss
- Resource Pressure Events: CPU and memory saturation under load

Prometheus, kubectl event queries, and synthetic load tests using kube-burner were used to gather metrics.

4.3 Results

4.3.1 Distributed Compilation

Distributed Code compilation was evaluated to determine whether the HCN could effectively offload and accelerate CPU-bound development tasks. The selected benchmark involved compiling the source code for **`htop`**, a representative open-source C project, using make with the distcc wrapper. Two clients were tested: **`dev1`** and **`dev2`.** Compilation tasks were offloaded to **`delane-pc`**, running native and cross-compilation daemons.

Local compilation using dev1 took 63.2 seconds. When using **`distccd-amd64-s76xj`** running on **`delane-pc`**, build time averaged 57 seconds, a 28% reduction, an increase that easily outweighs the milliseconds on data transfer. Always ready, deamons showing their advantage. When load was artificially placed on delane-pc using **`CPU-Z`**, the node was effectively tainted, and the pod was successfully removed. When that load was removed, new pods were scheduled.

`dev2` compiled the same project natively in 132 seconds. By leveraging distccd-cross-km9jt running on **`delane-pc`**, the total build time was reduced to an average of 62 seconds, a near 2x increase. This demonstrated that low-power devices with limited local compute can significantly benefit from access to heterogeneous compute nodes elsewhere in the home network, provided toolchain compatibility and architecture-aware job routing are correctly managed.

- Failover behaviour was observed: when nodes were unreachable or no longer eligible, distcc automatically reverted to local compilation without error.

4.3.2 AI Upscaling

Nvidia cuda stands out as a GPU platform that is well supported and optimised on WSL2, with a difference of 2-7% being seen, where other platforms can see deltas of up to 70% or GPU emulation disguised as GPU drivers. The Real-ESRGAN AI Upscaling workload shows the potential of fully optimised WSL drivers to simplify workflows across platforms and networks. This use case tested the system’s label-based scheduling accuracy and ability to utilise GPU resources in a decentralised, idle-aware setup efficiently. However, due to the lack of CUDA resources in the evaluation HCN, full details on scalability cannot be gathered.

Building the Docker image for the AI use case ranges from 2.5 hours on the Raspberry Pi 4 node to ~20 minutes on the **`delane-pc`** – the former likely due to the poor performance of cross compilers. Once the image is built and stored on the repository, however, pod spin-up time is ~30 seconds, and due to scheduling logic, it is regularly scheduled on nodes for on-demand use. The task was scheduled exclusively to the **`delane-pc,`** the only node with CUDA support and sufficient GPU performance.

In the default configuration, the model was run in FP32 precision with tiling enabled (480×480 image split into four patches). 205 frames were processed in approximately 178 seconds, resulting in a throughput of ~0.87 seconds per frame. This was consistent across 5 test runs. This result aligns with external benchmarks for tiled, FP32 Real-ESRGAN execution on similar GPUS. While acceptable for offline or asynchronous upscaling tasks, the frame rate is too low for real-time applications. No frames failed or timed out during testing, and container resource limits were not exceeded. GPU utilisation peaked at around 60–70%, indicating that the workload was not saturating the RTX 3070's full potential. The tiled inference strategy most likely caused bottlenecks, single-frame HTTP dispatch (no batching), and model execution in complete precision.

Inference latency remained predictable and low once the image was cached locally. Combined with the dynamic scheduling logic, this made the system well-suited for passive, opportunistic AI processing tasks on idle hardware.

4.3.3 Kubernetes Behaviour

The static and dynamic labelling system proved responsive yet stable, avoiding unnecessary API noise and ensuring that scheduling decisions reflected real-time device conditions.

kube-burner evaluated pod startup latency, deploying repeated bursts of 20 pods under varying cluster conditions. The average time from job submission to container readiness:

Node	Mean Startup Time (seconds)
delane-pc	~8.2
dev1 / delane-laptop	~11.7
Raspberry Pi 4	~19.4
dev 2 – Pi 3	~35

These differences were attributed to hardware throughput, disk I/O speed, container runtime behaviour, and image caching. All nodes responded reliably under load, and no scheduling failures were observed during stress conditions.

Node readiness at the infrastructure level also varied - **`delane-pc`** and **`dev1`** typically became schedulable within 1 minute of boot, **`dev2`** took longer to initialise. At initial setup readiness time for the Pi ranged between 45 and 80 minutes, with frequent occurrences of hitting 1-hour test limits. Highlighting the importance of medium-high-performance manufactured products, low-end devices have shorter lifecycles and delays resulting from slower hardware.

4.3.3.1 Node Dropout and Resilience

Devices were manually disconnected from the network to test fault tolerance, put to sleep, or suspended mid-execution. Kubernetes marked them as NotReady within 20–30 seconds and excluded them from future scheduling. Jobs already in progress were either re-queued or reattempted elsewhere, and no tasks became orphaned or stranded. Upon reconnection, node labels persisted or were refreshed automatically by the telemetry script, restoring their eligibility for scheduling.

Across all conditions tested, Kubernetes maintained consistent behaviour and scheduling integrity. Despite being designed for large-scale cloud deployments, it proved adaptable to the more volatile, lower-powered, and diverse conditions of home infrastructure. Dynamic labelling, taints, and self-healing node management confirmed its suitability for sustainable, decentralised compute coordination. See Appendix 4 for more details on software initialisation and flow.

4.4 Environmental Impact

The HCN aimed to reduce the environmental impact of computing by repurposing idle hardware, avoiding embodied emissions from new devices, and minimising unnecessary energy usage. While a full lifecycle analysis (LCA) was outside the project's scope, reliable industry figures were used to estimate the environmental value of the system.

Most emissions from electronic devices occur during production. By reusing two older laptops instead of replacing them, the HCN avoided an estimated 500–600 kg CO_2e in manufacturing emissions — the equivalent of driving a petrol car ~1,600 miles. These savings scale linearly with device reuse, reinforcing the sustainability case for extending hardware lifespan.

Jobs were only scheduled for idle and powered devices, avoiding unnecessary energy expenditure. Unlike cloud workloads, which often consume hundreds of watts per task, the HCN ran on passively available capacity. Even GPU jobs were executed only while the RTX 3070 node was on and unused. The Raspberry Pi nodes added minimal overhead (~2–7 W), and no device was powered up solely for computing.

Workloads were constrained to idle periods and excluded from laptops on battery. Write-heavy jobs were filtered out on SSD-only nodes, helping reduce wear. Low-memory nodes like the Pi 3 were excluded from AI workloads entirely, protecting them from crashes or premature ageing. No device failed or degraded during the evaluation period. More detail is seen in Appendix 5.

4.5 Discussion

4.5.1 Interpretation and Limitations

4.5.1.1 Windows Subsystem for Linux

WSL2 allowed Windows machines to participate in the HCN without requiring dual-booting or full Linux installs. However, integration introduced significant platform-specific complications and dependencies. GPU support varied across vendors: while NVIDIA's CUDA stack performed reliably with only minor degradation (~2–7%) compared to native Linux, AMD and Intel GPUS were unsupported or misrepresented due to fallback drivers.

In some cases, WSL2 nodes incorrectly reported **`vulkan=true`** via emulated backends, triggering false-positive labels that had to be manually filtered or excluded.

System services were another friction point, and WSL2's lack of full emulation of Linux services left areas with issues. Networking issues, such as broken hostname resolution, required manual bridging using PowerShell scripts. These workarounds created a fragile onboarding experience that undermined automation and reproducibility.

4.5.1.2 Use Case Limitations

Distcc provided a lightweight mechanism for distributed compilation to offload jobs to idle nodes. However, the necessity for architecture-matching and the lack of Kubernetes-native integration constrained its applicability. This highlights the concern for the feasibility of the goal of generalising computing in a networked local way. A Kubernetes-native approach like what's seen in the stale Kubecc could solve these issues by treating compilation tasks as standard Kubernetes Jobs.

Real-ESRGAN successfully ran on the RTX 3070 node for AI upscaling with consistent output and accurate platform filtering. However, latency remained a core limitation: the model achieved ~0.87s per frame (1080p, FP32), suitable for offline batch processing but not real-time enhancement. Targeting ~30fps for interactive use would require ~30x higher throughput, achievable only through aggressive model pruning, precision reduction, or hardware acceleration. Unlike commercial AI upscaling in smart TVS, which prioritises real-time performance using low-power, quantised models on embedded NPUS or DSPS, the HCN approach focused on higher fidelity but delivered significantly lower throughput and integration. While the pipeline functioned reliably for proof-of-concept purposes, it does not serve as an appropriate comparison point to the stated aims.

4.5.2 Reproducibility and Future Work

The HCN was designed to be reproducible across commodity hardware with minimal cloud dependency. Most components were containerised and orchestrated via standard Kubernetes tooling, and all telemetry and labelling services were written as shell scripts or lightweight Python utilities. However, some barriers to reproducibility remain.

The biggest challenge lies in platform heterogeneity. Differences between WSL2, bare-metal Linux, and low-power devices like the Raspberry Pi led to inconsistent startup behaviour.

The system's reliance on software that did not take advantage of Kubernetes scheduler servers is a point to improve upon. Where limitations in logic are seen, deeper integration into Kubernetes can solve this.

System scalability was limited by its reliance on binary label logic (idle=true, gpu=high) without weighted prioritisation or ranking. The bluntness of this approach means that fine detail is often missed, with little inefficiencies adding up when performance is limited or scale is increased.

Memory pressure posed another constraint. Devices with limited RAM (e.g., Pi 4 with 4GB) frequently approached saturation during typical workloads:

```
MiB Mem :   3784.6 total,   278.6 free,   1945.2 used,   1757.6 buff/cache
```

Linux's caching model prevented immediate failure, but service responsiveness degraded. Distccd helpers stalled, telemetry daemons became unresponsive, and container operations froze. In some cases, pod startups failed silently. Timeouts occurred during repository pulls, metrics scrapes, and image extraction, slowing the entire network.

Lastly, the absence of a GUI or job dashboard limits accessibility. All system operations were performed via CLI tools. this excludes those unfamiliar with Kubernetes or Linux internals. A web-based dashboard would dramatically improve usability and support wider adoption.

Chapter 5: **Conclusion**

The Home Computing Network demonstrates a new model for sustainable computer system design. A decentralised, local network that effectively shares computing performance across a heterogeneous set of devices. Low-performance devices can borrow excess computing capacity within the home. Through the static and dynamic labelling and tainting system – extending existing capabilities in Kubernetes – the HCN can effectively identify idle computers and share excess capacity on the network. The system can adapt to the various processor architectures seen within the home and the varying availability of devices. Evaluation shows that adapting professional cloud-native technologies to home environments can produce results. When static labels effectively correspond to real performance, they can reduce the complexity of scheduling logic. HCN meets its core goals. The use cases, distributed code compilation and AI media upscaling, showed the ease of scheduling based on current resource availability. Stress testing with kube-burner confirmed scalability under load, while Sonobuoy validated Kubernetes conformance, indicating that the system maintains reliability even in non-traditional deployments. Only by scheduling on already-powered devices that meet idleness thresholds does the system avoid unnecessary energy expenditure, aligning well with its environmental objectives.

Limitations in the design, as well as software support, reduced the effectiveness of the system. Inconsistencies in GPU driver support in WSL limit what the HCN can do – reliance on hardware vendors to open-source code or provide drivers means performance is lost. Due to variable network and device conditions, real-time inference workloads such as AI-based upscaling with strict latency requirements were difficult to manage across a distributed setup. Moreover, the system currently lacks a user-friendly GUI, restricting accessibility for non-technical users. Limitations in the software support and consistency mean the HCN is hard to expand. But strong results in the CUDA benchmark highlight that CPU & GPU performance can be equally strong.

The HCN provides a compelling model for sharing computing processing across the home. Extending the lifecycle of devices with genuine on-network use proves that the manufacture of new hardware isn’t strictly necessary. Several opportunities for extension of the system do arise, such as modification of existing software to work on the HCN with minimal complexity, which means devices have more use. Improvements to user experience, such as developing a lightweight GUI, can expose the system to a wider user base.

The HCN showcases how distributed systems thinking can be applied within the home to provide sustainable functional use. By aligning with current environmental goals and leveraging open-source tooling, this project contributes a working system and a model for future exploration in sustainable computing.

References

Designing Sustainable Computer Systems With An Architectural Carbon Modeling Tool, 2022. The 49th Annual International Symposium on Computer Architecture (ISCA'22), p. 16, New York, NY, USA.

Lindsay, D., Gill, S.S. & Smirnova, D., 2021. The evolution of distributed computing systems: from fundamental to new frontiers. Springer Nature.

Pramanik, P.K., Pal, S. & Mukhopadhyay, M., 2024. Sustainable edge computing with mobile crowd computing: a proof of concept with a smart HVAC use case. Springer Nature.

R, R.R. & S, T., 2024. Decentralised Home Computing Network: Leveraging IaaS, PaaS, and SaaS for Passive Income Generation with Kubernetes Clusters. Third International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT), pp. 1-6, Trichirappalli, India.

Xu, M., Toosi, A.N., Bahrani, B., Razzagh, R. & Singh, M., 2019. Optimised Renewable Energy Use in Green Cloud Data Centers. Springer Nature.

Zheng, C., Tang, H., Zang, M., Hong, X., Feng, A., Tassiulas, L. & Zilberman, N., 2023. DINC: Toward Distributed In-Network Computing. New York, NY, USA: Association for Computing Machinery.

Apple, 2019. Product Environmental Report: iPhone 11 Pro Max. [online] Available at: https://www.apple.com/environment/pdf/products/iphone/iPhone_11_Pro_Max_PER_Sept2019.pdf [Accessed 16 October 2024].

Barclays Research, 2024. Artificial Intelligence is hungry for power. [online] Available at: https://www.ib.barclays/our-insights/3-point-perspective/AI-power-energy-demand.html#:~:text=to%20the%20newsletter.-,1.,according%20to%20Barclays%20Research1. [Accessed 16 October 2024].

Edler, A.S.A. & T., 2015. On global electricity usage of communication technology: trends to 2030. Challenges, 6(1), pp. 117–157.

Masanet, E., Shehabi, A., Lei, N., Smith, S. & Koomey, J., 2020. Recalibrating global data center energy-use estimates. Science, 367(6481), pp. 984–986.

Gupta, U., Elgamal, T., Hills, R., Wei, G.-Y. & Hsien-Hsin S. Lee, 2022. ACT: Designing sustainable computer systems with an architectural carbon modeling tool. New York, NY, USA: Association for Computing Machinery.

Gupta, U., Wu, C.-J., Hsien-Hsin S. Lee, H. & Koomey, J., 2021. Chasing Carbon: The Elusive Environmental Footprint of Computing. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 854–867.

Harris, J., 2005. Emerging third world powers: China, India and Brazil. Race & Class, 46(3), pp. 7–27.

Huang, A.H., 2009. A model for environmentally sustainable information systems development. Journal of Computer Information Systems, 49(4), pp. 114–121.

Interaction Design Foundation - IxDF, n.d. What is Sustainable Design? [online] Available at: https://www.interaction-design.org/literature/topics/sustainable-design [Accessed 13 October 2024].

Zhang, Y., Lan, X., Ren, J. & Cai, L., 2020. Efficient Computing Resource Sharing for Mobile Edge-Cloud Computing Networks. IEEE/ACM Transactions on Networking, 28(3), pp. 1227–1239.

Aras, E., Ammar, M., Yang, F., Joosen, W. & Hughes, D., 2020. MicroVault: Reliable Storage Unit for IoT Devices. DCOSS 2020. IEEE, pp. 132–139.

Gupta, U., Elgamal, T., Hills, R., Wei, G.-Y. & Lee, H.-H. S., 2022. ACT: Designing sustainable computer systems with an architectural carbon modeling tool. ISCA'22. ACM.

Kinghorn, D., 2020. Does Enabling WSL2 Affect the Performance of Windows 10 Applications? Puget Systems. [Online] Available at: https://www.pugetsystems.com/labs/hpc/does-enabling-wsl2-affect-performance-of-windows-10-applications-1832/

R, R.R. & S, TA., 2024. Decentralised Home Computing Network: Leveraging IaaS, PaaS, and SaaS for Passive Income Generation with Kubernetes Clusters. ICEEICT 2024, pp. 1-6.

Apple Inc., 2021. Product Environmental Report: 14-inch MacBook Pro. [pdf] Available at: https://www.apple.com/environment/pdf/products/notebooks/14-inch_MacBook_Pro_PER_Oct2021.pdf [Accessed 1 May 2025].

Foxway, 2023. The Carbon Handprint of Reused Laptops. [pdf] Available at: https://www.foxway.com/wp-content/uploads/2024/05/handprint-report-laptops-2023-eng.pdf [Accessed 1 May 2025].

HP, 2024. Product Carbon Footprint Results: HP Laptop 14s-fq0xxx. [pdf] Available at: https://h20195.www2.hp.com/v2/GetDocument.aspx?docname=c08216185 [Accessed 1 May 2025].

European Environmental Bureau (EEB), 2020. Coolproducts don't cost the Earth: Extending the lifetime of notebook computers. [online] Available at: https://eeb.org/library/coolproducts-dont-cost-the-earth/ [Accessed 1 May 2025].

Get Online @ Home, 2023. The Carbon Impact of Manufactured vs Refurbished Computers. [online] Available at: https://www.getonlineathome.org/2023/11/22/the-carbon-impact-of-manufactured-vs-refurbished-computers/ [Accessed 1 May 2025].

The Restart Project, 2022. The Environmental Impact of Our Devices: Revealing What Many Companies Hide. [online] Available at: https://therestartproject.org/consumption/hidden-impact-devices/ [Accessed 1 May 2025].

From Insight to Impact, 2023. Building a Sustainable Edge Computing Platform for Smart Homes. [pdf] Available at: https://www.sciencedirect.com/science/article/pii/S2773167722000115 [Accessed 1 May 2025].

Appendix

Appendix 1: Benchmark Comparison – WSL2 vs Native Linux

Test Objective:
To compare the compression performance of WSL2 against native Linux on the same hardware platform.

Methodology:
The Phoronix Test Suite was used with the pts/compress-7zip-1.11.0 module. Both environments were idle, and three trials were run in each environment using identical 7-Zip compression tests.

Results Summary:

Environment	Run 1	Run 2	Run 3	Average	Deviation (%)
Native	41107	41802	41584	41598	0.48
WSL2	39366	37853	38362	38527	2

Performance Difference:
Native Linux outperformed WSL2 by approximately 7.4% in this benchmark.

Appendix 2: Software Setup flow: detailed overview

Node Orchestration

On each node, setup begins with init.sh, which installs Docker, Kubernetes (via **`kubeadm`**, **`kubelet`**, **`kubectl`**), and **`cri-dockerd`**. It detects WSL2 environments and, if found, modifies **`/etc/wsl.conf`** to enable **`systemd`**, disables swap, and opens required ports via **`iptables`**. On native Linux, equivalent access rules are configured using **`ufw`**.

The node then connects to the cluster using join-hcn.sh, which fetches and runs the Kubernetes **`join-command.sh`** over SSH and pulls the **`kubeconfig`** from the control plane. Optionally, **`join-storage.sh`** mounts an NFS volume shared by the control node at **`/mnt/shared_drive`** and configures Docker to trust the local container registry exposed on port **`30000`**.

Hardware Profiling and Labelling

GPU capability detection is handled by setup-drivers.sh, which labels nodes based on available platforms: **`cuda=true`**, **`vulkan=true`**, **`opencl=true`**, and/or **`opengl=true`**. In WSL2, GPU support is installed according to vendor (NVIDIA or Intel), including Vulkan (**`Dozen`**) and OpenCL drivers.

Next, **`node-perf.sh`** benchmarks each node using the **`Phoronix Test Suite`**. Tests are selectively run based on GPU support and include:

- build-linux-kernel for CPU
- vkmark for Vulkan

- octanebench for CUDA
- juliagpu for OpenCL
- unigine-heaven for OpenGL.

These results are parsed by static_labelling.py, which assigns discrete performance labels (e.g., `cpu=high`, `vulkan-perf=mid`) based on predefined thresholds. It also detects storage type (e.g., `storage=ssd`, `storage=virtual`) and whether the node has a battery (`has-battery=true`).

Dynamic Scheduling and Energy Awareness

To manage real-time scheduling, each node installs a dynamic labelling service that updates the idle label based on CPU usage and power conditions. On Linux, this is handled via `dynamic_labelling.py`, run as a systemd service by `setup-labelling.sh`. It continuously evaluates CPU load and battery status (via `psutil`) and updates Kubernetes node labels and taints accordingly:

- If idle and power-sufficient: `idle=true`
- Otherwise: `idle=false:NoExecute` taint applied

On WSL2, due to the lack of native battery telemetry, setup-wsl-labelling.sh installs a split pipeline using `monitor_status.py` (run in PowerShell) piped to `wsl_dynamic_labelling.py`, simulating equivalent logic in a hybrid manner.

5.1.1.1 Control Plane

The control plane is initialized via `init-control.sh`, which configures the Kubernetes master with `Calico` networking and generates a join token. If --net-drive is enabled, `setup-storage.sh` is invoked to configure an NFS server from a block device. It auto-detects the network range, updates `/etc/exports`, and deploys:

- A `PersistentVolume` and `PersistentVolumeClaim`
- A container `registry` exposed on `NodePort:30000`, backed by the shared volume

Appendix 3: Detail on System Functionality Analysis

Node Onboarding and Labelling
Each node was fully onboarded through hcn.sh, installing dependencies, joining the cluster, and triggering Phoronix Test Suite benchmarking. Based on results and driver support, each node was assigned static labels – cpu=high, gpu=low, or cuda=true. When jobs had specific label requirements, nodes with the appropriate labels were jobs, and pods were successfully scheduled.

Dynamic Behaviour
The core to leveraging idle computing was using long- and short-term underutilised hardware; dynamic labelling is the core to achieving this. A systemd service ran periodically on each device to report CPU load, GPU usage, and battery level (where applicable). When a node met the defined thresholds (e.g., CPU/GPU usage below 30%, battery > 80% or plugged in), it was dynamically labelled with idle=true and battery=true. These labels were removed when conditions changed, preventing job assignment to active or battery-limited nodes. This included WSL2 nodes, which don't natively report device utilisation. An edge case on low-storage devices triggered Kubernetes' native disk-pressure:NoSchedule taint. This highlighted the importance of taints in scheduling

control and reinforced the HCN's mechanism, which applied a custom idle=false:NoSchedule taint when utilisation thresholds were exceeded.
Fault Tolerance and Usability
Device availability varied during testing to simulate real-world usage. Nodes were disconnected, rebooted, or overloaded to observe system behaviour. Kubernetes responded correctly to all changes, marking unavailable nodes as NotReady or NoSchedule and excluding them from scheduling until they returned. Dynamic labels were refreshed upon reconnection, and no jobs were lost or misrouted because of node churn. A refresh period of 30 seconds and averaging load readings over 10 seconds meant that updates were not made overly frequently, which is positive for decreasing Kubernetes calls and unnecessary confusion in scheduling. This resilience confirmed the system's ability to operate reliably in a decentralised, semi-trusted environment.
Setup is 95% unattended, with a small prompt section at the start. Once set up, it is entirely out of the user's focus. The labelling system meant that jobs would not be scheduled onto incompatible platforms, which would slow down the workflow. All system services were entirely survivable and provided logging in case anything went wrong, and the user had high technical knowledge.

```
○ dynamic-labelling.service - Dynamic Node Labelling Service (WSL2)
     Active: inactive (dead) since Thu 2025-05-01 11:17:38 BST; 2s ago
May 01 11:17:37 delane-laptop systemd[1]: Started Dynamic Node Labelling
Service (WSL2).
psutil
May    01    11:17:38    delane-laptop    dynamic-labelling.sh[2099210]:
ModuleNotFoundError: No module named 'psutil'
```

Appendix 4: Environmental Metrics and Sources

Embodied Emissions:
Apple and HP report that 74% of a laptop's total CO_2e footprint is embodied in manufacturing. Foxway (2023) states a typical new laptop emits ~265 kg CO_2e, while refurbishing emits only ~11 kg CO_2e. Thus, reuse avoids ~254 kg CO_2e per device — a ~96% reduction.

Sources:
- Apple (2021). Environmental Progress Report
- HP (2024). Product Carbon Footprint Summary
- Foxway (2023). Carbon Handprint of Device Reuse
- EEB (2020). Coolproducts Don't Cost the Earth

Energy Use Comparison:
Cloud GPUs often consume 200–400 W continuously under load (Insight to Impact, 2023). The RTX 3070 node used in the HCN added ~88 W only during inference, and only when idle. Pi nodes used <7 W each.

Sources:
- From Insight to Impact (2023). Carbon Metrics in ML Workloads
- Local power draw measured using a smart plug (TP-Link Kasa)

Device Health Practices:
- idle=true label enforced CPU/battery thresholds
- Laptops excluded on battery
- Write-heavy tasks avoided on SSD-only nodes
- Pi 3 excluded from heavy jobs due to <1 GB usable RAM
- No SMART data collected, but no wear or failure observed during testing