State of the Art of edge computing orchestration and Unikernels technologies

D2.2

ORCHIDE

	Name & Role
Written by	Sergiu Weisz (UNSTPB), Virgile Robles (Tarides)
Verified by	Dragoș Petre (TAR), Adele Hankache (TAS), Mihai Carabaș (UNSTPB), Elena Mihăilescu (UNSTPB), Aurelien Castanie(TAS)
Advisory Board	Guillaume Pierre(IRISA), Nicolas Longepe(ESA), Fabien Vigeant(CNES), Daniel Smith(DSI)
Consortium	TAS, TAR, KPL, THR, UNSTPB
Approved by	Adèle KARAM HANKACHE, Coordinator (TAS-F)

Approval evidence is kept within the document management system

Co- Funded by the European Union Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

Change Records

ISSUE	DATE	§ CHANGE RECORDS	AUTHOR
1	29/05/2024	First issue	Sergiu Weisz
2	30/05/2024	Integrate first round of feedback	Sergiu Weisz
3	30/05/2024	First issue of Section 3	Virgile Robles

Table of contents

1 Introduction

1.1 Scope and purpose

The State of the Art document aims to create a summary of the latest developments in the area of edge computing orchestration and unikernels with the objective of gaining knowledge on what the issues and potential solutions might be implemented to attain the project objectives.

A comparison between currently available orchestration and unikernel solutions will help in designing the orchestration software for the ORCHIDE solution by answering the questions regarding current issues in the field. By creating a thorough State of the Art, the design phase will benefit from a broader range of options either to integrate or to measure against.

1.2 Applicable documents

The documents listed below are applicable to this document.

Internal code / DRL	Reference	Issue	Title	Location of record
AD01	HORIZON-CL4-2022-SPACE-01-11	01	ORCHIDE Proposal	

1.3 Reference documents

The reference documents are given below.

Internal code / DRL	Reference	Issue	Title	Location of record
RD1				

1.4 Definitions and Acronyms

Below, the acronyms used in this document:

Acronym	Definition
DPU	Data Processing Unit
FPGA	Field-programmable gate array
CI	Continuous Integration
CD	Continuous Delivery

K8s	Kubernetes
API	Application Programming Interface
HA	High Availability
CIS	Center for Internet Security
OIDC	OpenID Connect
OCI	Open Container Initiative
ASLR	Address Space Layout Randomization
KVM	Kernel-based Virtual Machine
MCU	MicroController Unit
RTOS	Real Time OS
ABI	Application Binary Interface

1.5 Document outline

The State of the Art document has a split focus between the two areas of work for the orchestration solution for ORCHIDE.

Section 2 presents the user requirements for the ORCHIDE project as established in the D2.3 User Requirements Document and Quality Performance Metrics deliverable in the context of orchestration tools. Based on the latest literature, events and industry trends five solutions have been chosen for comparison. Each orchestrator has been described in the light of the user requirements set.

Section 3 details the advantages of using unikernels for applications and what differentiates the unikernel implementations. Four unikernel solutions are analysed in terms of binary size, boot time, memory blueprint and performance in benchmarks. The issue of passing through hardware to unikernels, is analysed, as it is a necessary for running ORCHIDE applications in unikernels.

2 State of the Art on Orchestration solutions

Advancements have been made in the current technology environment which have allowed for the miniaturization of hardware and an increase in performance per watt, moving computing from the datacenter to new places closer to the user or data. By decreasing the distance between computing and data processing, the response time will be smaller thanks to the smaller latency and higher bandwidth between data and compute resources.

The move outside of datacenters, to the edge of computing environments, has been met with new issues and opportunities. The task of federating workloads has moved from single datacenter to multi-area deployment, requiring a new overlay layer to federate the deployments. It ensures cohesive configuration and manage access and application lifetime.

The popularization of low-power ARM-based processors, Data Processing Units (DPUs) based on FPGAs and AI-accelerating hardware have enabled novel applications of computing and an increase in flexibility for application deployments. The ORCHIDE project aims to take advantage of these innovations to make processing workloads run on satellites It is considered as an edge computing use case, that is flexible by giving users the ability to deploy different applications to satellites and enable multi-workflow processing.

By moving data processing on satellite, the ORCHIDE project decreases the response time for activities. Instead of raw data being moved from a satellite to a ground station, it can be processed on satellite, and only the result can be sent to the ground. Data transfer bandwidth is still the limiting factor in space telecommunications, which causes the delay in processing when transferring data to ground stations.

2.1 Solution requirements

As part of deliverable D2.3, requirements and expectations have been set for the ORCHIDE solutions. In looking for a possible orchestration solution, these requirements have been taken into account. All the solutions have been chosen based on these requirements and compared through them.

Applications will be developed by users who will deploy them to a digital twin working environment or to a launched payload in space. The deployment and testing framework has to offer as little friction as possible for users, to decrease the overhead of programming an application specific to the space edge environment. An orchestration platform needs to adhere to industry standards and provide a well-documented, user-friendly interface through which developers can interact with their applications.

The ORCHIDE consortium aims to build a hardware-agnostic platform which can be deployed on multiple target payloads. Thanks to the open source nature of the project, any manufacturer would be able to pick up the solution and modify it to suit their own needs and run it on their own hardware, thus increasing the potential market for the solution and the services offered through it. If an existing solution is to be chosen, this should be modular enough to allow for deployment on multiple types of hardware architectures such as single or multi-node clusters, computing using CPUs, GPUs or DPUs, and managing on-node or shared storage solutions. As part of the modular design for the solution, both single and multi-node deployments must be considered, requiring the ability for the solution components to be collocated.

Orchestrating unikernels is a requirement for an existing middleware platform, thus, it should be picked up for the ORCHIDE project. Actions such as starting, stopping, restarting, reconfiguring, managing logs and managing input files and the lifetime manager should be compatible with multiple unikernel platforms, to give developers the flexibility of using the platform of their choice, without requiring them to port their software to another unikernel framework.

Space edge computing environments through their remote, isolated and low-power nature require efficient software solutions to be run at maximum capacity. The cost of deploying such an environment cumulated with the requirements to maintain them leads to the need for efficient use of the available resources. Any overhead added for managing the software should be kept to a minimum to allow for more applications to run in parallel, or for more resources to be dedicated to a single application.

In future iterations, the ORCHIDE project aims for multi-satellite networks which could balance the workloads between them. This requirement means that the orchestration solution must offer federation support, or an interface must exist to integrate it with other multi-cluster management solutions.

A multi-tenant approach must be used to grant a cluster the ability to deploy applications for multiple users while isolating them from each other and from the underlying hardware. Many cluster management tools offer support for this, but security controls such as role-based access control, authentication, storage and communication encryption must be set in place to guarantee system security.

Monitoring is crucial for performance tuning and ensuring availability. It can be used for preventative actions, such as noticing that the storage system capacity is becoming close to critical in which actions can be taken to prevent issues. Reactive actions can be taken based on monitoring in the case of a system failure, to trace back the issues and help prevent future similar faults. An orchestration solution should be able to manage and send monitoring data from applications to a central storage and further to the users.

The community aspect of a project is important, especially in the open source world, because the community can inspire new features, contribute code, report issues and provide solutions, even if the maintainers don't. A project's community can build on top of its available features to enhance it while providing the modifications to other members of the community, which can increase the project's lifetime. If the ORCHIDE project opts to use an external project, a differentiating criterion will be the community behind it.

Dataset management can be handled by the orchestrator to free the user from the overhead of data handling. An orchestration solution focused on usability and flexibility must be able to create, copy, move and enforce permissions for users' data files. If an orchestration solution includes data management, it will reduce the cost of implementing the ORCHIDE project and allow more focus on the aspects which require more customization.

The ORCHIDE solution design requires the pipelining of action, which helps in automating processing workloads. Instead of sequential actions being done by hand by developers, they could be accomplished automatically through triggers and pipelines, such as in a Continuous Integration/Continuous Deployment solution. Such actions can mean that an image is taken, which triggers a processing action, which then launches an identification process, which then starts a counting process and so on. Pipelines must be managed through priority queues to allow for tasks to be ordered not only by their insertion time in the queue but also by their mission importance.

2.2 Kubernetes based solutions

Kubernetes, abbreviated as K8s, is a cloud-native orchestration solution used primarily in datacenters focused on running services built according to the micro-service paradigm. It has gained a large adoption rate because of the automatic scalability and flexibility it brings to datacenter. In Kubernetes, applications can be automatically scaled based on metrics such as the number of connections to them as opposed to virtual machines or whole-node servers, where scaling requires manual intervention and a larger resource investment.

Kubernetes has enabled a new way of thinking about applications by using containers and pods. An environment can be split up into multiple services which it provides that can be scaled individually. Each service is built as an application, giving it the advantage of modularity, as a container can run on any system which can start containers, no matter the underlying libraries and operating system, giving way to more flexible deployment strategies.

Container-based development and the CI/CD paradigm has enabled users to test, deploy and ensure that their code runs on remote servers as on their local or testing machine. This approach to development is enhanced by Kubernetes and is a factor which has led to its widespread acceptance as the de facto standard for cloud native computing. Pipelines can be integrated and triggered in a cluster by applications such as Argo Workflows, Apache Airflow or Kubeflow for Machine Learning workflows.

The community has developed the *urunc* and *runu* projects which both approach the subject of seamlessly running unikernels inside of Kubernetes architectures by plugging into the container runtime component and implementing its API for managing containers. Unikernel image management is done through the internal Kubernetes API, which helps offload the deployment application workflows to it.

Monitoring in Kubernetes must use specialized containers which can handle the logs. Logs come in three varieties:

- Audit logs storing every action taken inside of the cluster.
- System logs, logs from middleware services.
- Application logs which are reported by specific workloads.

Both edge and regular deployments user services such as Greylog, Nagios or Prometheus gather cluster metrics and use a visualizer to inspect them, while a log forwarder sends logs to a database such as Loki to index and tag them. Monitoring and logging applications present the issue of increased complexity and performance requirements from a cluster, adding more software on a node just to monitor the system.

Kubernetes has integrated workflows which need access to accelerator-type hardware. It uses operators which abstract the complex driver setup away from the users and share the hardware resource with the pod. Such operators need to be imported into the Kubernetes cluster software.

Thanks to its large adoption and the open source code, Kubernetes has enjoyed many forks, offshoots and side-projects adapting its workflows to their use cases. There exist many Kubernetes projects which aim to port it to the edge computing use case. The distributions fix the issue of resource efficiency for Kubernetes by decreasing the footprint of the binary and decreasing the service footprint on the CPU.

Four Kubernetes solutions have been chosen for comparison based on the current market trends in the Kubernetes world. The solutions come from different vendors and have implementations which cater to certain use cases. KubeEdge has been chosen for its ability to run in resource constrained environments while connecting to lightweight edge hardware, K3s and K0s have been chosen because they are de facto distributions in the Kubernetes world, having large community support, and MicroK8s has been chosen for its ease of use and install, integrating with pre-built modules to create an easier to setup environment for resource-constrained hosts.

2.2.1 KubeEdge

KubeEdge[1] extends Kubernetes K8s to edge devices, enabling communication and management. Installation involves setting up a cloud side (Kubernetes cluster) and an edge side (edge nodes), using a tool like *keadm*. KubeEdge is a cloud-native platform that integrates centralized data centers with distributed edge computing, offering management of containerized applications across both environments using Kubernetes tools, and includes functionality for offline operation.

In Figure 1: KubeEdge architecture we can see the architecture that this tool proposes consists in two main components CloudCore and EdgeCore. CloudCore is responsible for handling tasks like device management, synchronization, and resource allocation. CloudCore serves as the central hub that ensures the efficient operation of the entire KubeEdge cluster. The second component of KubeEdge, EdgeCore, functions on edge nodes for workload and data handling. It communicates with CloudCore for instruction reception and data synchronization and is responsible for device connectivity, computing at the edge, and managing local resources.

[1] KubeEdge - https://kubeedge.io/docs/

KubeEdge offers the advantages of running a centralized Kubernetes instance which connects to the edge nodes such as ample resources for running API services, which regularly use more resources but can integrate with other business workflows. By introducing the connection to the CloudCore a dependency has been added which cannot be easily satisfied in the case of edge computing on satellites where downtimes can be long.

KubeEdge requires the smallest RAM allocation of the edge-deployed Kubernetes offshoots, needing 70MBs. The lack of Kubernetes components on the EdgeCore device has allowed the developers to shrink the memory requirement.

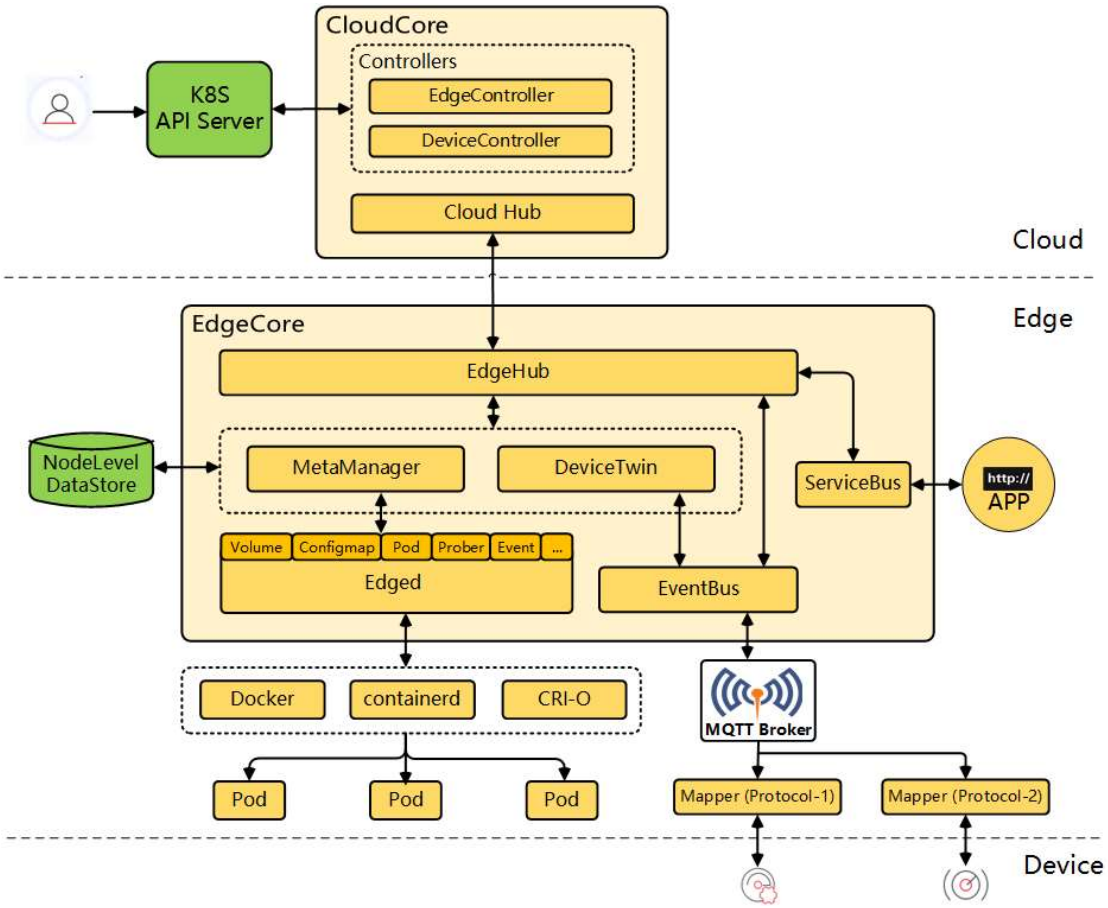

Figure 1: KubeEdge architecture

2.2.2 K3s

K3s[2] is a lightweight Kubernetes (K8s) distribution designed for environments with limited resources, such as Internet of Things and Edge devices. It is used as the benchmark for lightweight Kubernetes deployments because it was one of the first such lightweight deployments. K3s is shipped as a single binary using less than 100 MB disk space and requiring 512 MB of system memory and a single core processor, while the recommended available memory per node must be 1 GB. The solution is compatible with Kubernetes operators, but it ships with a limited container network interface which restricts the networking customizability. The internal database configured for K3s has been changed from etcd[3] to SQLite[4] by default for a lighter environment.

The K3s architecture, seen in Figure 2: K3s architecture, consists of two main components: servers and agents. This structure facilitates the deployment of K3s in various environments, including those with limited resources, without the need for a dedicated cloud component, unlike KubeEdge. K3s servers host the Kubernetes control plane, including the API server, scheduler and can use an internal database or connect to an external database for cluster state storage. They can be set up in a High Availability (HA) mode for increased redundancy and stability. K3s agents, running on worker nodes, are responsible for running containers, they manage local resources like storage and networking.

A compromise made by the K3s system is the lack of advanced security features. While it offers CIS controls, NetworkPolicies for connection management, it does not offer modern authentication protocols like OIDC and NetworkPolicies and Pod Security Policies are disabled by default, offering a less secure experience out of the box which will require further setup steps at deployment time.

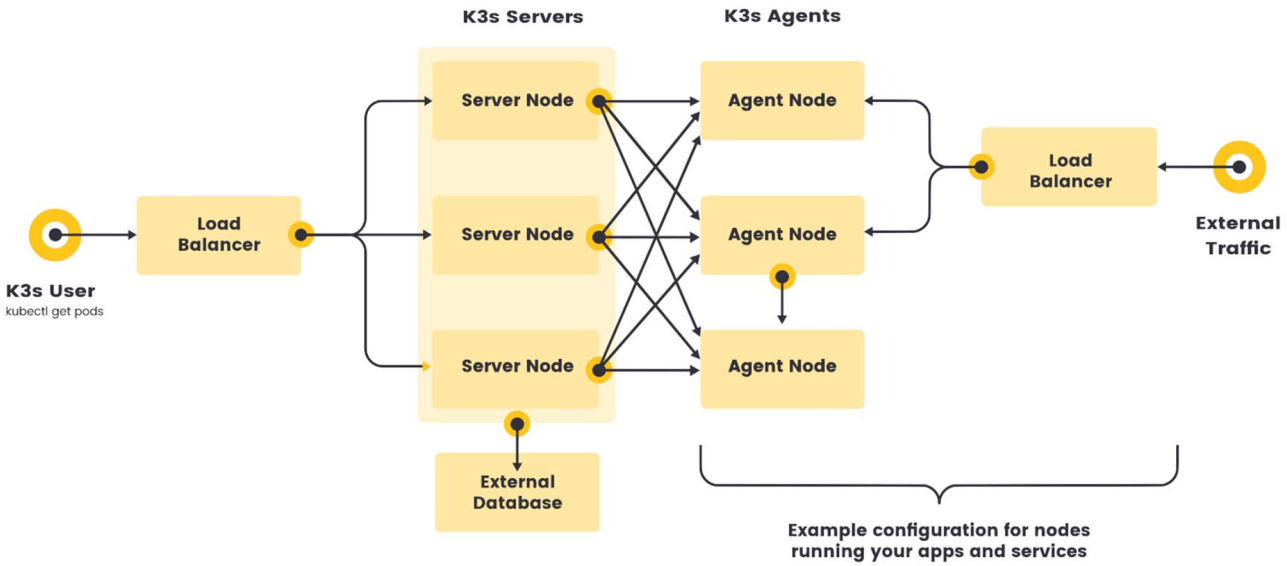

Figure 2: K3s architecture

[2] K3s - https://k3s.io/

[3] Etcd - https://etcd.io/

[4] SQLite - https://www.sqlite.org/

2.2.3 K0s

K0s [5] is another lightweight distribution of Kubernetes. It is deployed as a single binary, making it easy to install in various environments. As opposed to K3s and KubeEdge, it is deployed as a full Kubernetes instance, containing services such as *etcd*. K0s requires minimal dependencies on the host operating system and operates with low system requirements (1 vCPU, 1 GB RAM), while using 512 MB of memory and a binary file smaller than 250 MB.

Out of the box K0s is delivered with the most ready-to-use configuration, having been configured with default NetworkPolicies, Pod Security Politices, GPU operators and other advanced features. By decreasing the number of configurations changes a deployer must make, there is a smaller chance of introducing misconfigurations while also moving the development time from configuring K0s to integrating the workloads. Compared to K3s, K0s offers a more well-rounded and configurable solution, compatible with K8s modules, while losing in resource usage and performance.

2.2.4 MicroK8s

MicroK8s [6] is a Kubernetes distribution developed by Canonical with the intent of deployment in DevOps or CI/CD environments where the goal is to run an easy to deploy test environment. To this extent Canonical has created a snap-based installer which automates the cluster deployment steps. The recommended system requirements for MicroK8s are a 2-core CPU and 4 GB of RAM.

MicroK8s benefits from the support of Canonical, one of the largest open source oriented tooling companies, which affords them a clearer use case, branding, and development path, not relying on community contributions. Within the MicroK8s environment, various addons can be used. These extensions enhance its capabilities by providing extra functionality and features such as load balancing, GPU acceleration, observability, and metric collection tools.

Of the Kubernetes based solutions analyzed, MicroK8s has proven the least mature, relying on the snap package management for setup, requiring more resources than the other distributions and providing limited scalability options. Unikernel support is provided natively by MicroK8s through Kata Containers, a unikernel runtime which plugs into the Kubernetes API. MicroK8s is the only edge computing Kubernetes distribution which supports unikernel lifetime management.

As this paper[7] demonstrates that K3s and MicroK8s, offer better performance in most of the tests compared to the full-fledged Kubernetes deployment using kubespray[8]. Specifically, they demonstrate better cold start delays and serial execution performance. However, the differences in performance between K3s and MicroK8s are not statistically significant in many cases, suggesting that both are comparable in their efficiency and effectiveness for serverless computing at the edge.'

[5] K0s - https://k0sproject.io/

[6] MicroK8s - https://microk8s.io/

[7] Kjorveziroski, Vojdan & Canto, Cristina & Gilly, Katja & Filiposka, Sonja. (2022). Implementing Multi-Access Edge Computing with Kubernetes.

[8] Kubespray - https://github.com/kubespray/kubespray

2.3 Other solutions

Other orchestration solutions have been developed for distributed computing, with examples such as Apache Mesos [9] or HashiCorp Nomad[10]. These solutions accomplish the same goal, but are not built for a low-power, high efficiency environment and have high resource requirements. The field of orchestration is still a growing one and it requires developer attention. The field of edge orchestration is poised to grow as demands for offloading work to areas nearer to the users become a reality with the wide scale adoption of 5G and 6G technologies.

2.3.1 Oakestra

Oakestra[11] is a lightweight and scalable orchestration framework for edge computing. It overcomes the limitations of traditional frameworks like Kubernetes, which do not always fit for dynamic and constrained edge environments. Oakestra features a federated three-tier resource management, delegated task scheduling, and semantic overlay networking. This design allows consolidation of multiple infrastructure providers and supports applications in varied edge conditions. The use case for Oakestra has been made in its white paper to be pipelining image processing tasks, a task which would be compatible with the ORCHIDE use case for pipelining data processing.

Unlike traditional Kubernetes distributions, Oakestra introduces the federated three resource management system as seen in the figure below. This hierarchical approach includes a root orchestrator and multiple cluster orchestrators, enabling efficient resource management across diverse and dynamic edge environments. This contrasts with the more straightforward, singular architecture of K3s, MicroK8s, and K0s.

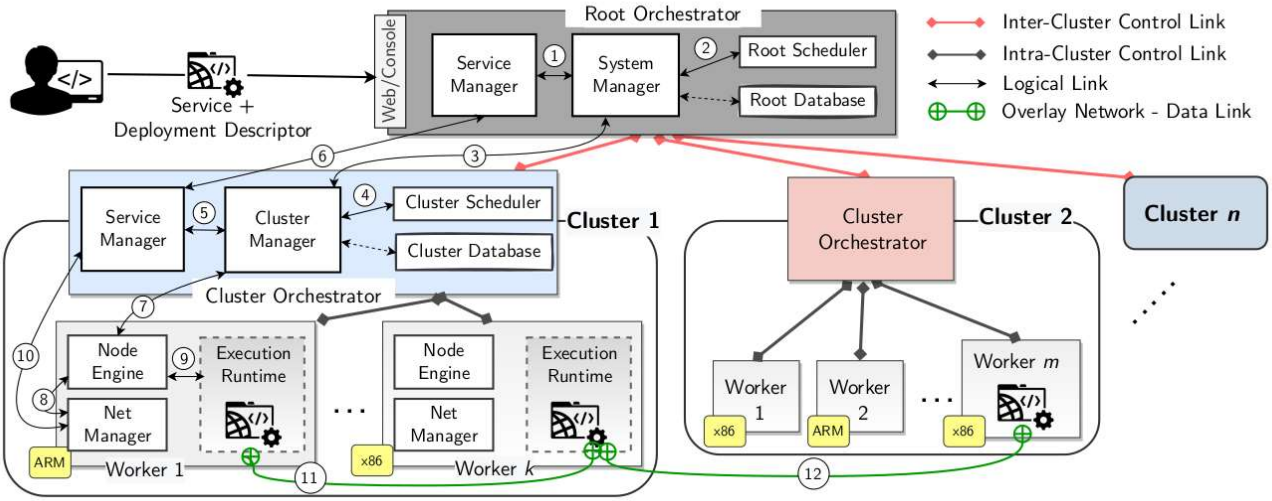

Figure 3: Oakestra architecture

[9] Apache Mesos - https://mesos.apache.org/

[10] HashCorp Nomad - https://www.nomadproject.io/

[11] G. Bartolomeo et al. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing, USENIX ATC '23

Oakestra implements a delegated task scheduling mechanism that separates coarse-grained cluster choices made by the root orchestrator from fine-grained resource placement decisions within clusters. This feature allows Oakestra to effectively handle the variability and dynamism of edge conditions, which is not a primary focus of K3s, MicroK8s, or K0s.

Oakestra's semantic overlay networking is designed to support edge-oriented load balancing policies and ensure portability of cloud-native applications in edge environments. GPU support is limited in Oakestra, support has been implemented only for Nvidia drivers. With the lack of DPU support, ORCHIDE would need to implement its own resource management layer to handle such cases.

QEMU-based Unikernels are supported by Oakestra. Their lifetime can be managed the same way as containers, including running applications from Open Container Initiative (OCI) images downloaded from repositories. This has the advantage of being able to seamlessly run containers or unikernels depending on the workload and environment. Only including QEMU-based unikernels is a downside, as it excludes unikernels based on other environments such as Solo5 which runs MirageOS.

A monitoring dashboard is deployed as part of the Oakestra package, It offers a view of the running workloads together with information such as its location, CPU usage and memory usage. This is still a rudimentary interface which does not expose workflow or system-specific metrics which might be use for prevention or debugging purposes.

The Oakestra project's development can be traced back to 2021 through its Git repository history. Because of its young age, it lacks support for widely used security or usability features such as role based access control or storage management. A community has not been formed around the project, which will affect its growth potential, and the ORCHIDE project cannot take advantage of prior experienced users to have gone through and documented issues and fixes, and community modules and enhancements cannot be found as easily as for Kubernetes based solutions.

2.4 Solution comparison

A comparison matrix has been created to highlight the differences between the solutions described in the orchestration state of the art subsection. The metrics chosen for comparison have been deemed as important for the project success according to the D2.3 User Requirements deliverable.

	KubeEdge	K3s	K0s	MicroK8s	Oakestra
Familiarity to developers	De facto micro-service hosting and deployment framework	De facto micro-service hosting and deployment framework	De facto micro-service hosting and deployment framework	De facto micro-service hosting and deployment framework	New development framework based on
Community support	1.7k forks 6.5k start	2.2k forks 26.7 stars	344 forks 2.9k stars	751 forks 8.2k stars	18 forks 50 stars
Unikernel support	Using third party tools such as runc	Using third party tools such as runc	Using third party tools such as runc	Using third party tools such as runc	QEMU/KVM based unikernel support
Pipeline support	Using third party tools for workflow management	Using third party tools for workflow management	Using third party tools for workflow management	Using third party tools for workflow management	Supports internal pipelining mechanism

Accelerator support	No support for GPUs	Operator can be integrated to manage DPUs and GPUs of various models	Operator driver pre-packaged on deployment	Operator driver pre-packaged on deployment	Only CUDA GPUs supported
Monitoring features	Requires third party tools to be deployed	Requires third party tools to be deployed	Requires third party tools to be deployed	Easy to install addons provided to enable monitoring	Barebones dashboard running on central node
Scalability	Highly scalable using lightweight edge node definition	Scalable vertically, but requires specialized multi-cluster software	Scalable vertically, but requires specialized multi-cluster software	Does not scale	Highly scalable using federated multi-layer
Security features	Detailed threat model and configurations considered, minimizing attack surface by minimally exposing critical components.	Barebones security configurations present at deploy time which can be hardened	Pre-packaged with basic security features enabled	Barebones security configurations present at deploy time which can be hardened	Limited security controls

Table 1: Comparison of edge computing orchestrators

Performance metrics have been considered in publications and talks comparing the above solutions. In their KubeCon 2024 talk, Shivay Lamba and Saiyam Pathak[12] have presented their benchmarking results obtained by running IO-bound and CPU-bound on K3s, K0s and MicroK8s:

	K3s	K8s	MicroK8s
Iperf network test (larger is better)	1 Gb/s	559 Mb/s	542 Mb/s
sysbench CPU test (larger is better)	553.55 events/s	384.61 events/s	335.07 events/s
CPU and memory test (larger is better)	528 BogoMIPS	363 BogoMIPS	231 BogoMIPS

1 [12] G. Bartolomeo et al. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing, USENIX ATC '23

Table 2: Performance benchmarks for Kubernetes distributions

They have noticed in terms of raw CPU power, K3s is the better choice, as it has a smaller CPU overhead compared to the other devices in a resource container environment running 1 core, 2 GB memory.

Oakestra has been compared by its authors in the 2023 USENIX Annual technical Conference [13] to K3s as the closest in performance of it. For an object tracking processing workload, K3s and Oakestra have obtained comparable results of ~300-400ms, while for more resource intensive work Oakestra has outperformed K3s by 10%.

Deployment speed has not been chosen as a benchmarking metric between the tools. The time from the workflow trigger to the image deployment is negligible compared to the latency between the ground station and the satellite.

3 State of the Art on Unikernel Solutions

A major aim of the ORCHIDE project is to provide a framework within which not only applications can be efficiently orchestrated, but also efficiently executed: that is, the deployment of an application and its dependencies (upload, host communication, monitoring, etc.) should incur the lowest overhead possible on the constrained on-board resources.

It has been decided early on that within ORCHIDE, application would be packaged as unikernels as much as possible, as this technology generally allows to significantly reduce the size of the application binary image, as well as other runtime resources (time, memory, etc.), while maintaining a high standard of security by leveraging any on-board hypervisor to isolate applications between each other and the host (as applications are generally packaged into full-fledged virtual machines (VM)). The unikernel approach generally complies with the requirement of flexibility imposed by ORCHIDE, whereas generally applications should be self-contained (with no dependency on the host other than potentially hardware) and hardware-agnostic: an application should be able to be re-targeted, and executed, for different hardware architectures (in particular, CPU architectures), as long as the interfaces it needs to the outside world are present.

This section of the state of the art gives a summary of the technology, explores the available implementations of the unikernel concept with respect to several criteria for the ORCHIDE project, provides performance metrics for the Unikraft and MirageOS implementations, and gives a few leads towards interaction between unikernels and hardware than can be found in satellites that would be “unusual” for the usual cloud environments that unikernels are generally targeted for.

3.1 Unikernels, a summary

Unikernels are specialized, single-address-space machine images constructed by combining application code with only the minimal set of operating system (OS) functionalities required to run that specific application. Unlike traditional operating systems that support a wide variety of applications and users, unikernels are tailored for specific tasks, leading to several notable advantages:

- By including only the necessary components, unikernels have smaller footprints and consume fewer resources. This results in faster boot times and lower memory usage.
- With less overhead and fewer context switches compared to general-purpose OSes, unikernels can offer improved performance for certain workloads.

[13] G. Bartolomeo et al. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing, USENIX ATC '23

- The reduced attack surface, due to fewer included components and services enhances security. Each unikernel can be highly specialized and hardened, minimizing vulnerabilities.
- Unikernels run in isolated environments, typically as virtual machines (VMs) or on cloud platforms, providing strong isolation between instances. This isolation is beneficial for multi-tenant environments.
- They can be deployed across various environments, including on-premises data centers, public clouds, and edge devices, due to their lightweight nature and minimal dependencies.

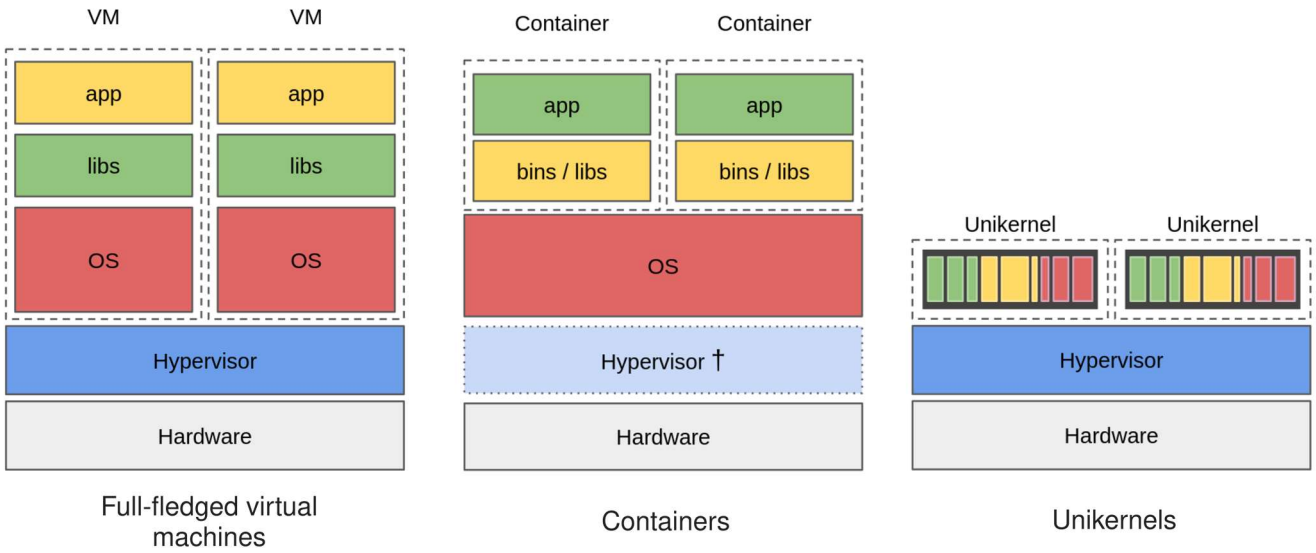

Figure 4: comparison of sandboxing models

Cloud infrastructure has historically been a natural use case for unikernels, as a sandboxing model both more lightweight than full-fledged virtual machines, and more secure than containers (architectural differences as illustrated in Figure 4).

The unikernel approach often involves building the application together with its necessary OS components into a single binary image. This process is supported by unikernel libraries and toolchains, and here lies the difference between most implementations. These toolchains help developers specify and compile the required functionalities directly into the unikernel, ensuring the resulting image is optimized for its intended purpose.

An implementation of the unikernel concept is guided by several fundamental decisions concerning is design:

Application domain. The implementation can be general-purpose, or targeted towards specific domains or use cases, hence providing mostly components that are useful in that prospect. For example, implementations exist that target only operating network routers.
Within ORDHIDE, only general-purpose solutions are considered.

Architecture design. Implementations can choose to be either specialized monoliths, on which applications can build upon (and in this case, such implementations may more accurately be described by the term “microkernel”), or a set of modular components than can be assembled (or dropped) depending on the specific needs of an application.
Within ORCHIDE, only modular solutions are considered.

Application compatibility. One of the most defining points is compatibility with existing applications, and constraints on newly written applications. This is guided by the interface model chosen between running applications on the unikernel runtime. In particular, some implementations choose to expose the POSIX interface (or part of it), ensuring POSIX applications (built e.g. for Linux) can be compiled against unikernel libraries, or even run as unmodified binaries. On the other hand, some implementations choose to drop completely the POSIX model to explore the interface space, and provide sometimes simpler, more efficient, more explicit or more abstract interfaces to the applications. This can hinder compatibility with existing applications which have to use these interfaces to communicate with the external world, and more generally hinder support for certain programming languages, of which the runtime relies on POSIX.
Within ORCHIDE, both approaches are considered.

Execution model. Several factors affect where and how unikernels can be executed. In particular, the sandboxing model used to isolate applications: which hypervisors are supported, with which monitor? Are there options if hardware virtualization is unavailable on the target? More generally hardware compatibility is a key factor: CPU architectures compatible with the implementation runtime, available drivers for external devices, and exposed interfaces for communicating with such devices.
Within ORCHIDE, solutions supporting the broadest range of hardware targets and sandboxing models are considered.

Security. The design of the unikernel implementation guides the security and safety of its own components and the running applications. Factors include the programming language used for the kernel components (with respect to memory safety, data race detection, maturity), the maturity and trust in specific kernel libraries, in particular the ones exposed to the outside traffic such as protocol implementations or cryptography libraries (algorithms available, test suites, formal methods applied, etc.). Furthermore, as the kernel components are often alongside the application in the same address space in the final binary, a range of risk mitigation can be applied by the implementation to limit manipulation of that address space by untrusted applications (such as address randomization (ASLR), mutually exclusive execution or modification of data (W^X), etc.).
Within ORCHIDE, solutions are compared on their security maturity and efforts.

3.2 Overview of existing implementations

Considering only open-source implementations of unikernel, we could list the following projects: ClickOS (https://github.com/sysml/clickos), Clive (https://lsub.org/clive/), Drawbridge (https://www.microsoft.com/en-us/research/project/drawbridge/), HaLVM (https://github.com/GaloisInc/HaLVM), HermitCore/RustyHermit/Hermitux, IncludeOS (https://www.includeos.org/), LING, Nanos, MirageOS, OSv, Rumprun (https://github.com/rumpkernel/rumprun), runtime.js (http://runtimejs.org/), Toro kernel, Ultibo, UniK (https://github.com/solo-io/unik) and Unikraft.

Excluding all projects not satisfying the criteria of the last section, alongside with the constraint that project should have seen at least one maintenance update in the last four years, yields the following short list, which we discuss afterwards:

- RustyHermit (https://github.com/PJungkamp/rusty-hermit)
- Hermitux (https://ssrg-vt.github.io/hermitux/)
- NanOS
- MirageOS
- OSv
- Toro kernel (https://torokernel.io/)
- Unikraft

Comparison of these implementations is presented in Table 3, according to criteria highlighted in the last section, as well as **maturity** (proof of concept, experimental or used in production), **ability to run on bare metal** and **KVM monitor** (the program used on the host when the hypervisor is KVM, which has an impact on both performance and security).

Given the ORCHIDE need to run on diverse CPU architectures, and in particular ARM-based cores which are omnipresent in embedded contexts and space in particular, we further reduce that list to consider only unikernel implementations that are compatible with at least x86_64 and ARM-based processors, yielding the final list: **NanOS**, **MirageOS**, **OSv** and **Unikraft**.

As the consortium of the ORCHIDE project comprises experts on MirageOS and Unikraft (resp. Tarides and UNSTPB), these implementations will be first-class citizens within the ORCHIDE solution and explicit support is planned. This does not preclude compatibility with other implementations, which is why we discuss all four solutions below.

	Maturity	Bare metal	Hypervisors	Non-amd64 CPU archs	KVM monitor	POSIX compat	Core lang.	App language
Rusty Hermit	Exp.	NO	KVM	None	qemu, uhyve	NO	Rust	Rust, C, C++, Go, Fortran
Hermitux	PoC.	NO	KVM	aarch64	proxy	YES + ABI	C	C, C++, Fortran, Python, Lua
NanOS	Prod.	NO	KVM, HVF, ...	aarch64	ops	YES	C	C, Go, PHP, JS, Rust, ...
MirageOS	Prod	PARTIAL	KVM, Xen, bhyve, vmm, muen, seL4, ...	aarch64, RISC-V, PPC, IBM-Z	solo5	NO	OCaml	OCaml, C, C++, Rust
OSv	Prod	NO	KVM, Xen, VMWare	aarch64	qemu, firecracker	YES + ABI	C/C++	C, C++, Go, Rust, JVM, Python, JS
Toro Kernel	Exp.	NO	KVM, Xen, ...	None	qemu, firecracker	Partial	Pascal	Pascal
Unikraft	Exp.	NO	KVM, Xen, ...	aarch64	qemu, firecracker	YES + ABI	C	C, C++, Rust, Go, Python, Flask, Node, ...

Table 3 comparison of considered unikernels

3.2.1 NanOS

NanOS (https://nanos.org) is written in C and targeted towards cloud environments. It aims for broad software compatibility by reimplementing part of the Linux system call interface (thus supporting a diverse family of programming languages) and platform support, supporting hypervisors KVM, Xen, Bhyve, ESX, Firecracker, and Hyper-V. It has limited support for RISC-V alongside x86_64 and arm64.

While being written in C, NanOS has several runtime security characteristics: several risk mitigation features (ASLR, page protections, etc.). Additionally, one defining feature compared to most other unikernels is that the kernel/user mode switch is kept as a security measure, which impacts performance for increased isolation.

NanOS in general provides several opinionated models for developing applications, with a preferred file system (TFS), data structure implementations, etc.

Regarding orchestration and lifecycle management, it integrates with several known cloud providers, and has partial compatibility with Kubernetes-based orchestrators.

3.2.2 MirageOS

MirageOS (https://mirage.io) is written in OCaml (a memory-safe programming language) and one of the oldest unikernel implementations, initially targeted for cloud environments. By design, it does not implement POSIX support in any way, instead relying on a set of its own Mirage interfaces and abstractions for communication between applications and host. On the other hand, platform and hardware support is extensive: KVM (solo5), Xen, Bhyve, muen, seL4 hypervisors, on various CPU architectures (see Table 3). Furthermore, the bytecode support for the OCaml compiler means an application written for unikernels may be executed on extremely limited targets such as MCUs or RTOS, as long as a C compiler is available.

In our survey, this is the solution that can be deployed on the most diverse combinations of CPU architectures and platforms. In practice, this large support is made possible by the small surface between unikernel runtime and platform MirageOS keeps, which makes it easily portable and more secure (as the attack surface is reduced). However, this means the kernel and application may not have access to all the granularity of the hardware, which can reduce the ability to exploit it and often leads to poorer performance compared to other unikernels.

In that regard, MirageOS is the most opinionated implementation in this survey (both in terms of available programming languages, exposed interfaces and data structures), but also the most security and reliability oriented.

It has limited compatibility with existing orchestrators, and instead has its own tools for local management *albatross* (https://github.com/robur-coop/albatross).

3.2.3 OSv

OSv (https://osv.io/) is written in C/C++ and targeted towards cloud environments. Compatibility with applications, and in particular the Linux API, is a guiding goal for the project, and as such a large number of non-forking applications can be run unmodified on OSv. Java in particular is a first-class application programming language supported by OSv. In turn, this makes porting to other CPU architectures a large effort and OSv currently supports only x86_64 and partially arm64. It is however available on a diverse set of platforms and providers, including KVM (qemu or Firecracker), Xen and VMWare.

OSv provides no particular security features that are not already provided by the underlying hypervisor, and rather focuses on performance and compatibility goals. It provides comprehensive tooling for building and packaging applications through its Capstan tool. Orchestration and lifecycle management is generally delegated to cloud providers the built unikernels run on.

In general, in the survey OSv provides less granular customization compared to others but provides a simplified environment tailored to running applications efficiently, rather than striving for minimalism.

3.2.4 Unikraft

Unikraft (https://unikraft.org/) is written in C and targeted towards cloud environments. It is based around the concept of small, modular libraries, each providing a part of the functionality commonly found in an operating system (e.g., memory allocation, scheduling, filesystem support, network stack, etc.). Like OSv, there is an important focus on POSIX compatibility, whereas a large subset of system calls is implemented, and an experimental ABI compatibility layer is provided.

Unikraft is not easily portable to new platforms and architectures, and as such, it focuses mainly on the KVM hypervisor (through qemu or firecracker), with Xen available as well. Backends are available for the x86_64 and arm64 (experimental) architectures. On the other hand, in the survey, Unikraft was the most performant unikernel for a set of test applications, through its efficient use of the hardware combined with the minimalism of its modular components' implementations.

While being written in C, Unikraft has implemented a set of runtime risk mitigations features (stack protections, ARM-specific security features, etc.), with more planned. Additionally, thanks to its very modular architecture, internal components of Unikraft can be selectively rewritten in other (potentially safer) languages such as Rust (and there have been efforts towards that goal), resulting in a mix and match of languages along its internal interfaces.

Unikraft has comprehensive tooling for building/deploying unikernels (via its *kraft* tool) and good orchestration support. It is one of the most actively maintained projects of the surveyed solutions, being driven by a large community of users, students and a dedicated cloud company.

3.3 Performance

In both our testing and third-party benchmarks, Unikraft has generally proven to be more efficient, both in terms of runtime performance (except for boot time), runtime memory footprint, and binary size.

Table 4 presents benchmarks conducted on the four unikernel implementations presented in the last section, based on a standard implementation of a web server in each case. For NanOS, OSv and Unikraft, the nginx web server has been compiled and packaged as an unikernel (using provided binaries by unikernel authors when available), and for MirageOS the standard web server implementation of the OCaml ecosystem has been used (since nginx was not easily portable to MirageOS). Benchmarks were all done on KVM, with qemu as a monitor except for MirageOS where solo5 was used. We added nginx running natively on Linux as a comparison point. Unikernel binaries were stripped in each case, and all optimizations enabled.

Performance was measured as the number of HTTP GET requests each unikernel could handle per second (where the target of the request was a document of 1 Kb). In the case of nginx, stock configuration was used.

	NanOS	MirageOS	OSv	Unikraft	Native Linux
Binary size (Mb)	3	2.2	5 (native)	0.8	5
Boot time (ms)	~80	~2	~90	~3.3	N/A
Request/s (x1000)	40.5	27.4	234.2	298.4	230.2
Memory footprint (Mb)	28	10	27	5	12

Table 4 Comparison of performance metrics across different unikernels

Our findings were generally consistent with metrics measured by Unikraft authors in "U*nikraft: fast, specialized unikernels the easy way*" by S. Kuenzer et al., which however didn't include a comparison with NanOS.

3.4 Hardware Interfacing

Unikernels implementations are generally designed to run in cloud environments, where hardware interactions are limited to mostly networking (through network interfaces) and storage (through block device), with varying levels of performance depending on the granularity of hardware details exposed to the application through the interface. In the world of unikernels, this interface is either POSIX (through a limited set of supported system calls) and/or a custom abstraction (as is the case with Unikraft and Mirage).

Repurposing unikernels to execute satellite payloads introduces new challenges: on one hand, the satellite infrastructure doesn't always cleanly match the expectation of cloud with readily available IP network links (for internal and external communication), and some effort might be needed to fill the gap between the exposed application interfaces, and communication with real devices. On the other hand, a lot of payloads will want to interact with new kinds of devices that were previously not properly supported or even considered in unikernels.

3.4.1 On-board resource usage

Sensors, cameras, and other important devices that justify the presence of a satellite in the first place, are the main input from which value can be derived in software payloads. A unified way of interacting with such devices will need to be designed within ORCHIDE, to allow orchestrated delivery of input data to payloads (on a "push" model). But some applications require actively querying on-board resources as well ("pull" model), for example to reconfigure a sensor, request a new orientation of a camera, request a snapshot be taken at a normally idle time, etc.

This implies two lines of work: give unikernels the means to interface with these devices (most commonly, with the on-board bus), with proper abstractions; and fair arbitration of resource usage. The former means an extension of the surface between unikernels and hypervisors (which is a task of varying difficulty depending on the underlying used abstractions, and hypervisor monitor), and having device drivers available. For that last part, this can be either through reuse of existing (Linux) drivers for specific devices, or reimplemented as unikernel components on top of low-level interfaces like I2C.

The latter line of work, arbitration of resource usage, is needed to avoid contention and denial of service between payloads on common resources. The classical architecture to achieve that is having a central, privileged software component (unikernel or not) directly with the resource and forcing all payloads to go through that component to communicate with devices. That component is then responsible for rate-limiting, assigning usage slots, etc. This also implies that ORCHIDE should give control to the operator to set device access permissions, time ranges, quotas, etc. for each payload/device combination.

3.4.2 Hardware accelerators

One special case of on-board resources is that of hardware accelerators and in general non-CPU compute units: GPGPUs, FPGAs, TPU/DPUs, etc. This is important, as their use is becoming more and more prevalent in on-board payloads, in particular for AI applications such as image inference (one simple case would be cloud detection, to avoid sending useless images to the ground). Notably, such devices will be part of the ORCHIDE demonstrator, and AI payloads are planned.

As devices, what sets them apart is the sheer complexity of their drivers, vendor libraries and SDKs (and the general fragmentation across very different ecosystems). Additionally, their computational nature brings security concerns that are hard to properly solve. The aforementioned complexity means that: generally, drivers and libraries for accessing these devices cannot be rewritten as unikernel components; and existing AI applications are generally deeply coupled with a specific AI framework and cannot be easily rewritten.

Regarding security, it is generally admitted (for example by Tensorflow authors), that running inference on an arbitrary model is equivalent to running arbitrary code on board. With a lack of hardware virtualization available on accelerators, this implies a breach to the sandboxing model, no matter if the controlling code is executed from a VM or a container. This implies that proper controls should be given to ORCHIDE operators to restrict access to accelerators to only trusted and audited models, if they so wish for.

Once the security problem is acknowledged (and the usage arbitration problem above as well), there is still the matter of interfacing between virtualized applications and these devices. One first requirement is proper support for such devices in the orchestrator itself, so it can present an interface to guests (see the Orchestration section of this document). Then like the section above, unikernel interfaces can be extended to communicate with these devices, with a small custom interface, which can be part of the ORCHIDE SDK. This approach has been experimented with successfully in "*GPU Acceleration in Unikernels Using Cricket GPU Virtualization*" by N. Eiling et al., where CUDA-like capabilities are added to RustyHermit and Unikraft.

While this is potentially acceptable for newly developed application, this is not applicable to existing AI applications, which make use of AI framework APIs such as Xilinx, Tensorflow, Pytorch etc. This is because embedding the whole underlying libraries corresponding to these frameworks (excluding the driver) is often very hard or infeasible, as they expect extensive POSIX compatibility and direct communication with the (Linux) driver. One approach to tackle that problem is the one of vAccel (https://docs.vaccel.org/), illustrated in Figure 5.

Another, simpler approach, is dropping these layers necessary to unikernels, and considering running payloads interfacing with accelerator devices in classical containers. To keep the benefits of unikernels in that case, the application developer could split the application between the core device interactions (which would live in a container), and the companion pre- and post-processing code, which executes on the CPU and can still be packaged inside unikernels.

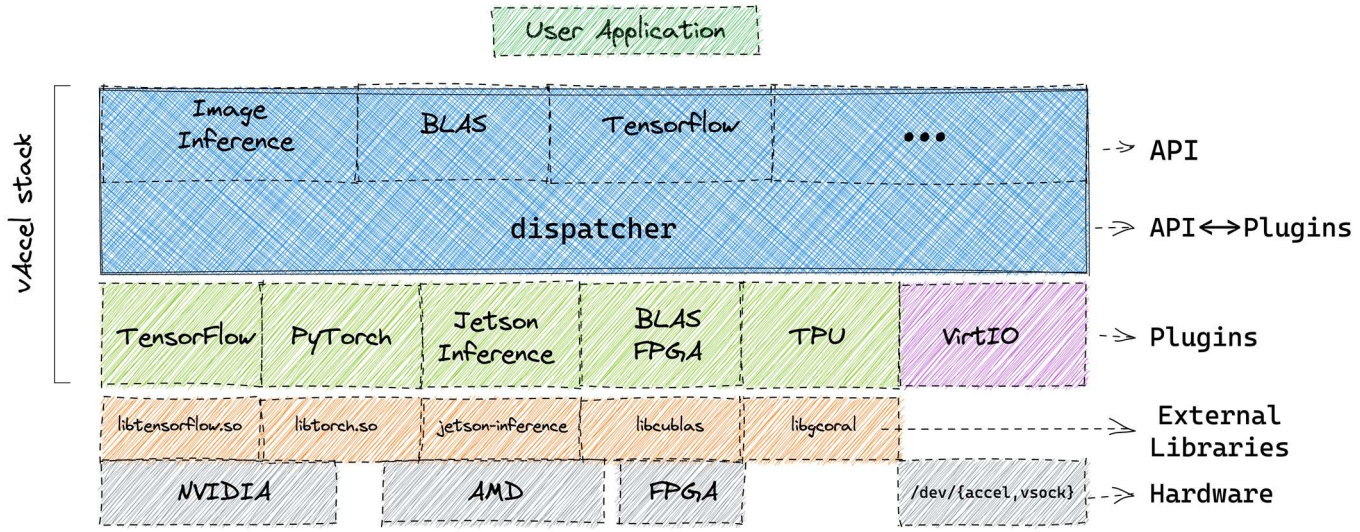

Figure 5: layers of the vAccel approach to unikernel/accelerator interaction

4 Conclusion

Orchestration and unikernel technologies are developing at a rapid pace, with improvements happening constantly for many industry and research solutions. In this environment, a comparative analysis needs to be done to determine which solutions fit to certain use cases.

In this deliverable orchestration solutions have been analysed to choose a solution that will fit the requirements set forth by the users. Four solutions have been taken into consideration and compared in terms of security, performance and features which are required for the project to function and integrate well with the user applications.

Unikernels have been described in the deliverable, with their advantages over regular virtualization and containerization being highlighted. Four unikernels have been compared to see the advantages of the different approaches in terms of ease of use, performance and integrability with other platforms. A particular emphasis has been placed about interfacing with hardware devices because this would allow users to run accelerated loads inside of unikernels, in a more secure and performant environment where certain compromises, such as security or compatibility, must be done to allow unikernels access to accelerators.

END OF DOCUMENT