Unikernel

Analysis of the possibility to map applications and small OS on virtual machine ready for migration/consolidation on edge nodes, when real-time constraints need to be taken into account

POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2025

Dipartimento di Elettronica, Informazione e Bioingegneria
Unikernel

Milan, 29/08/2025

Simone Calzolaro

MSc Student

CONTACTS

simone.calzolaro@mail.polimi.it
https://www.deib.polimi.it/eng/home-page

POLITECNICO MILANO 1863

Unikernel

Simone Calzolaro 260121

V1.2

29.08.2025

Introduction

Overview of the main scenarios and related problems

Cyber-Physical Systems scenario

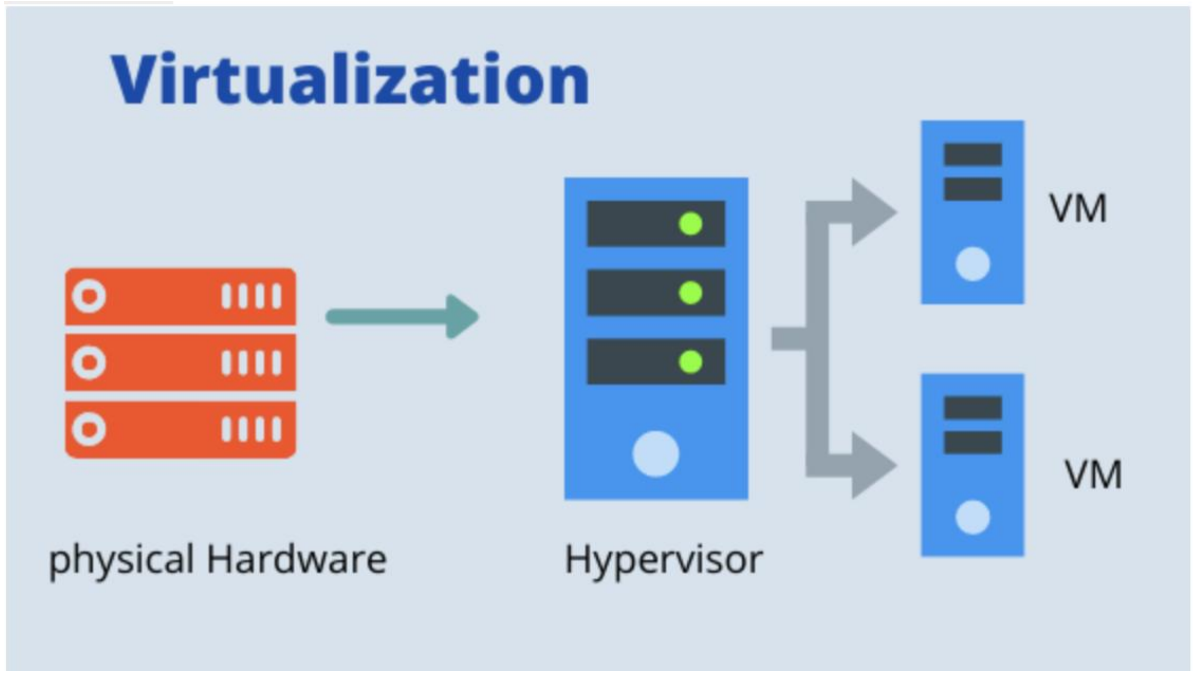
- Virtualization technology consolidates multiple systems as virtual machines (VMs) into the same platform

- It allows for:
    1. Cost reduction
    2. Increased efficiency
    3. Enhanced flexibility

- Virtualization was initially not designed to cope with strict timing constraints: in CPS like systems timing requirements are as important as functional correctness

Cyber-Physical Systems scenario

- Thus virtualization must be compatible to real-time software stack and satisfy the time constraints by employing hypervisor-level real-time scheduling policies



Cyber-Physical Systems scenario

- In data-centers and cloud environments each VM is expected to to host a general purpose operating systems to ease the effort of porting legacy software

- Legacy software has a lot of inherent libraries and functionalities

… it can be overkilled in some use cases ☹ …

- In fact, most CPS applications are **functionally dedicated single purposed**, thus not dependent on additional functionalities

Cyber-Physical Systems scenario

- Due to the encapsulation of superfluous libraries and the execution of non-essential processes, that are not related to the task of interest, deploying real-time tasks of CPS on a general purpose OS arises various issues:
  1. Resources efficiency
  2. Timing predictability

Cyber-Physical Systems scenario

- **Measured worst-case response time (WCRT) with unikernel VMs under Xen + deferrable servers (RTDS): (see later)**

VM	**Min**	**Mean**	**σ**	**WCRT (meas.)**	**WCRT (bound)**
VM1	1.00 ms	1.11 ms	0.047 ms	1.215 ms	1.25 ms
VM2	3.88 ms	4.42 ms	0.49 ms	5.83 ms	12.25 ms
VM3	7.86 ms	10.99 ms	2.02 ms	14.03 ms	26.25 ms
VM4	8.84 ms	15.54 ms	5.14 ms	27.83 ms	79.25 ms

Embedded Systems scenario

- In the era of Big Data, more and more applications of smart devices are computing-intensive
- Strong demand for task offloading to cloud data centers
- However this gives rise to network delay and privacy data leak issues

Embedded Systems scenario

- Edge computing can effectively solve:
    1. Latency
    2. Bandwidth occupation
    3. Data privacy problems

- However the deployment of applications are also limited by hardware resources

- With the fast development of Internet of Things more and more smart devices are connected to the Internet

Embedded Systems scenario

- These devices inevitably produce large amounts of data
- Moreover 5G technology can increase network bandwidth and reduce the latency of data transmission
- So we face the storage and computing pressure for big data and the geographical limitation of a cloud center
- In addition many latency-sensitive applications need milliseconds or even microseconds response time

Embedded Systems scenario

- Comparison between Docker containers and Unikraft unikernels:

	Cold-Start (boot)	Footprint
Docker	508.355 ms	109 MB
Unikraft	3.188 ms	1.9 MB

- **Why it matters at the edge:** milliseconds-level start and tiny images enable **on-demand spin-up** near sensors/actuators with limited storage and power

What is a Unikernel?

Description and definition of the technology,
main aspects, architecture and use cases

Basics of Unikernel

- Unikernel are sealed, single-purposed virtual machines (VMs) images that can be constructed using the concept of library operating system (LibOS)

- Unikernel instance is intended to run a single application, hence it include only a minimal amount of system software code required to execute that application

Basics of Unikernel

- This leads to a significant reduction in:
    1. Memory/Disk footprint
    2. Boot time
    3. Attack surface

$\Rightarrow$ When compared to traditional virtual machines

- LibOS's allow forn the tailoring of an OS code base to the particular needs of a given task

- Only those parts of the OS API are included in the VM image

Basics of Unikernel

- Unikernels are characterized by a minimal VM image size which highly increase their security properties

⇒ Due to the minimal attack surface for malicious code injection

- This also translates to a substantial reduction of overall system’s resource usage

- Moreover, Unikernels can be instantiated and become fully functional within only a few milliseconds

Basics of Unikernel

- Boot time and image size:

	Boot Time	Image Size
Container	508 ms	109 MB
Unikernel	3.19 ms	1.9 MB

Basics of Unikernel

- It has further been shown that unikernels are more efficient and safer than modern container technologies

- Unikernels have several advantages:
    1. High-security
    2. High start-up speed
    3. High simplicity

- It has attracted the researcher’s interest in the field of virtualization

- **Unikernel is the state-of-the-art virtualization technology**

Definition and Characteristics of Unikernel

- Unikernel is a specialized, single-address space image constructed by library operating systems (LibOSs)

- **Unikernel's essence is a kind of image file with special purpose**

- In order to support the execution of programs, developer select the smallest set of class libraries from a module stack, together with application code and configuration files

$\Rightarrow$ All built into a dedicated image file

- So Unikernels have a smaller code base and only run the components that are required for the specific application

$\Rightarrow$ Resulting in a smaller overall footprint

Definition and Characteristics of Unikernel

- Unikernels improve performance by removing unnecessary components from applications
- Unikernel is a single-purpose appliance that is specialized at compile time into a stand-alone kernel and is protected from modification after deployment
- Individual applications can be isolated in Unikernels, reducing the risk of cross-application interference
- They are self-contained and do not require a separate runtime environment

Definition and Characteristics of Unikernel

- So they can be deployed quickly and easily
- However, because of their limited runtime environment, unikernels are more vulnerable to security threats
- Due to the limited runtime environment and lack of standard tools, debugging unikernel can be difficult
- They also increase security by reducing the attack surface

Definition and Characteristics of Unikernel

- Unikernels may have a distinct set of vulnerabilities

⇒ An attacker who successfully penetrates one may be unable to threaten others

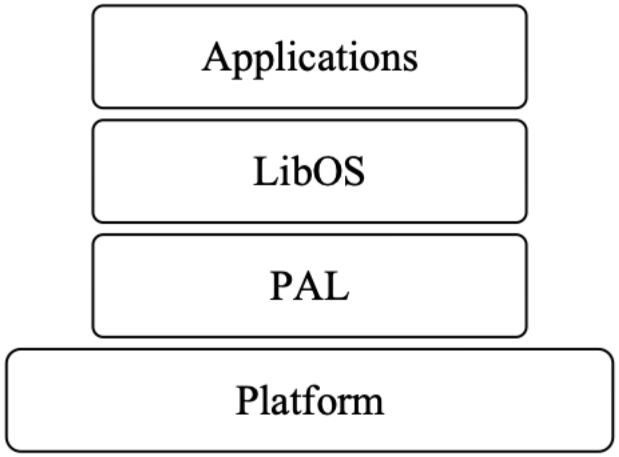
- Unikernel can run directly on such hypervisors such as:
    1. Xen
    2. Kernel-based Virtual Machine (KVM)
    3. Hardware without operating system

Definition and Characteristics of Unikernel

- Syscall surface comparison:
    1. Linux: exposes 393 syscalls
    2. Unikraft: exposes 146 syscalls
    3. Nanos: exposes 148 syscalls

Architecture of Unikernel

- Unikernel is divided into 3 layers over hardware platform or hypervisor:
    1. Application layer
    2. LibOS layer
    3. Platform Adaptation Layer (PAL)



Architecture of Unikernel

- User applications run at the application layer

- These application programs are generally:
    1. Single function
    2. Small size
    3. High-performance

⇒ Which are convenient for fast deployment

⇒ Their characteristics conform to microservice application

- LibOS is at the middle layer

Architecture of Unikernel

- It is specifically divided into a kernel base and functional system

- LibOS complete the basic functions of an OS:
    1. Task scheduling
    2. Thread management

- Platform Adaptation Layer are the minimum core components to support application execution

Architecture of Unikernel

- PAL includes:
    1. An adaptation layer with host environments running on a bare machine
    2. A virtual environment running on a hypervisor

- PAL is used mainly for LibOS application migration and need not change LibOS or application code when hardware is updated

Use cases of Unikernel

- Unikernel was originally intended for cloud computing
- However, its small footprint and flexibility make it ideal for other applications:
    1. Real-time applications: unikernels are lightweight and efficient, so they can handle real-time applications
    2. Edge-gateways: acting as a middleman between devices and the cloud
    3. Autonomous vehicles: unikernels can be used in self-driving cars to provide real-time data processing and decision making capabilities
    4. Cloud-Native Edge computing
    5. Network functions

Use cases of Unikernel – Short numeric sub-case

- Microservices/REST services
  1. **Throughput (single thread):**
     I. Go: +38%
     II. Java: +16%
     III. Python: +15%

  requests/s for OSv unikernel vs. Docker

  2. **Heavy compute (CPU-bound):**
     I. Go: 3% slower
     II. Java: 1% faster

The choice of Unikernels

- The choice of unikernels are driven by a variety of factors:
    1. Resource efficiency: unikernels are compact and efficient, making them excellent for edge computing applications where hardware resources are restricted
    2. Security: because of their small size unikernels have a lower attack surface, resulting in a higher degree of protection
    3. Isolation: provide good isolation between diverse workloads, they are ideal for instances where numerous tenants share the same hardware
    4. Simplified management: unikernel facilitate the management and deployment of applications at the edge by minimizing the overhead of standards OSs

Unikernel-Based Real-Time Virtualization under Deferrable Servers

Analysis and Realization

Introduction

- Real-time virtualization optimized the hardware utilization by consolidating multiple systems into the same platform

$\Rightarrow$ While satisfying the timing constraints of their real-time tasks

- We consider here virtualization based on Unikernels

- Each Unikernel is a guest operating system in the virtualization and hosts a single real-time task

- We consider deferrable server in the virtualization platform to schedule the unikernel-based guest operating systems and analyze the worst-case response time of a sporadic real-time task under such a virtualization architecture

Introduction

- Virtualization must be compatible to real-time software stack and satisfy the time constraints by employing hypervisor-level scheduling policies

- The Xen hypervisor, periodic server-based approach have been widely used

$\Rightarrow$ Especially **deferrable servers**

- Each virtual CPU (vCPU) is treated as one deferrable server assigned with an execution budget and a replenishment

Introduction

- The scheduling decision involves 2 levels:
    1. The hypervisor scheduler
    2. The scheduler within the VMs

$\Rightarrow$ In a hierarchical manner

- To account for the interplay of servers and tasks, the applicability of such server based approaches is based on the tightness of corresponding worst-case response time analyses

Introduction

- Main types of tasks in real-time systems:
    1. **Periodic**: Tasks that activate at **regular, fixed intervals**.
    2. **Sporadic**: Tasks that arrive **irregularly**, but with a **minimum inter-arrival time** between jobs.
    3. **Aperiodic**: Tasks that arrive **randomly**, with **no minimum time between jobs**.

Goals

1. We present why unikernel-based virtualization can facilitate the schedulability of deferrable servers

2. We explain how to realize our unikernel-based approach on top of the Xen hypervisor with a few design details

Deferrable Server and Task model

- Deferrable servers are adopted to preserve the required bandwidth of each virtualized CPS application, so called virtual machine (VM)

- Each VM is realized as a unikernel with one specific vCPU

- Each VM is treated as one deferrable server DS_i serving only one sporadic task τ_i

- One sporadic task release an infinite number of task instances, called jobs

$\Rightarrow$ In which the worst-case execution time (WCET) of any of them is at most C_i

$\Rightarrow$ And the arrival times of any two consecutive jobs of them must be separated by at least the minimum inter-arrival time T_i

Deferrable Server and Task model

- The jobs of task τ_i are served based on the first-come first-serve policy within DS_i
- Deferrable servers are scheduled based on preemptive fixed-priority (static-priority) scheduling

Service Condition and Execution Scenarios of Deferrable servers

- If the capacity of a deferrable server DS_k within the given replenishment period is feasible, we say that DS_k **fullfills its service conditions**

$\Rightarrow$ i.e. any request of Q_k amount of computation demand of its served task τ_k at the moment when the budget is fully replenished must be finished within P_k amount of time

- A DS_i may impose back-to-back interference to lower-priority servers or tasks

- A sufficient service condition test for DS_k is a sufficient test to validate whether DS_k fullfills its service conditions or not

Service Condition and Execution Scenarios of Deferrable servers

- Sufficient service condition for uniprocessor preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers:

$$\forall \mathrm{DS}_k, \quad \exists 0 < t \le P_k, \qquad Q_k + \frac{\sum_{\mathrm{DS}_i \in hp_k} \left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i}{m} \le t$$

- The equation $\sum_{\mathrm{DS}_i} \frac{Q_i}{P_i} \le \ln \frac{3}{2} \approx 0.40546$ ensures that the condition stated in the previous equation hold

Service Condition and Execution Scenarios of Deferrable servers

- Sufficient service condition test for multiprocessor systems, under partitioned scheduling, under global preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers on m homogeneous physical processors:

$$\forall \mathrm{DS}_k, \quad \exists 0 < t \leq P_k, \qquad Q_k + \frac{\sum_{\mathrm{DS}_i \in hp_k} \left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i}{m} \leq t$$

- Let $I(t)$ be the worst-case interference of the higher priority servers and or tasks for an interval length t

$\Rightarrow$ We will assume the sufficient service condition for DS_k as:

$$\exists 0 < t \leq P_k, \qquad Q_k + I(t) \leq t$$

Service Condition and Execution Scenarios of Deferrable servers

- We further need two additional properties based on finer granularity of the service provided by DS_k:
    1. **Worst-case response time** for requesting x amount of computation demand for $0 \leq x \leq Q_k$:

$$R^{-\text{DS}}_{k}(x) = \inf\{t | x + I(t) = t\}$$

    2. **Worst-case resumed time** for $0 \leq x < Q_k$ - longest time that DS_k finished x amount of computation demand and is scheduled to serve further demands if they exist:

$$\begin{aligned} R^{+\text{DS}}_{k}(x) &= \inf\{t | (x + \epsilon) + I(t) = t\} \text{ for infinitesimal } \epsilon > 0 \\ &= \inf\{t | x + I(t) < t\} \end{aligned}$$

Service Condition and Execution Scenarios of Deferrable servers

- At the job release of J there may be unfinished backlog $L(r_j)$
- Whenever there is available budget B(t) > 0 of the server DS_k the budget can be used to serve first the backlog L(t) and if the backlog reached L(t)=0 then the budget can be used to serve the computation demand C(t) of J.
- The first time the computation demand reaches 0 is called the finish f_j of J i.e. $C(f_j) = 0$
- If at the release r_j of the job there is enough budget to complete both backlog $L(r_j)$ and the computation demand $C(r_j)$ then the job J finishes as soon as additional $L(r_j) + C(r_j)$ budget is consumed

Service Condition and Execution Scenarios of Deferrable servers

- **Lemma 1**: if a job J of task τ_k at time r_j and the remaining budget of DS_k is higher than the execution demand $C(r_j)$ of J at time r_j plus backlog $L(r_j)$ from previous jobs of τ_k at time r_j then J finishes within $R^{-\mathrm{DS}}_{k}(L(r_J)+C_k)$ time units.

- **Definition 2 – Exhausted budget:** we say the budget B is exhausted by job J if there exist a point in time t such that the following conditions are met:
    1. There is a remaining computation demand C(t)>0 that wants to consume the budget
    2. The budget B(t)=0 has reached 0
    3. Instead the processor idles or a lower priority server or task is served

Service Condition and Execution Scenarios of Deferrable servers

- **Lemma 3:** if after a budget replenishment at time br the remaining backlog L(br) and the remaining computation demand C(br) can be fully served i.e. $Q_k \geq L(br) + C(br)$ then the job has a response time of at most $(br - r_J) + R^{-\text{DS}}_{k}(L(br) + C(br))$ time units

Worst-Case Response Time Analysis for One Single Task

- Assumptions:
  1. We assume $R-_{k}^{\text{DS}}(Q_k) \leq P_k$
  2. The utilization of task τ_k is assumed to be no more than the utilization of the deferrable server DS_k i.e. $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$
- Suppose that $r_k(t)$ is the accumulated workload in time interval [0,t) for task τ_k

Worst-Case Response Time Analysis for One Single Task

- Let S be some S>0 such that $r_k(s+S) - r_k(s) \leq \frac{Q_k}{P_k} \times S, \qquad \forall s \geq 0$

- Under the assumption that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$, setting S to $C_k \times \frac{P_k}{Q_k}$ ensures that:

$$S = C_k \times \frac{P_k}{Q_k} \leq C_k \times \frac{T_k}{C_k} = T_k \text{ and } r_k(s+S) - r_k(s) \leq C_k = \frac{C_k}{S} \times S = \frac{Q_k}{P_k} \times S,$$

$\Rightarrow$ i.e. the equation holds

- As for deferrable server DS_k, if it has a full budget at time t, its service provision from time interval t to $t + R^{-\text{DS}}_{k}(h)$ is at least h

Worst-Case Response Time Analysis for One Single Task

- Let h be the minimum value $\leq Q_K$ such that

$$\frac{h}{R^{-\mathrm{DS}}_{k}(h)} \geq \frac{r_k(s+S) - r_k(s)}{S}, \qquad \forall s \geq 0.$$

- Let H denote $R^{-\mathrm{DS}}_{k}(h)$ for brevity: Cuijpers and Bril showed that the worst-case response time of τ_k is upper bounded by S+2H

- With the above definition of H and S we can restate the worst-case response time analysis from Cuijpers and Bril for sporadic tasks

Worst-Case Response Time Analysis for One Single Task

- **Theorem 4:** suppose that the deferrable server DS_k fullfills its service conditions and that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$ then:

  1. S can be set to $C_k \times \frac{P_k}{Q_k} \leq T_k$
  2. H is upper bounded by $R^{-\text{DS}}_{k}(Q_k) \leq P_k$

  $\Rightarrow$ The worst case response time of a sporadic task τ_k served by DS_k is upper-bounded by:

$$C_k \times \frac{P_k}{Q_k} + 2R^{-\text{DS}}_{k}(Q_k) \leq C_k \times \frac{P_k}{Q_k} + 2P_k \leq T_k + 2P_k.$$

Worst-Case Response Time Analysis for One Single Task

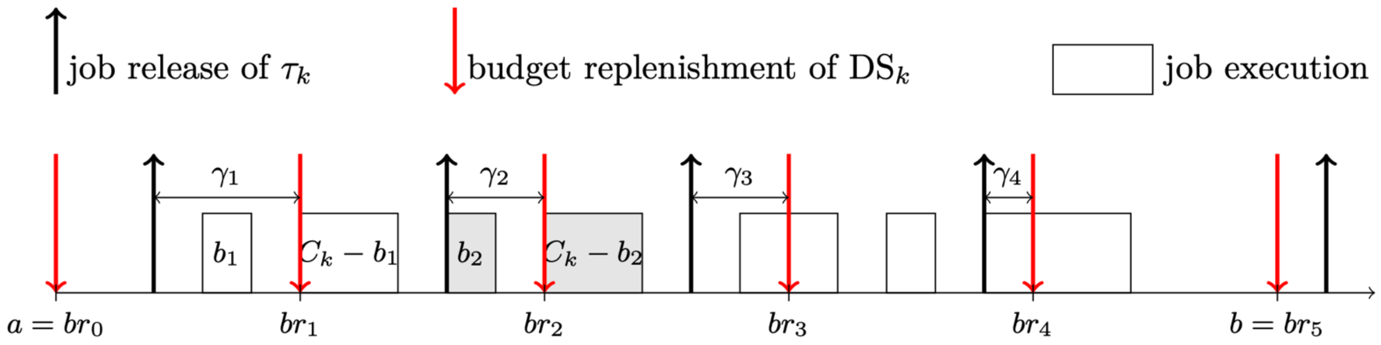
- **Definition 5 – Consecutive DS service interval:** an interval G=[a,b) is called a consecutive service interval if is a minimal interval such that the following properties are met:
    1. a and b are time instances where DS_k replenishes its budget
    2. All jobs of τ_k that are released during G finish their execution during G
    3. Only jobs of τ_k that are released during G can be executed

Worst-Case Response Time Analysis for One Single Task

- Consecutive DS service interval can be constructed in the following way: let a be a time instant such that
  1. DS_k replenishes its budget at a
  2. There's no unfinished job of τ_k at time a
  3. A job of τ_k is released during $[a, a + P_k]$

Then we set b to be the time of the next replenishment of DS_k, such that there is again not unfinished job of τ_k at time b.

$\Rightarrow$ **The interval [a,b) is a consecutive DS service interval**



Worst-Case Response Time Analysis for One Single Task

- **Lemma 6:** Under the assumption that $C_k \leq Q_k$ and $T_k \geq P_k$ the following holds:
    1. Between any two budget replenishments there is at most one job releease
    2. Every job finishes until the second replenishment period after the job release
    3. There is at most one previous unfinished job of τ_k at any job release of τ_k
    4. There are at most two jobs of τ_k executed between two consecutive budget replenishments of DS_k

Worst-Case Response Time Analysis for One Single Task

- **Theorem 7:** suppose that the deferrable server DS_k fullfills its service condition and that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$. If $C_k \leq Q_k$ *and* $T_k \geq P_k$ then:

$$R_k^{\tau} \leq \max\left((P_k - T_k) + \sup_{0 \leq x < C_k} (R^{+}{}_{k}^{\mathrm{DS}}(x) + R^{-}{}_{k}^{\mathrm{DS}}(C_k - x)),\ R^{-}{}_{k}^{\mathrm{DS}}(C_k) \right)$$

- **Theorem 8 – Dominance discussion:** the worst-case response time bound presented in Th. 7 dominates the bound from Th. 4 when $C_k \leq Q_k$ *and* $T_k \geq P_k$

Worst-Case Response Time Analysis for One Single Task

- For the computation of the worst-case response time upper bound presented in Th. 7 the supremum $\sup\limits_{0 \leq x < C_k} {R^+}_k^{\mathrm{DS}}(x) + {R^-}_k^{\mathrm{DS}}(C_k - x)$ has to be computed
- Let's discuss a method to do this efficiently without computing the values for ${R^-}_k^{\mathrm{DS}}(x)$ and ${R^+}_k^{\mathrm{DS}}(x)$ at every point using fixed-point iterations
- In particular the interection between the functions t -> t and t -> I(t)+x is computed

Worst-Case Response Time Analysis for One Single Task

- The efficient presentation and formulation presented in this section is based on the observation that ${R^-}_k^{\mathrm{DS}}(x)$ and ${R^+}_k^{\mathrm{DS}}(x)$ coincide and grow linearly if the intersection with I+x is on a plateau

Algorithm 1 Computation of all values in $\mathcal{S}$ with $c_i \leq C_k$.

Input: $I(t) = \sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \cdot \rho_i$
Output: The set $\mathcal{S}$ with all values (c_i, d_i) where $c_i \leq C_k$.

```
1: $\mathcal{S} := [\,]$; $x := 0$
2: **while** $x \leq C_k$ **do**
3:     Compute ${R^-}_k^{\mathrm{DS}}(x)$ and ${R^+}_k^{\mathrm{DS}}(x)$ by fixed-point iterations.
4:     $c := x$; $d := {R^+}_k^{\mathrm{DS}}(x) - {R^-}_k^{\mathrm{DS}}(x)$
5:     Add $(c, d)$ to the set $\mathcal{S}$
6:     $x := x + \min_i \left(-(x + j_i) \mod P_i\right)$        ▷ Time until next jump.
7: **return** $\mathcal{S}$
```

Architecture Model

- The most Cyber-Physical Systems (CPS) applications are specialized and functionally dedicated tasks and can be implemented as single-purpose appliances

- **CPS applications provide an excellent target for unikernels**

- The case-study results are linked to the **Xen hypervisor**, so we are going to see briefly what’s Xen and how it works

Architecture Model: Overview on Xen hyervisor

- Xen is a type 1 hypervisor and allows for consolidation of multiple systems on a single platform
- Xen runs directly on host’s hardware and is the first software layer to execute after the bootloader
- The hypervisor is responsible for managing hardware resources, including CPUs and memory
- It also handles timers and scheduling of VMs

Architecture Model: Scheduling Architecture

- Specific to Xen is a privileged VM called Domain 0
- Domain 0 is the first VM to load under Xen, and holds the **drivers** to the underlying hardware and this is also where the toolstack resided that enables management of further VMs
- Domain 0 is typically deployed on Linux
- The drivers consist of two parts:
    1. Front-end: situated in the guest VM
    2. Back-end: resided in Domain 0

$\Rightarrow$ Both are isolated and communicate through shared memory

Architecture Model: Scheduling Architecture

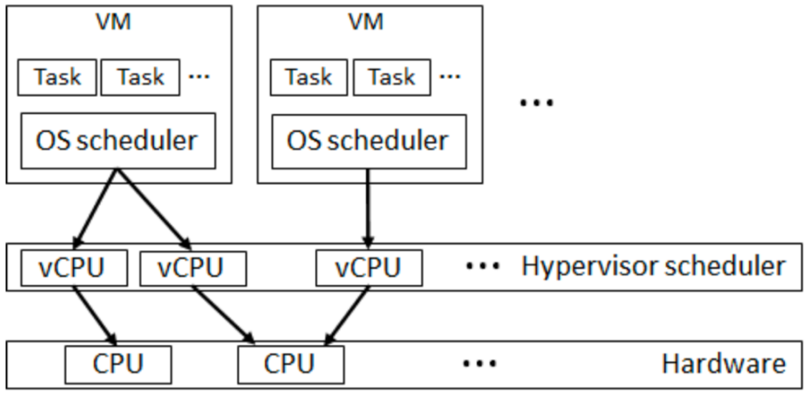
- The scheduling decision on Xen is split into two tiers:
    1. Bottom tier: constituted by the hypervisor scheduler which assigns the vCPUs to pCPUs
    2. Upper tier: consists of the guest operating system schedulers within VMs, which in turn assigns their threads to vCPUs

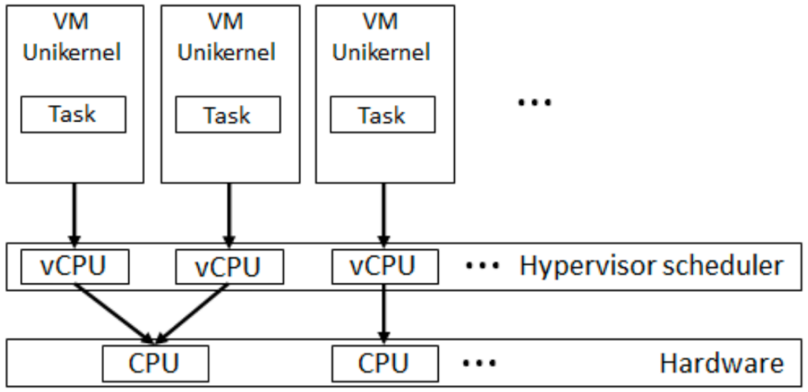
- The splitting is needed for 2 reasons:
    1. Possibility to abstract physical resources into logical resources
    2. Allows the hypervisor to enforce timing isolation between the concurrently running VMs

Architecture Model: Scheduling Architecture

- This architecture allows for high resource utilization while at the same time preventing any fault or compromised VM from hijacking system resources
- One sporadic task per server aligns well with the concept of unikernels by which CPS applications are deployed as single tasked VMs
- Due to the fact that each of our unikernels host a single task, there is no point of assigning more than one vCPU per unikernel

Architecture Model: Scheduling Architecture





Architecture Model: Real-Time Networking

- CPS applications commonly assume distributed architectures

- The I/O processing of network packets is a matter of particular importance

$\Rightarrow$ Xen handled packet processing in Domain 0 where the network driver resides

- Each instantiated VM under Xen is connected to a dedicated virtual network interface (VIF) and a corresponding dedicated VIF-thread

Architecture Model: Real-Time Networking

- By default the VIF-thread in Xen are scheduled independently of the priority of their VMs

⇒ This can lead to priority violation: the order of packet processing mismatches the priority of vCPUs

- In order to solve this issue, we align the priorities of the packet processing threads with the corresponding priorities of the vCPU in the hypervisor scheduler

Architecture Model: Design Principles

- In our model each vCPU is implemented as a deferrable server, described by its capacity and replenishment period
- Due to the adaptation to unikernel each vCPU has only one task assigned to it
- The vCPU is released under the sporadic task activation
- **Budget**: amount of time the vCPU is set to the given capacity at its **replenishment** periodically

Architecture Model: Design Principles

- The budget of the vCPU is set to the given capacity at its replenishment periodically
- The vCPU can either be:
    1. Runnable: it consumes its budget
    2. Blocked: it doesn’t consumes its budget
- A CPU with depleted budget will not be scheduled
- If there are no eligible vCPUs, the hypervisor will schedule a idle vCPU

Architecture Model: Design Principles

- As vCPU are implmented as deferrable servers, they can defer their budget to be used at a later time
- However the budget cannot be preserved and transferred into the next period => budgets that were not consumed during their current periods are lost
- Our implementation relies on partitioned queues i.e. each pCPU possesses and manages its own run queue of vCPUs
- Priorities are statically assigned to the vCPU according to their replenishment periods following the *rate-monotonic (RM) policy*

Architecture Model: Design Principles

- The utilization bound ln(3/2) ≈ 0.40546 guarantees the sufficient service condition of the deferrable servers on one physical processor
- Each time a vCPU is assigned to a run queue, the vCPU is inserted accordingly to its priority
- In the case of the RM algorithm the highest priority is given to the VM with the shortes replenishment period

Architecture Model: Implementation on Xen

- The Xen hypervisor provides an interface to schedulers by exposing an abstract scheduler struct which contains pointers to functions which have to be implemented when adding a new scheduler on Xen

- The scheduler policy independent code is situated in the schedule.c file

- The most important function in schedule.c is *schedule()*

- In order to choose the next vCPU to run, it deschedules the currently running vCPU and calls the specialized function *do_schedule()*

Architecture Model: Implementation on Xen

- In our model we have extended Xen with out own scheduler that implements our deferrable server model

- *do_schedule()* functions:
    1. Updates the bookkeeping (consumed budget) for the currently running vCPU
    2. Chooses the next vCPU to run from the top of the run queue

    => In case there are no eligible vCPUs to run, the algorithm returns the idle vCPU

Architecture Model: Implementation on Xen

- Budget replenishment functionality is transferred into timers
- For each of the vCPUs a timer is instantiated with a period that equals the replenishment period of the task
- The replenishment itself is happening inside the timer handler

Architecture Model: Implementation on Xen

Algorithm 2 Pseudocode of the do_schedule() function from our deferrable server scheduler.

```
 1: next = null
 2: consumeBudget(getCurrentVCPU)
 3: for each vCPU in RunQ do
 4:     if vCPU.isRunnable then
 5:         if vCPU.hasBudget then
 6:             next ← vCPU
 7:             break
 8: if vCPU == null then
 9:     next ← idleVCPU
10: return next
```

Experiment Setup

- Platform: Intel NUC i5-8259U (4 cores @ 2.3 GHz), Turbo/HT/power mgmt disabled; Ubuntu 20.04 in Dom0
- Hypervisor: Xen 4.14.1 with a custom Deferrable Server (DS) scheduler; partitioned RM (one vCPU/server per physical core queue)
- Workload: 4 periodic tasks deployed as unikernel VMs on one pCPU; Dom0 on a separate pCPU

Experiment Setup

- Networking alignment: VIF-thread priorities aligned with VM priorities to avoid priority inversions in packet processing

- Network stack overhead (host): worst-case ≤ 250 µs, average 157 µs—included in WCRT bounds

Measurements Methodology

- Benchmark: UDP client–server; client triggers computation in each unikernel; server replies when done
- Three latencies captured per request/response:
    1. Task response time (release→finish) via SystemTap hooks at L2 in Dom0 TCP/IP
    2. Task execution time (in-VM), cycle-accurate (used to sanity-check WCET vs hardware)
    3. Client round-trip time for plausibility checks
- Samples:
    1. 22,000 response-time measurements
    2. Bounds adjusted by ≤ 250 μs worst-case host networking latency

Case-Study Results: Tasks and Servers

VM	Task (T, C) [ms]	DS (P, Q) [ms]	WCRT bound (incl. net)
VM1	T=12, C=1	P=10, Q=2	1.25 ms
VM2	T=20, C=4	P=20, Q=4	12.25 ms
VM3	T=60, C=8	P=50, Q=10	26.25 ms
VM4	T=130, C=9	P=100, Q=10	79.25 ms

*Each VM hosts a **single task** and the budget is consumed only while serving that task.*

Case-Study Results: Measured vs Bound

- No outliers across 22k samples → deterministic behavior under DS scheduling

- Measured WCRTs (incl. network) vs computed bounds:
  1. VM1: 1.215 ms (bound 1.25 ms), σ 0.047 ms
  2. VM2: 5.83 ms (bound 12.25 ms), σ 0.49 ms
  3. VM3: 14.03 ms (bound 26.25 ms), σ 2.02 ms
  4. VM4: 27.83 ms (bound 79.25 ms), σ 5.14 ms

- Observation: bounds hold for all VMs; conservatism increases at lower priorities (larger periods)

Numerical Simulation: Tightness of Analysis

- Goal: compare new WCRT bound (Theorem 7) vs RTC-derived bound (Theorem 4)

- Synthesis (per test):
  1. $n \in \{10, 50, 100\}$ servers; UUniFast for U_i summing to U.
  2. P_i log-uniform in [1, 100] ms; $Q_i = P_i \cdot U_i$; T_i uniform in [1.0 P_i, 1.5 P_i]; C_i uniform in [0.5 Q_i, 1.0 Q_i]
  3. Priorities by rate-monotonic

- Service condition constraint: $U = \Sigma(Q_i/P_i) \leq \ln(3/2) \approx 0.40546$ to ensure DS service feasibility

- Results (boxplots of WCRTOur / WCRTSOTA):
  1. Median improvement ≈ 90.5% (across n = 10, 50, 100).
  2. At U=0.40, median improvement ≈ 84.4% (still strong).
     A. Trend: gap closes as U increases, but new analysis remains substantially tighter within service-feasible region.

What These Numbers Means: Takeaways

- Feasibility in practice: unikernel-per-task + DS on Xen yields predictable response times; measured WCRTs < bounds for all VMs
- Networking accounted for: host network stack adds ≤ 0.25 ms (avg 0.157 ms) and is included in the analysis for fair end-to-end bounds
- Analysis tightness: new bound is ~85–90% tighter than RTC-derived bound in the service-feasible region ($U \leq \ln(3/2)$), enabling higher utilization without violating deadlines
- Design implication: “one task per unikernel per vCPU” simplifies timing (no guest scheduler), reduces jitter, and aligns with RM-based DS provisioning

Conclusions

- In this work we proposed to leverage the scheduling architecture of unikernel-based virtualization to facilitate the schedulability of deferrable servers
- We presented how to derive the worst-case response time analysis under practical scenarios
- **The evaluation results of the experiment show that out analysis outperforms the restated analysis based on the state-of-the-art**
- In addition we showed that the unikernel-based architecture can be effectively implemented on top of the Xen hypervisor

Major Unikernel Distributions

Brief overview about the main software distributions

MirageOS

- **MirageOS** is a LibraryOS Unikernel compiling Ocaml apps into a sealed VM image
- Runs on Xen hypervisor (paravirtualized) and via Solo5 on KVM-like monitors
- Supports x84-64 and ARM architectures
- Designed for cloud but also applied in edge scenarios

OSv

- **OSv** is a general-purpose unikernel OS optimized for running a single linux application on a VM

- Supports KVM/QEMU and Xen mainly x86-64

- Provides a POSIX-like environment to run high-level language runtimes (JVM, Node.js …) in the cloud or edge

IncludeOS

- **IncludeOS** is a minimal unikernel written in C++ for cloud services
- Targets x86-64 virtual hardware and runs on KVM/QEMU
- It focuses on small footprints and fast boot
- Suitable for network functions or microservices on edge servers

Rumprun (NetBSD Rump Kernels)

- Unikernel that uses NetBSD’s rump kernel to run POSIX applications as unikernels
- Can run on Xen or KVM and reuse NetBSD drivers
- It has been used for network appliances e.g. ClickOS uses rumprun base

Hypervisor and Hardware Platform Compatibility

Brief overview about the main platform supports

Xen Hypervisor

- Widely used for unikernels, especially in early research
- Xen’s paravirtualization allows unikernels to run with minimal emulation overhead
- Many unikernels were designed for Xen on x86 and ARM
- Xen supports real-time scheduling to meet CPS timing constraints

KVM and MicroVMs

- Most unikernels run on KVM/QEMU as standard x86 or ARM VMs

- Lightweight VMMs build on KVM to launch microVMs with minimal device emulation

⇒ An ideal match for unikernels' small footprint

- Combining unikernels with microVMs in order to maximize isolation and startup speed in FaaS platforms

Bare-Metal and Other Platforms

- Unikernels generally rely on a hypervisor but some can run directly on hardware or as unikernel processes
- Unikernel images can run on a hypervisor or on bare hardware, not requiring a host OS
- The Unishyper project boots on embedded ARM/RISC-V devices without an OS, using a thin platform layer
- Some unikernels can also run as a process on Linux for development

CPU Architectures

- Initial unikernels targeted x86-64 but now many support ARM for IoT and mobile edges
- RISC-V support is emerging e.g. Unikraft and Unishyper include RISC-V as a target architecture
- Multi-platform support is becoming common in newer unikernels, reflecting the heterogeneous hardware in edge environments

Benefits of Unikernels for Edge Computing

Description and motivation of the advantages of unikernels for the edge computing field

Minimal Footprint and Fast Boot

- Unikernels compile only the necessary OS functions and the single application into one image

⇒ Extremely small VM images and rapid boot times

- For edge scenarios this means that more instances can run on resource-constrained nodes and services can scale up on-demand with virtually no startup delay

- Unikernels can cold-start and be fully functionl within a few milliseconds

⇒ Far faster than traditional VMs

High Security by Design

- Tiny attack surface by removing all unused OS components
- Isolation and reduction of code results in **fewer vulnerabilities**
- Studies note that a minimale unikernel image greatly increases security properties compared to general purpose OSs

Real-Time Performance

- Unikernels avoid the overhead of a full OS, which benefits real-time behaviour
- With a single address space and no kernel/userspace transitions, interrupt and scheduling latencies can be lower and more predictable
- Each unikernel typically runs one application thread pinned to a vCPU, eliminating in-guest scheduing variability

Efficiency and Scalability

- Running only what is needed yields extremely low memory and CPU overhead
- Unikernels typically consume an order of magnitude of memory less than a Linux VM or even a minimal container
- This means that edge nodes can run more services concurrently

Tailored Functionalities

- Unikernels are single-purpose by design which fits with the microservice model often used at the edge
- Developers select only the libraries needed, resulting in a custom OS for each application
- This can be tuned for performance or footprint, it also improve **adaptability**
- In edge computing where applications might be specialized, the ability to tailor the OS per app is a strength

Numbers at Glance

Some numerical results about unikernels

Startup and Footprint vs performance

- Cold-Start and Image size:
  1. Image: 1.9 MB (unikernel) vs. 109 MB (container)
  2. Boot: 3.188 ms (unikernel) vs. 508.355 ms (container)

- Throughput/compute:
  1. REST (single thread):
     i. Go: +38% req/s
     ii. Java: +16% req/s
     iii. Python: +15% req/s

  $\Rightarrow$ Respect to containers
  2. CPU-Bound:
     i. Go: -3%
     ii. Java: +1%

Startup and Footprint vs performance

- Real-Time Xen + RTDS:
  1. WCRT measures: 1.215 - 27.83 ms across 4 VMs
  2. Host network stack: $\leq 250\ \mu s$

Startup and Footprint vs performance

Metric	Container	Unikernel	Delta
Image size (Nginx)	109 MB	1.9 MB	−98%
Cold-start (Nginx)	508.355 ms	3.188 ms	~169× faster
REST throughput (Go)	baseline	+38%	+38%
REST throughput (Java)	baseline	+16%	+16%
REST throughput (Python)	351 ± 1 rps	395 ± 1 rps	+15%
CPU-bound	—	≈ parity	Go −3% / Java +1%

Benchmark and methods

- REST testbed:
    1. Java (Vert.x)
    2. Go (net/http+Gorilla)
    3. Python (Flask)

    $\Rightarrow$ 40 threads × 50k requests

    $\Rightarrow$ OSv unikernels vs Docker

- CPU-bound:
    1. Bubble-sort workload
    2. Log-scale timing
    3. Same languages

Benchmark and methods

- Edge/RT case:
    1. Request/response timestamps inside Dom0 TCP/IP execution time clock-cycle precise
    2. 22k measurements

- Boot/size test:
    1. Prebuilt images: Unikraft and Docker Hub
    2. ns-level boot script
    3. du -s --si for image size; wrk+Prometheus/Grafana for load stats

Limitations, Future Outlook and Emerging Trends

Overview of the current limitations, future prospects and roadmap

Limitations

- **Orchestration and tooling:** no Docker/K8s-grade ecosystem yet; formats and build tools differ across projects, so deploying/updating large fleets is still manual and brittle

- **Developer Adoption and Ecosystem:** niche languages/APIs and scarce tooling slow onboarding; porting apps and debugging without shells/SSH is cumbersome

- **Fragmentation and Standardization:** many incompatible approaches and ABIs; lack of standards makes swapping or orchestrating unikernels hard

Limitations

- **Language and Compatibility Constraints** : often language-specific with partial POSIX; running existing binaries is experimental, so teams may maintain separate code paths
- **Debugging and Monitoring:** minimal environments lack familiar probes (strace/perf,/proc); you rely on hypervisor tracing/instrumentation, raising ops effort
- **Peripheral and Device Support** : limited drivers beyond virtio; true bare metal needs custom NIC/storage/sensor drivers or offloading to a host OS, adding complexity

Future Outlook and Emerging Trends

- **MicroVM and Container Ecosystem Integration:** converging to VM isolation with container agility; expect build/orchestrate flows that feel like Docker/K8s
- **Scalability and Orchestration:** unified schedulers, image registries, and lifecycle tools tailored to unikernels plus standardized image metadata
- **Real-Time and Safety-Critical Adoption:** better RT schedulers and timing introspection could bring unikernels into mixed-criticality and certified domains

Future Outlook and Emerging Trends

- **Security and Isolation Enhancements:** intra-unikernel isolation (MPK/MTE), immutable builds, and memory-safe stacks (e.g., Rust) to harden edge deployments
- **Edge-Native Apps and Libraries:** “Compile-to-unikernel” frameworks and accelerator-aware images (e.g., TFLite) to make ML and edge services turnkey and dense

Real-World Deployment Scenarios

- **IoT Gateways and Edge Appliances:** tiny, fast-boot services for protocol translation, filtering, and encryption; strong isolation limits blast radius on compromised nodes

- **5G/6G Telco Edge (MEC/NFV):** lightweight, quickly (cold)-startable network functions (NAT, FW, UPF) at cell sites; lower attack surface and on-demand provisioning

- **Autonomous Systems and Robotics:** deterministic, low-overhead control/sensor modules per task; quick startup and isolation suit compute-constrained robots/drones

Real-World Deployment Scenarios

- **Serverless / FaaS:** paired with microVMs for millisecond cold starts and high density; “unikernel-as-a-function” reduces per-invocation overhead.

- **Embedded and CPS:** multiple isolated RT tasks consolidated on one board under a hypervisor; analyzable timing with strong fault isolation.

References

Peer-reviewed papers

[1] K.-H. Chen, M. Günzel, B. Jablkowski, M. Buschhoff, and J.-J. Chen, “Unikernel-Based Real-Time Virtualization,” *ECRTS 2022*, LIPIcs, pp. 6:1–6:22, 2022. doi:10.4230/LIPIcs.ECRTS.2022.6

[2] C. G. Paulus, D. Socci, C. Marquezan, and L. R. P. M. Vitório, “Unishyper: Safe, memory-efficient, and fast hypervisor using Rust,” *Middleware ’23 Companion*, ACM, 2023.

[3] H. Zhao, Z. He, and L. Wang, “An Edge Computing Architecture Based on Unikernel,” (preprint / proceedings copy).

[4] A. Marnerides, M. Psarakis, *Enhancing Edge Computing with Unikernels in 6G Networks*, University of Manchester repository, 2023.

[5] A. Ahuja and V. Jain, “Challenges and Opportunities for Unikernels in Machine Learning Inference,” *Proc. 9th Int’l Conf. on Reliability, Infocom Technologies and Optimization (ICRITO)*, 2021.

[6] A. A. Nizar, S. A. Karimah, and E. M. Jadied, “Analysis of Virtualization Performance on Resource Efficiency Using Containers and Unikernel,” *2024 Int’l Conf. on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*, 2024.

[7] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, “Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications,” *2018 IEEE 8th Int’l Symposium on Cloud and Service Computing (SC2)*, 2018

[8] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis, R. Atkinson, and X. Bellekens, “A Security Perspective on Unikernels,”.

[9] A. Wollman and J. Hastings, “A Survey of Unikernel Security: Insights and Trends from a Quantitative Analysis,” *Cyber Awareness and Research Symposium (CARS)*, 2024.

Official Project sites

[10] OSv unikernel: official site and docs.

[11] MirageOS: project home (OCaml unikernels) / org repos.

[12] IncludeOS: project resources (C++ unikernel) — current status mostly via project repos/news.

[13] NanoVMs / Nanos: vendor and OSS resources (Rust/C apps as unikernels; tutorials).

[14] Solo5: unikernel sandbox / base layer (docs & architecture).

[15] Xen RTDS (Real-Time Deferrable Server) scheduler: overview, docs, and tooling.