

Project Title	Online Data Intensive Solutions for Science in the Exabytes Era
Project Acronym	ODISSEE
Grant Agreement No.	101188332
Start Date of Project	01/01/2025
Duration of Project	36 Months
Project Website	odissee-project.eu

## D4.1 – Software Framework Requirements Capture & Integration Plan

Work Package	<b>WP4 – Software Framework</b>
Lead Authors (Org)	<b>Jaume Moragues (BSC)</b>
Contributing Author(s) (Org)	<b>Eduardo Quinones (BSC) Sara Royuela (BSC) Bernat Xandri (BSC) Ayman Bourramouss (BSC)</b>
Due Date	<b>30.06.2025</b>
Date	<b>17.07.2025</b>
Version	<b>Final</b>

### Dissemination Level

<input checked="" type="checkbox"/>	PU: Public
<input type="checkbox"/>	PP: Restricted to other programme participants (including the Commission)
<input type="checkbox"/>	RE: Restricted to a group specified by the consortium (including the Commission)
<input type="checkbox"/>	CO: Confidential, only for members of the consortium (including the Commission)

## Versioning and contribution history

Version	Date	Author	Notes
0.1	16.05.2025	Jaume Moragues (BSC), Eduardo Quinones (BSC) Sara Royuela (BSC) Bernat Xandri (BSC) Ayman Bourramouss (BSC)	Version circulated to consortium
1.0	17.07.2025	Jaume Moragues (BSC), Eduardo Quinones (BSC) Sara Royuela (BSC) Bernat Xandri (BSC) Ayman Bourramouss (BSC)	Version taking into account comments from the consortium
Final	21/07/2025	Damien Gratadour (CNRS, Observatoire de Paris), Clémentine Ferré (Neovia Innovation)	Final edition for submission

### Disclaimer

This document contains information which is proprietary to the ODISSEE Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to a third party, in whole or parts, except with the prior consent of the ODISSEE Consortium.

AI tools may have been used to produce this deliverable. The ODISSEE consortium is ultimately responsible for its content and ensured that every AI generated content has been proof-read and checked if necessary. AI tools have been used within the boundaries of existing laws, in particular regarding the respect of privacy, confidentiality and Intellectual Property Rights.

## Table of Contents

Versioning and contribution history .....	2
Table of Contents .....	3
List of Figures .....	3
Executive Summary .....	4
1. Introduction .....	5
2. Use cases .....	5
2.1. LHCb .....	6
2.2. SKAO .....	7
3. Tools and frameworks .....	8
3.1. COMPSs .....	8
3.2. OpenMP .....	10
3.3. FirecREST .....	11
3.4. Extrae/Paraver .....	12
3.5. EAR (EAS) .....	12
4. Hardware platforms .....	13
4.1. Rhea platform .....	14
4.2. Maverick platform .....	14
5. References .....	16

## List of Figures

Figure 1 Typical FirecREST context .....	11
Figure 2 Maverick2 1/4 die shot with various components .....	15

## Executive Summary

The goal of this document is to describe the best way of ensuring maximum resource usage, while taking into account certain aspects, like sustainability, of data processing pipelines in ODISSEE. In particular, we present the current status of platforms Allen (LHCb) and SKAO (including precursors). We describe possible programming models and tools in order to achieve such performance and finally we describe on which hardware platforms we aim to deploy such software stacks. This report places the relevant pieces (use cases, end users, tools & programming models and hardware) in such a way that it depicts the project's big picture and so facilitates proper technical decisions which shall lead to the fulfilment of project ODISSEE goals.

## 1.Introduction

---

Work Package (WP) 4 of the ODISSEE project will apply state of the art programming models to software infrastructure provided in [4] in order to improve performance and reduce power consumption. Of special interest are aspects like interoperability between the different components (see 3.2) that will take place in this work package (with special emphasis on IO and compute operations).

Indeed, we will deal with different kinds of hardware (from industrial partners like SiPearl and NextSilicon) and software (from use cases from CERN and ASTRON) so it is of great importance to ensure that chosen programming models shall work well on both use cases and so a deep analysis of Allen and SKAO codebase infrastructure must be conducted (mainly its current status of parallelization). In this regard it is of equal importance that the software components selected fit well in the hardware provided for this project (SiPearl, NextSilicon). As an output of this study, a list of requirements shall be created and software components (compiler and HPC runtime) shall be chosen.

On the other hand, it is of great importance that this work-package identifies properly the needs and constraints of the end users of these technologies. In particular, the ODISSEE project counts on

- Consulting bodies: These communities shall establish specifications of the forthcoming applications from this project. Generally speaking, they are more close to the science produced by those applications.
- Infrastructure bodies: These communities shall provide a set of requirements over the hardware and software being used by the applications used in this work-package. Generally speaking they are more close to the engineering team behind those applications.

This document has been divided into three main sections:

1. Use Cases: In this section we describe the use cases which are taken under consideration in this work-package as represent well all the goals we want to accomplish at the end of the ODISSEE project: high throughput, low latency, sustainability and evolution after project.
2. Tools and frameworks: in this section we describe the set of tools and frameworks chosen for the development of WP4 throughout the ODISSEE project.
3. Hardware: In the last section we do a high level description of the hardware that will be used within the ODISSEE project.

The order of these sections has been decided so we present first the nature and challenges of the applications that will be run, next which software approach we take to overcome those challenges and finally in which hardware those applications will be executed.

## 2.Use cases

---

This project will use LHCb and SKAO use cases as project-testing scenarios due to its demanding requirements both in input/output throughput and computation needs (see [4]). Indeed, one of the first and main motivations for this project has been to enhance handling very high volumes of data, both in terms of communications (not only in terms of data ingestion but as well in re-transmission) and computational capabilities. In particular, from a computational perspective, this work-package is devoted to offer

some programming paradigms that aims to ensure maximum usage of present resources taking into account aspects like sustainability<sup>1</sup>.

## 2.1. LHCb

---

The LHCb experiment is a research facility devoted to the study of high energy physics by colliding high energy beams, and so trying to reproduce extreme physical conditions that can lead to confirm some physical theories or searching for beyond the standard model particles (see [4]).

In this work-package, one of the use cases we will focus on is the Allen platform<sup>2</sup>, which is a software infrastructure in charge of studying beam collisions in near real time. In particular, the Allen platform is a software system capable of receiving beam collision data and detecting certain events that will trigger algorithm executions mapped to those events. It is because of this that we say that the Allen platform is an *event-based* system.

The main end users of this technology are high energy physicists interested in studying matter properties in extreme conditions. Since such scientists prefer to abstract their research from technical details regarding aspects like high throughput or hardware accelerators, this platform has been well divided into two main parts:

- A set of low-level kernels written in C++ and CUDA performing operations which are embarrassingly parallel. We define an algorithm as a simple enough kernel that may be run in an accelerator (or the CPU, according to some compilation flags).
- A set of high-level processing pipelines written in Python, composed of sequences of algorithms. Each algorithm in a pipeline is defined by a kernel and a set of input and output parameters, making it possible to detect dependencies among algorithms.

This division allow the science community to conduct research at a higher level and so without concerns about programming models or hardware infrastructure. On the other hand, the engineering team may focus only on software and hardware infrastructure while defining the Allen platform, and so abstracting it from the algorithms and its pipelines, with kernels being the only common point.

All the computationally intensive operations are run in accelerators (i.e., NVIDIA GPUs) due to their highly parallelizable nature, so we may infer that technical challenges reside in the volume of the incoming data, and no so much in the complexity of the kernels at each stage of the sequence. Although there is some CPU parallelization inside Allen codebase, it is not of significant importance.

According to all this, we can depict the algorithm sequences as linear graphs, in which each node is an algorithm, which contains kernels that shall be executed in a given accelerator (e.g., GPU). The main issue with this approach is the synchronization in the CPU memory space between each pair of nodes of this graph. At the beginning of every kernel execution, incoming data is offloaded to device memory, and, once it has finished, is taken back to host memory, becoming a bottleneck if the number of nodes of a given sequence is big enough.

---

<sup>1</sup> In the context of the ODISSEE project we assume the concept of sustainability by three dimensions: societal equity, environmental responsibility and economic efficiency.

<sup>2</sup> <https://gitlab.cern.ch/lhcb/Allen.git>

On the other hand, the Allen platform is provided with a set of resources controlled by a SCADA<sup>3</sup> system called *Experiment Control System* (ECS for short). Every time the event thread detects a new batch of events, each event is assigned to a CUDA stream in a *Round Robin* fashion, so they can be processed independently and in parallel. Since event batches have no dependencies among them heuristics on resource acquisition seem to be of little relevance here, however if new constraints such as sustainability are added it may be of some importance.

## 2.2. SKAO

The Square Kilometre Array Observatory (SKAO) is a next-generation radio telescope which intends to provide sensitive high-resolution images of the radio sky through radio interferometry, operating over a wide range of frequencies (see [4]).

Within WP4 we will focus on the Self-Calibration Pipeline, the component of the Science Data Processor (SDP) responsible for turning raw interferometric visibility into high-resolution images, suitable for scientific analysis. This is achieved by means of an iterative loop, which is composed (among others) of three main stages (see [4]):

- Calibrate: which solves direction-independent complex gains using the current sky model.
- Predict: which subtracts model visibility (outliers and/or bright sources) from the data.
- Image: which produces an updated version of the sky model.

In the SKAO prototype pipeline, this workflow is expressed as a two-layer programming model:

1. **Coarse-grain orchestration:** Written in Python and executed with a Dask-based task wrapper that handles the distribution of computational load under the hood, the high-level stages that compose the pipeline, previously described, run sequentially in a cycle fashion:

Calibrate → Predict → Image

The overall flow of the pipeline is described as a list of cycles through a YAML file, ensuring that every cycle contains a Calibrate and Image operation, while Predict remains optional.

These operations present strict data-dependencies, which are expressed through the exchange of deterministically named data-files, thus ensuring consistency between cycles. The execution of the pipeline must, therefore, wait for each operation to complete before moving to the next stage, ensuring that only fully written files are handed over.

<sup>3</sup> SCADA stands for Supervisory Control And Data Acquisition.

**Fine-grain orchestration:** The high-level pipeline delegates work to lower-level, high-performance C++ codes, relying on tools like DP3<sup>4</sup> or Quartical<sup>5</sup> for Calibration and Prediction, as well as WSClean<sup>6</sup> or DDFacet<sup>7</sup> for Imaging. In each cycle, for Calibration and Predict, the observation data can be split into time chunks and assigned to one of the pre-instantiated Distributed Dask Workers. Within each worker, intra-task parallelisms can be utilized if DP3 or Quartical are built with OpenMP, enabling multi-threaded execution. To speed up the Imaging stage, multiple nodes can also be used by running WSClean or DDFacet in distributed mode, which makes use of MPI to parallelize the imaging over multiple nodes, independently from Dask.

The SKAO self-calibration pipeline's clear separation between the orchestration at pipeline level, and the fine-grained kernels, as well as its file-based dependency mechanisms and configurable chunking parameters, offers a fitting test-bed for the exploration of advanced task-graph runtimes, heterogeneous accelerator back-ends and energy-aware scheduling policies.

## 3. Tools and frameworks

---

In the context of the ODISSEE project an efficient usage of existing resources is crucial. Indeed, in research infrastructure like SLICES<sup>8</sup> running in contexts like the cloud continuum, once a set of resources has been assigned to a given execution, we must ensure that it is used at its maximum capacity (improving efficiency). Moreover, if we want to add constraints, such as power consumption or load balance among nodes, smart workload orchestrators will be required.

As we saw in 1, there are two main end users for this project: Scientific bodies and Hardware infrastructure bodies, each of them working from more high level concepts (pipelines and algorithms) to more low level detail (accelerator code, workload distribution among connected nodes). Because of this, our approach is two-fold:

- A high-level Python framework capable of orchestrating low-level, accelerated code through tools such as COMPSs. This component is considered to be high level and suited for the scientific committees.
- Kernel definition made in C++ and decorated by OpenMP pragmas. This component is suited for the hardware committees since it deals with performance and hardware acceleration.

The rest of this Section provides more details about the specific programming models, frameworks and tools that will be used in ODISSEE.

### 3.1. COMPSs

---

COMP Superscalar (COMPSs) [2] is a task-based programming model which aims to ease the development of applications for distributed infrastructures, such as large High-Performance clusters, clouds and container managed clusters. COMPSs provides a programming interface for the development of the applications and a runtime system that exploits the inherent parallelism of applications at execution time.

---

<sup>4</sup> <https://github.com/lofar-astron/DP3>

<sup>5</sup> <https://quartical.readthedocs.io/en/latest/>

<sup>6</sup> <https://gitlab.com/aroffringa/wsclean.git>

<sup>7</sup> <https://github.com/saopicc/DDFacet>

<sup>8</sup> SLICES stands for Scientific Large-scale Infrastructure for Computing/Communication Experimental Studies



To improve programming productivity, the COMPSs programming model has the characteristics:

- **Sequential programming:** COMPSs programmers do not need to deal with the typical duties of parallelization and distribution, such as thread creation and synchronization, data distribution, messaging or fault tolerance. Instead, the model is based on sequential programming, which makes it appealing to users that either lack parallel programming expertise or are looking for better programmability.
- **Agnostic of the actual computing infrastructure:** COMPSs offers a model that abstracts the application from the underlying distributed infrastructure. Hence, COMPSs programs do not include any detail that could tie them to a particular platform, like deployment or resource management. This makes applications portable between infrastructures with diverse characteristics.
- **Single memory and storage space:** the memory and file system space is also abstracted in COMPSs, giving the illusion that a single memory space and single file system is available. The runtime takes care of all the necessary data transfers.
- **Standard programming languages:** COMPSs is based on the popular programming language Java, but also offers language bindings for Python ([5]), C/C++, and recently to R (RCOMPSs<sup>9</sup>). This makes it easier to learn the model since programmers can reuse most of their previous knowledge.
- **No APIs:** In the case of COMPSs applications in Java, the model does not require to use any special API call, pragma or construct in the application; everything is pure standard Java syntax and libraries. With regard the Python, C/C++ and R bindings, a small set of API calls should be used on the COMPSs applications.

PyCOMPSs/COMPSs can be seen as a programming environment for the development of complex workflows. For example, in the case of PyCOMPSs, while the task-orchestration code needs to be written in Python, it supports different types of tasks, such as Python methods, external binaries, multi-threaded (internally parallelised with alternative programming models such as OpenMP or pthreads), or multi-node (MPI applications). Thanks to the use of Python as programming language, PyCOMPSs naturally integrates well with data analytics and machine learning libraries, most of them offering a Python interface. PyCOMPSs also supports reading/writing streamed data.

At a lower level, the COMPSs runtime manages the execution of the workflow components implemented with the PyCOMPSs programming model. At runtime, it generates a task-dependency graph by analyzing the existing data dependencies between the tasks defined in the Python code. The task-graph encodes the existing parallelism of the workflow, which is then scheduled and executed by the COMPSs runtime in the computing resources.

The COMPSs runtime is also able to react to tasks failures and to exceptions in order to adapt the behaviour accordingly. These functionalities, offer the possibility of designing a new category of workflows with very dynamic behavior, that can change their configuration at execution time upon the occurrence of given events.

As we saw in Section 2.2, the usage of a Dask-based orchestration represents a great opportunity for introducing PyCOMPSs in this use case since it is a high level orchestrator. This could allow scientists to conduct research in HPC clusters just by decorating Python code with PyCOMPSs. Effort will be put on the integration of

---

<sup>9</sup> <https://arxiv.org/pdf/2505.06896>

PyCOMPSs with existing Dask-based existing orchestration. While at this point how this integration will be performed is uncertain, it will probably consist on using PyCOMPSs for high-level tasks that include Dask workflows inside.

In the case of LHCb, and in particular in the Allen platform, we see the way algorithms are expressed a potential entry point for the PyCOMPSs framework, being a risk how handcrafted is the resource acquisition.

## 3.2. OpenMP

OpenMP has been one of the *de facto* standards in the HPC ecosystem for the last decades as it offers:

- **Programmability:** It provides an easy way to express parallel behavior of software based on macros. At the same time, it offers a programmatic API that resembles the same capabilities than that based on pragmas. This point is key for the Scientific body since it offers an abstraction from parallel primitives in languages like C++. One of the goals of the ODISSEE project is to provide a software platform that may be used later on by scientific communities which may prefer to abstract their research from the theoretical aspects of parallelism, making OpenMP a great deal for this.
- **Interoperability:** OpenMP is a software standard implemented by many vendors and with a standard committee behind, all in all ensuring proper interoperability among main platforms. Apart from this, since we are working with OpenMP based on LLVM we can benefit from the usage of intermediate representations (MLIR), which by definition are abstract enough to avoid machine code (and so most of the platform caveats) and still express all the operations of the original code. As we saw in Section 2.1, some of the end users of this platform may run their applications in accelerators like GPUs, and OpenMP ensures compatibility with these hardware, for example, via CUDA translation or by LLVM-based MLIR code.
- **Performance:** Since OpenMP offers an abstraction from the system primitives it will always incur in some non-zero abstraction cost, and so there will always some performance degradation. However, through the usage of tasks, and even more with the taskgraph feature (officially shipped in OpenMP 6.0 version), we can alleviate profoundly this penalty for the kind of task-based parallel applications arising from the LHCb and SKAO use cases (see [1]).

As we explained in Sections 2.1 and 2.2, LHCb and SKAO are two sequential pipelines, where some parts of the computations rely on data parallelism. Regarding the Allen infrastructure, it is a massively parallel kernel sequence, where each kernel is a node in a linear graph. Currently, a major bottleneck in the Allen software platform is the data transfer between host and device memory at each node of the graph. We see here an opportunity for applying some OpenMP features, especially the taskgraph approach (see [1]). In addition, the interoperability OpenMP offers is of great importance to enable new, experimental accelerators to be tested and integrated, which is often needed in state of the art infrastructures such as that of the LHCb.

The challenges of using this programming model are mainly the existing coupling between the kernel definition and the resource assignment, making it certainly difficult to define the Allen software codebase agnostic from the resource orchestrator.

In the fine-grain orchestration of SKAO we see the opportunity to improve DP3 library parallelism by means of OpenMP directives since it supports OpenMP programming model.

### 3.3. FirecREST

FirecREST<sup>10</sup> is an open-source, lightweight REST API designed to provide seamless access to High-Performance Computing (HPC) resources through standard web protocols. Acting as a proxy, FirecREST offers a unified, standardized interface to HPC infrastructures, supporting user authentication and authorization, and integrating with a wide variety of schedulers, storage systems, and file system types.

With FirecREST, users can automate interactions with HPC systems, enabling a broad range of programmatic use cases such as CI/CD pipelines, workflow manager integration, automated data transfers, development of HPC web portals, etc.

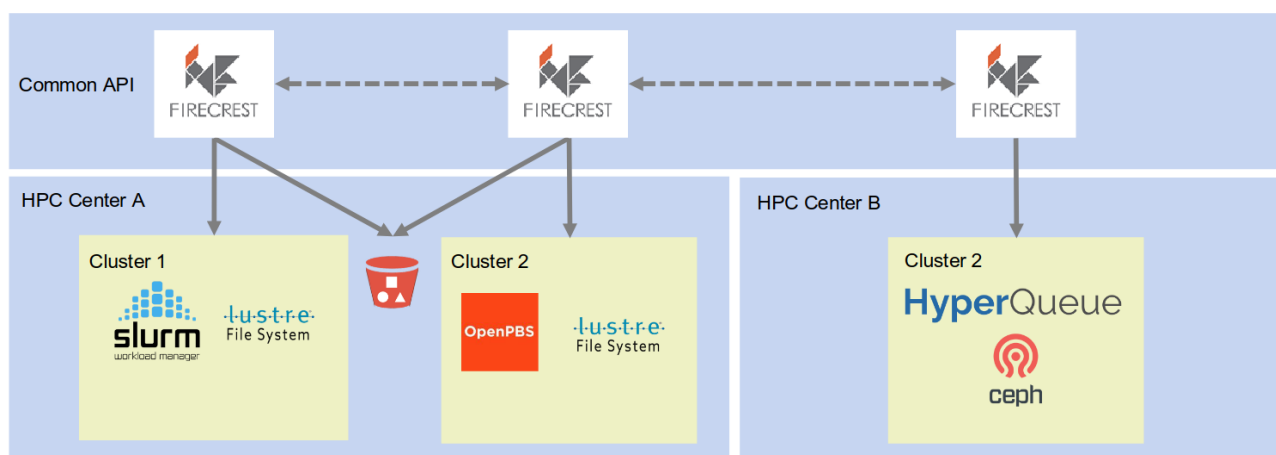


Figure 1 Typical FirecREST context

#### Key Features:

- **Authentication & Authorization:** A robust security layer built on OAuth 2.0<sup>11</sup> and OpenFGA<sup>12</sup> standards. FirecREST supports flexible, configurable authentication mechanisms that integrate with popular identity providers (e.g. Keycloak, Shibboleth, Okta, Azure Active Directory, Google Identity Platform). It also offers fine-grained access control via relationship-based policies or custom JWT claims.
- **Technology Abstraction:** FirecREST abstracts the complexities of underlying HPC technologies—including job schedulers, file systems, and storage backends—through a consistent API. This allows users to submit and monitor jobs, transfer data, and perform file system operations without needing in-depth knowledge of specific HPC environments. Furthermore it allows to run the same workflow across HPC sites and HPC technologies without making changes to the software.
- **High-Performance API:** Built on asyncIO<sup>13</sup> and utilizing an asynchronous SSH connection pool, the API is optimized for high-throughput operations, making it suitable for demanding, large-scale HPC workflows.

<sup>10</sup> <https://github.com/eth-cscs/firecrest-v2>

<sup>11</sup> <https://oauth.net/2/>

<sup>12</sup> <https://openfga.dev/>

<sup>13</sup> <https://docs.python.org/3/library/asyncio.html>

- **System Health Monitoring:** An integrated cluster health checker continuously monitors the availability and status of backend services, ensuring operational integrity and protecting users from unreliable or failing systems.

As part of the ODISSEE project, FirecREST enables federated access to distributed HPC resources. Its modular architecture supports integration with diverse identity providers, schedulers, file systems, and data transfer protocols. Future enhancements include support for EAR (Energy-Aware Resource management).

Through FirecREST, COMPSs will streamline the access of HPC resources across multiple sites while guaranteeing high throughput and efficient resource utilization.

### 3.4. Extrae/Paraver

---

Extrae<sup>14</sup> is a dynamic instrumentation package capable of tracing programs compiled and run with the shared memory model (like OpenMP and pthreads), the message passing (MPI) programming model or both programming models (different MPI processes using OpenMP or pthreads within each MPI process). It is currently available on different platforms and operating systems: IBM PowerPC running Linux or AIX, and x86 and x86-64 running Linux. It also has been ported to OpenSolaris and FreeBSD.

Paraver is a flexible visualization and analysis tool based on an easy-to-use wxWidgets GUI. It was developed responding to the need of having a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Its clear and modular structure plays a significant role towards achieving these targets.

Because Extrae generates trace files that can be visualized with Paraver, the combined use of Extrae and Paraver offers an enormous analysis potential, since it provides a large amount of information useful to decide the points on which to invest the programming effort to optimize an application. With these tools the actual performance bottlenecks of parallel applications can be identified. The microscopic view of the program behavior that the tools provide is very useful to optimize the parallel program performance.

### 3.5. EAR (EAS)

---

The Energy Aware Runtime (EAR)<sup>15</sup> package is a comprehensive energy management framework for HPC and AI data centres, offering three main services—**monitoring**, **optimisation**, and **powercap**—to enhance energy efficiency and system stability. These services are delivered by five core components: the EAR Library (EARL), acting as Job Manager for runtime monitoring and dynamic optimisation; the EAR Daemon (EARD), serving as Node Manager to enforce frequency and power settings; the EAR Database Manager (EARDDB), which buffers and aggregates monitoring data; the EAR Global Manager (EARGMD), coordinating powercaps across the cluster; and the Energy Data Centre Monitor (EDCMON), which complements EARGMD for a more holistic approach to data centre management.

- **Monitoring:** EAR provides detailed energy and performance monitoring across compute nodes, jobs, and applications, extending to non-computational devices and infrastructure. EARL collects real-time application metrics by loading

---

<sup>14</sup> <https://tools.bsc.es/doc/pdf/extrae.pdf>

<sup>15</sup> <https://github.com/eas4dc/EAR>

automatically alongside applications thanks to its seamless integration with workload managers like SLURM, PBS, or PyCOMPSs. At the node level, EARD gathers additional data such as device frequencies and power usage, forwarding it efficiently via EARDBD, which reduces direct database load. EAR supports multiple reporting plugins compatible with relational databases (e.g., MariaDB, PostgreSQL) and non-relational systems like EXAMON. EARGMD and EDCMON expand monitoring to include cooling systems, thermal limits, and AC power, offering a holistic energy profile of the entire data centre.

- **Optimization:** EAR's lightweight optimisation dynamically improves efficiency by adjusting CPU, memory, and GPU frequencies based on real-time application and node data. EARL analyses metrics during execution to select optimal frequencies, which EARD enforces at the node level. This operates transparently, requiring no code changes or manual tuning. Integration with PyCOMPSs is also being enhanced to support advanced energy-aware workload scheduling.
- **Powercap:** EAR ensures safe and efficient operation through robust power capping at node and cluster levels. EARGMD manages cluster-wide power caps with a scalable, distributed approach, while EARD enforces local caps via frequency adjustments. EDCMON adds thermal cap management and dynamic cooling monitoring for groups of nodes, ensuring thermal and power constraints are maintained. The system is configurable to adapt policies dynamically or simply monitor global energy use.

EAR is highly configurable and supports a wide range of architectures<sup>16</sup>. For application monitoring, it can even function under installations without privileges.

## 4. Hardware platforms

Another variable in which we ground this work-package is the hardware which will be used to run the use cases exposed in Section 2. As we saw in Section 2, the use cases expose high amounts of parallelism, so it is mandatory to consider highly parallel hardware platforms.

Taking for example the case of the Allen platform, we see that in presence of GPU accelerators, kernels are scheduled to be executed on these devices, with all its benefits and disadvantages. While executing highly parallel code in a GPU accelerator may present dramatic improvement compared to the same execution on a typical CPU, there is a drawback in terms of data movement from host to device memory.

In the case of SKAO, as we saw in Section 2.2, this system relies on distributed computation among target devices, so the selected hardware platform shall ensure high throughput and low latency while offering support for the *de facto* standards for distributed computing such as MPI.

Finally, as we exposed in Section 1, sustainability is one of the key goals of this work-package and so is mandatory that the hardware platform we use for running the use cases listed in Section 2 offer a rich enough API that allows us to track certain measures we may use to ensure aforementioned sustainability.

---

<sup>16</sup> <https://github.com/eas4dc/EAR>

## 4.1. Rhea platform

---

The Rhea1 and Rhea2 platforms are two SiPearl arm-based processors holding multiple Neoverse V1 cores well suited for connecting to external accelerators. With the usage of HPC tools like OpenMP, we ensure workloads not suited for accelerators are mapped to compute units in an optimal way. On the other hand, this processor offers a high bandwidth memory (HBM), which will bring overall acceleration as it will improve efficiency of part of the workload running on CPU.

So in short, for the Rhea1 platform we find the following features:

- Processor with many neoverse V1 cores.
- 2x256 SVE (Scalable Vector Extension) per Neoverse V1 core.
- High throughput IO.
- Provide a better byte/flop ratio compare to other CPUs.
- Any Linux distribution will be supported, Debian, RHEL based.
- Mainstream HPC tools (such as OpenMP).
- Open API, suitable for tracking measures like power consumption.
- Provision of Performance provision units (PMUs) for energy consumption.

As exposed in Section 3, using tools such as OpenMP and COMPSs we shall ensure optimal usage of the arm cores, while adding little overhead thanks to statically computed dependency graphs. We shall offload those tasks considered to be highly embarrassingly parallel among data to accelerators through exactly the same API.

## 4.2. Maverick platform

---

The Maverick-2 platform is a hardware accelerator that offers a flow-based approach to the massively parallel applications problem. The Maverick accelerator is composed of a grid of 4 computation clusters:



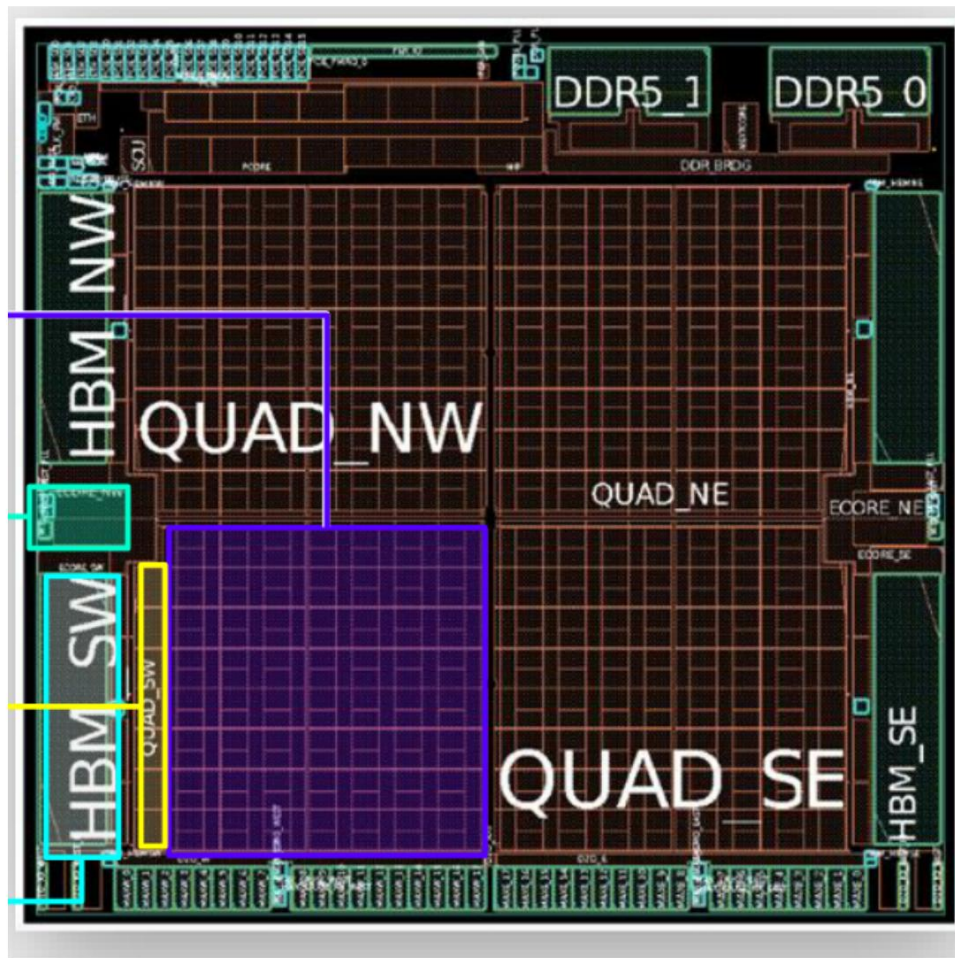


Figure 2 Maverick2 1/4 die shot with various components

Each element of this computation grid is composed of:

- A grid of 56 computation units.
- A RISC-V efficiency core block.
- 7 Tensor core units.
- A high bandwidth memory.

The Maverick2 platform tries to detect hotspots in the code execution flow so they are targeted to specific computation units of the device, giving a better performance over time.

Actually, since it operates in dataflow manner (constructing data dependencies among computations), it really fits well in the taskgraph approach as explained in [1].

On the other hand, the Maverick accelerator also offers a high throughput low latency communication system among memories: both scratch memory on the computing grid, HBM on the die and memory on the host . In particular, HBM offers a single/dual presentation coming with the following specs:

- 96/192 GB memory capacity.
- 3.2/6.4 TB/s bandwidth.
- Multilevel cache hierarchy.

Both for the LHCb and SKAO use cases, we can benefit from the inter device memory data movement. As we saw in Section 2.1, we are really interested in avoiding host to device memory movement, so in case we want to deploy these platforms in a grid of

Maverick accelerators, we will be able to work as much as possible in device memory ensuring low latency.

Finally, regarding interoperability, the Maverick accelerator offers support for the main de facto standards in the HPC ecosystem (to be updated during the lifetime of the project):

- OS: Currently support Rocky 9.X with specific kernel version. NextSilicon can support RHEL 9.X with a specific kernel version and work needed for other OS like Debian.
- Programming support: Currently C/C++, Fortran coming soon, other languages like Julia, Rust, Python needs more work.
- Programming models: Currently support OpenMP, future plan for HIP / CUDA, OpenACC needs more work.
- Distributed memory: Currently OpenMPI, MPICH needs more work

In the rest of the ODISSEE project, if the currently supported features will not suffice, we will prioritize the needed work to add support to the rest and/or work around it and use the supported features.

As we saw in Section 3, OpenMP may offer a very good approach to the resource utilization problem in the context of this project.

## 5. References

---

- [1] C. Yu, S. Royuela and E. Quiñones. Taskgraph: A Low Contention OpenMP Tasking Framework. Technical report, BSC, 2024.
- [2] Rosa Maria Badia Sala, Javier Conejero, Carlos Diaz, et al. Comp superscalar, an interoperable programming framework. SoftwareX, 2015.
- [3] C. Broekema, E. Rennault, V. Casillas, O. Margalit, A. Margoullis. WP3 - D3.1 Hardware platform demonstrators . Technical report, 2025.
- [4] C. Broekema, M. Girone, V. Gligorov, D. Gratadour, N. Neufeld, E. Tolley. WP1 - D1.1 Converged science program specification capture document. Technical report, 2025.
- [5] Enric Tejedor, Yolanda Becerra, Guillem Alomar, et al. Pycompss: Parallel computational workflows in python. Int. J. High Perform. Comput. Appl., 2017.
- [6] Xiran Zhang, Javier Conejero, Sameh Abdulah, Jorge Ejarque, Ying Sun, Rosa M. Badia, David E. Keyes, and Marc G. Genton. Rcompss: A scalable runtime system for r code execution on manycore systems, 2025.