Reading Group Fall 2024

TEXAS
The University of Texas at Austin

Perspectives on Learning-Directed Operating Systems

Christopher J. Rossbach

The University of Texas at Austin

Outline

- Vision: Learning-directed OSes
- LAKE: early experience supporting ML in the OS
- Some practical challenges
- Addressing those challenges with LDOS

LDOS: A Clean-Slate Paradigm for OS Design and Implementation

Aditya Akella, Alex Dimakis, Swarat Chaudhuri, *Chris Rossbach,* Sebastian Angel, Joydeep Biswas, Shuchi Chawla, Isil Dillig, Brighten Godfrey, Daehyeok Kim, Sanjay Shakkottai, Michael Swift, Shivaram Venkataraman, and Gang Wang

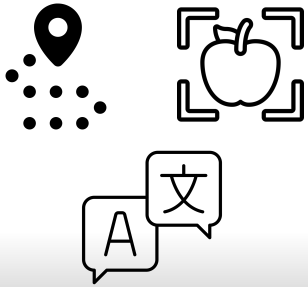
TEXAS
The University of Texas at Austin
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Penn
University of Pennsylvania
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

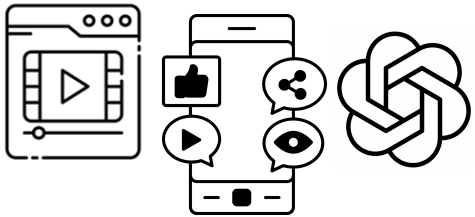
Vision

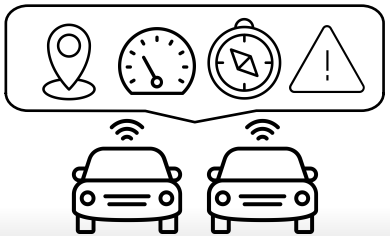
To build an intelligent, self-adaptive OS
that optimally meet modern applications' strict needs
in dynamic and complex scenarios.

Apply Machine Learning to achieve maximal efficiency and enable transformative use cases that otherwise impose prohibitive cost and effort.

OSes: Core Computing Infrastructure



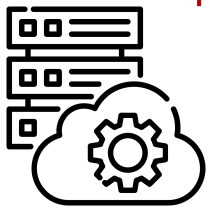


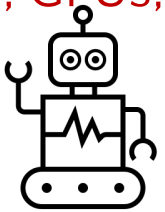


Applications

Operating System (OS)
Resource management: scheduling, allocation

Computer systems
Resources: CPUs, GPUs, Network, Memory…





OSes: Core Computing Infrastructure

Real-time

High throughput

Diverse resource needs

Increasingly demanding

Operating System (OS) / Runtimes

Effective resource management is crucial!

Complex, dynamic conditions

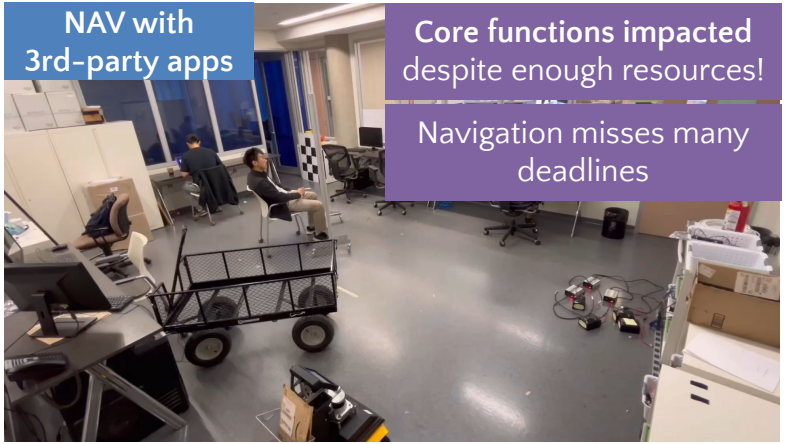
Complex environments

Constant change

OS Resource Management is Broken

Cannot support modern concurrent apps in complex, dynamic settings

Consumer autonomous robots



OS Resource Management is Broken

Cannot support modern concurrent apps in complex, dynamic settings

Consumer autonomous robots

3rd party added-function apps can't run alongside core apps

Cloud computing

Poor dependability

Mobile access edge

Multiplexed real-time apps suffer

Overprovisioning and Custom Design

Extra CPUs, GPUs, memory

Wasteful, prohibitively expensive, infeasible

Cloud utilization ~ 10-20%

Alibaba SoCC'21; Azure SOSP'17

Untenable ecological impact

Cloud carbon footprint > the aviation industry

Near-real-time kernel

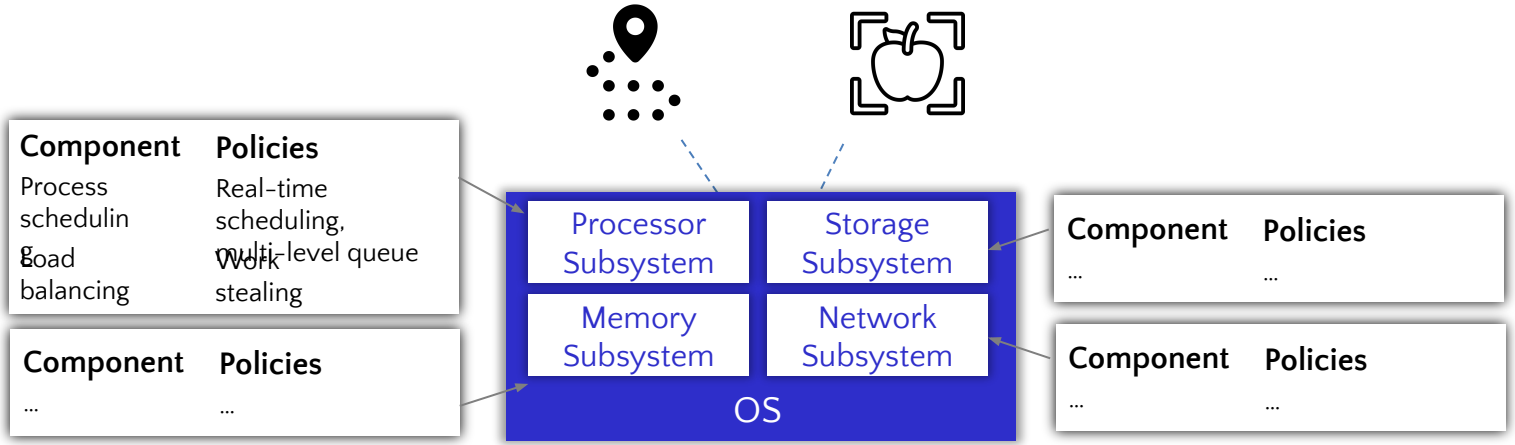
Long time to develop

Easily become obsolete

MIT Press:
http://t.ly/9Xwzv

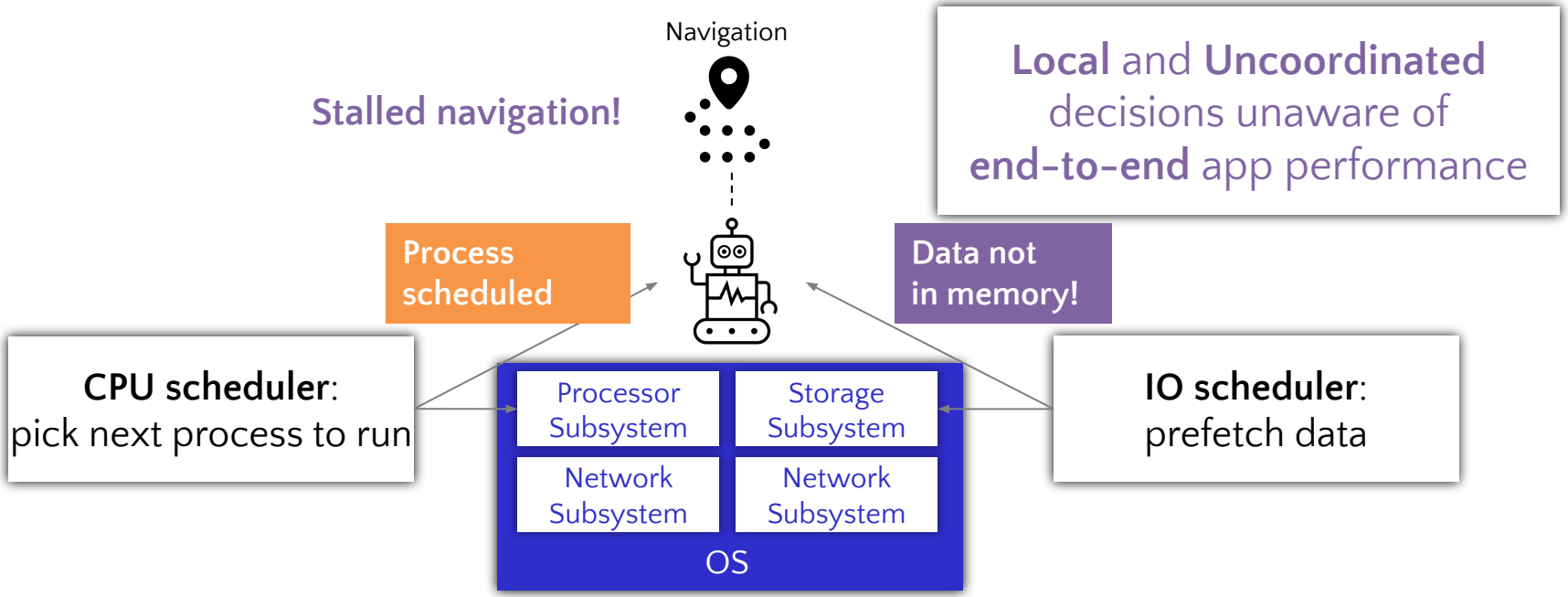
Why is Resource Management Broken?

Resource management **policies**



Policy architecture has two fundamental drawbacks

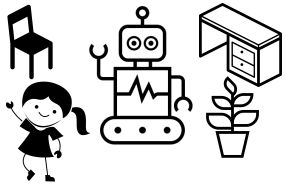
Poor Composability of Decisions



Limited and Rigid Decision-Making

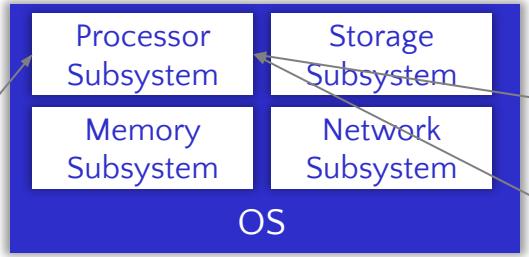
Cannot **adapt** to changes

Poor decisions amid **complexity**



Heuristics with fixed logic

Multi-level feedback queue



Component-specific inputs

Input: process CPU usage, …

Decisions for next time step

An Impending Crisis

Optimal decisions exist per setting, but

- Infeasible for humans to find
- OSes cannot express them due to policy drawbacks

The best today's OSes can do vs. optimal: gap can be significant

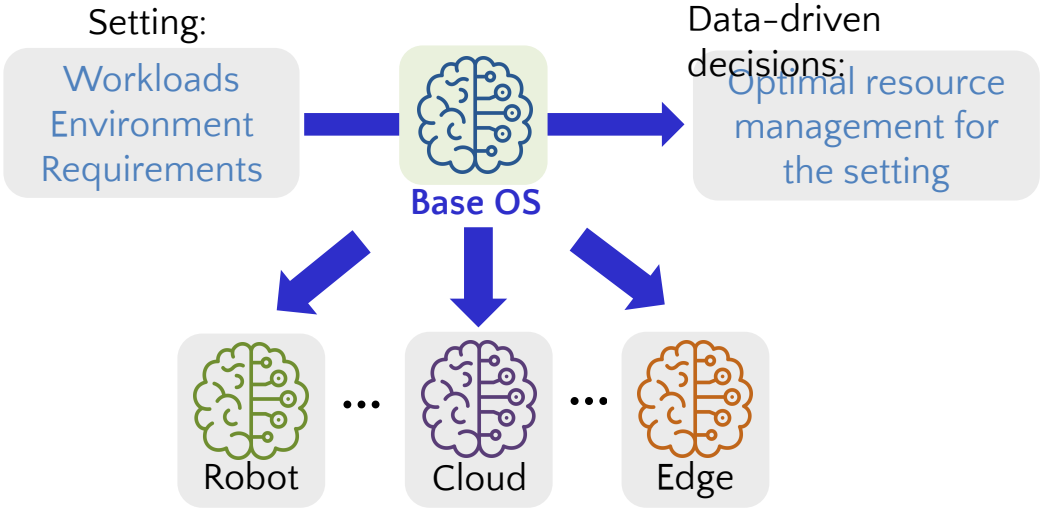
An unmanageable crisis is brewing

Performance and efficiency needlessly sacrificed to ever-greater degrees despite substantial human engineering effort

Intelligent, Self-Adaptive OSes and Runtimes

Intelligence

Machine Learning-based end-to-end resource management



No more overprovisioning!

No custom OS design or tuning!

Self-adaptation

Single base that auto-adapts to different settings

Learning-Directed Operating System (LDOS)

Clean-slate OSes / runtimes with a new ML-driven resource management policy architecture

Cross-cutting approach that fundamentally advances generative AI, ML, formal methods, and systems

Gen AI, ML, and formal methods innovations

- ML policies driven by rich run-time data, and trained using diverse synthetic data
- Policies that compose to guarantee strict end-to-end app and system needs

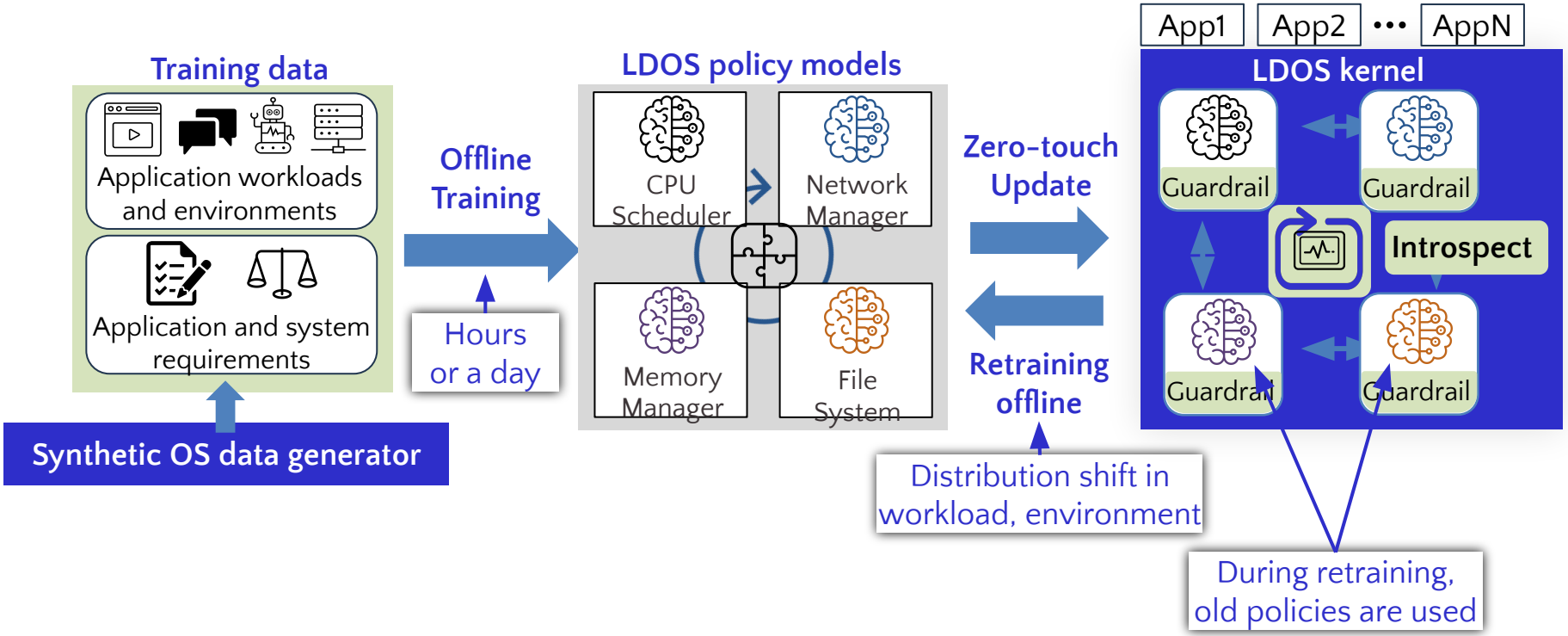
ML-centric OS design

- Easy policy integration, automatic adaptation
- Low overhead, security, manageability

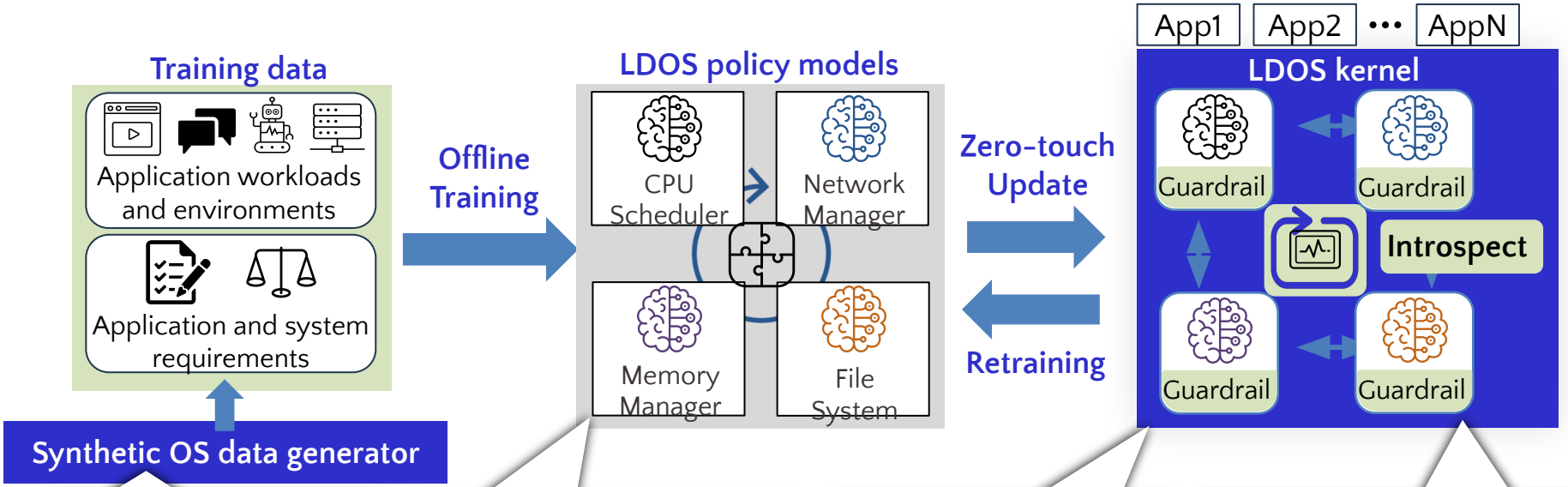
Pragmatic phased approach

- Substantial benefits per phase

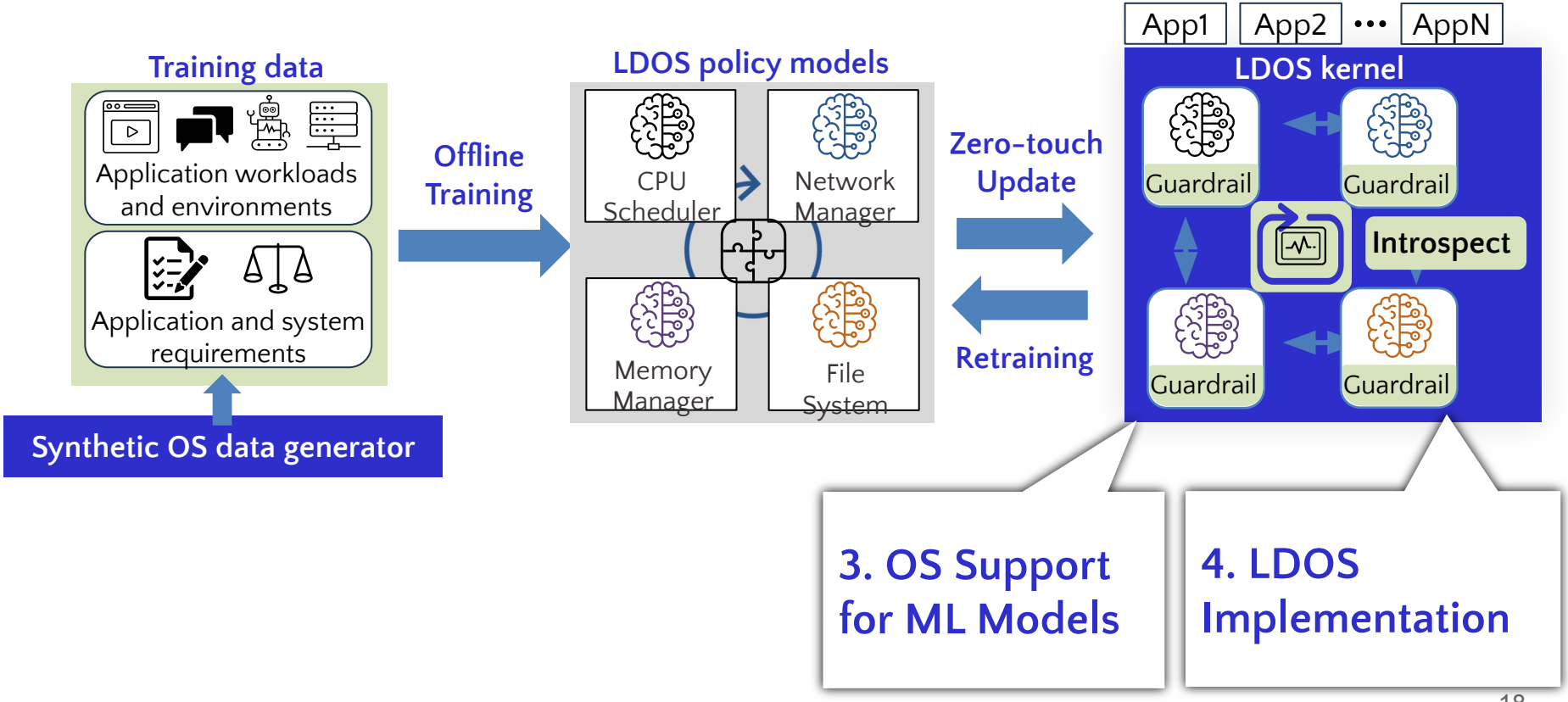
LDOS Approach



Research Thrusts



LAKE: Early Experience



ASPLOS 2023

TEXAS
The University of Texas at Austin

Towards a Machine Learning-Assisted Kernel with LAKE

Henrique Fingler*, Isha Tarte *, Hangchen Yu †, Ariel Szekely ‡,
Bodun Hu *, Aditya Akella *, Christopher J. Rossbach *§

*The University of Texas at Austin †Meta ‡ Massachusetts Institute of Technology § Katana Graph

ML to Replace OS Heuristics

- **KML:** (HotStorage ‘21) – storage management
- **SmartOS:** (APSys ‘21) – resource allocation
- **MLLB**: (APSys ‘20) - load balancing
- **LinnOS:** (OSDI ‘20) – IO scheduling
- **Kleio:** (HDPC ‘19) - page scheduling
- **Lynx:** (NVMSA ‘16) - File prefetching
- On the Feasibility of Online **Malware Detection** with Performance Counters (ISCA ‘13)
- Others..

Conjecture:

- ML will be used in OSes
- ML can adapt to dynamic demand

LAKE: ask fundamental questions to ***explore challenges, implications for OS design***

[1] https://qumulo.com/blog/non-uniform-memory-access-numa/
[2] https://core.vmware.com/blog/understanding-persistent-memory-pmem-vsphere

ML in the OS: Challenges

1) ML is compute expensive

- Subsystem may be latency-sensitive
- Can not contend for resources with user applications
- We are currently stuck with small and cheap models

2) Capturing features for inference and training is difficult

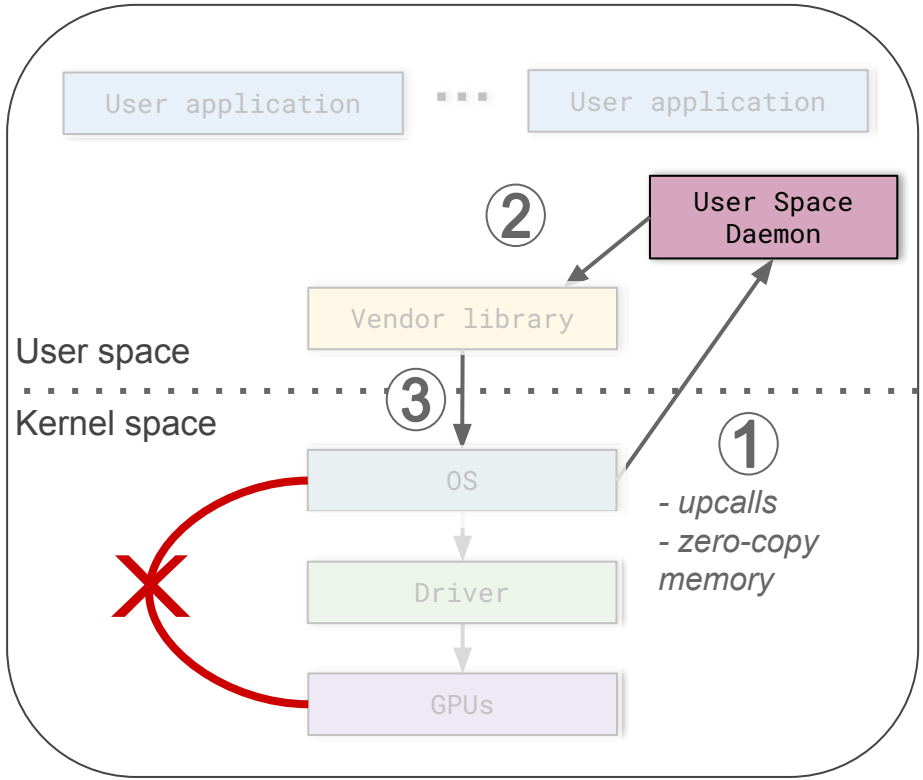
- Scattered code and data structures
  - Prone to critical system level bugs
  - Aggravated by ML subsystem composition
- Locking discipline
  - Features can be in different kernel subsystems
  - Careless locking may deadlock the system

Key Innovations in LAKE

LAKE's contributions:

- ML models are compute expensive → Enable hardware acceleration in kernel
- CPU and accelerator contention → User-defined policies for adaptive control
- Difficulty to manage feature vectors → In-kernel feature registry

Supporting Acceleration in OSes



Hardware acceleration: key

LAKE enables accelerators in the OS through upcall-based API remoting

LAKE prototype provides CUDA APIs to OS

GPU is **inaccessible** because libraries:

- are designed for user space
- may use **kernel bypass** communication
- wrapped in user APIs

1. API calls are remoted to user space
   Bulk transfers are zero-copy

Example: *cudaMemcpy*

- IO control calls are opaque
  - e.g. command queues
- Need prop

2. Daemon calls original accelerator API

3. Accele
   then r

LAKE's API
provide a Te

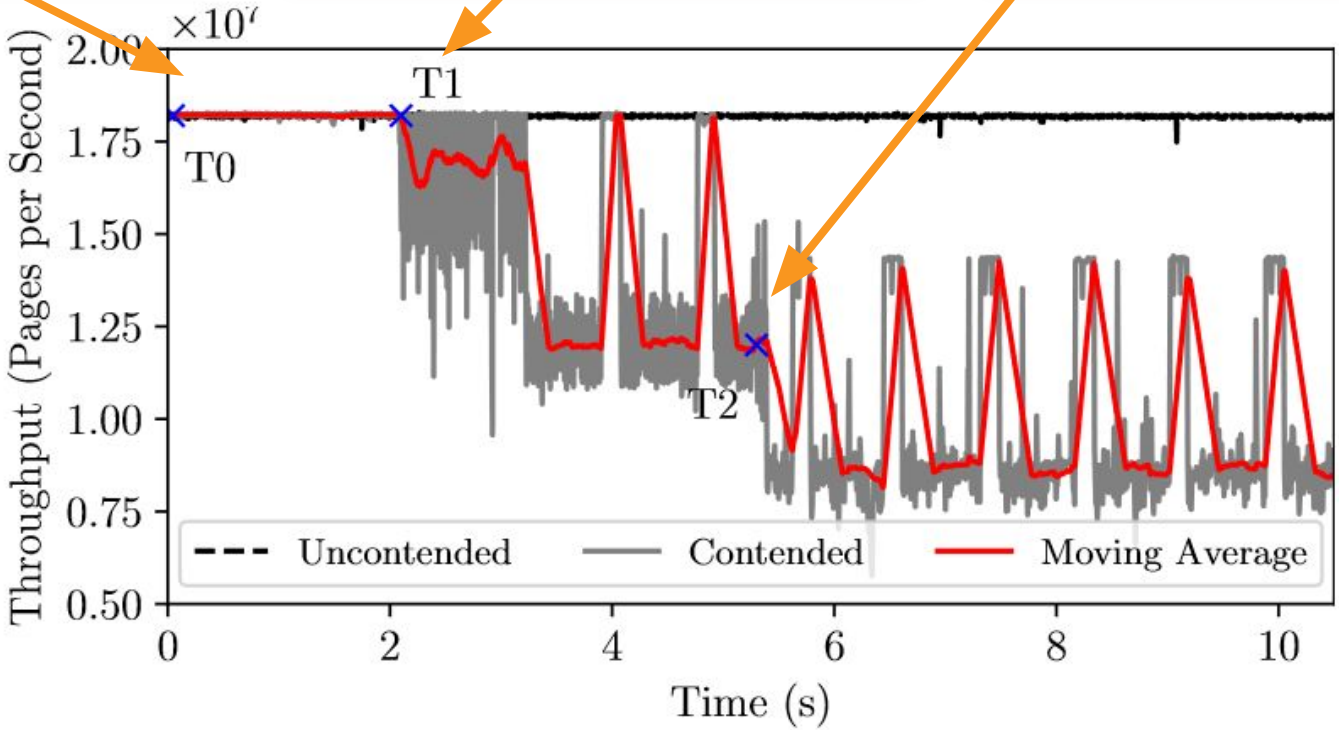
Open research question: will such techniques be needed by LDOS?

Managing Contention for the GPU

T0: user space is using the GPU

T1: page warmth classifier in the kernel starts using the GPU

T2: I/O latency classifier in the kernel starts using the GPU

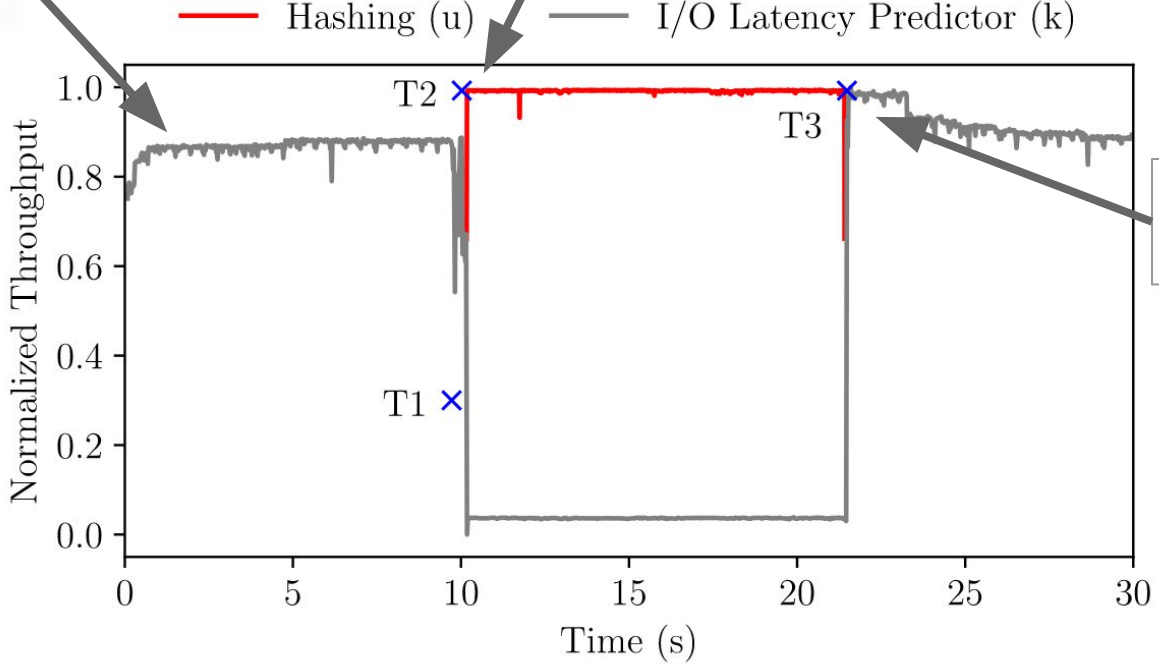


Managing Contention for the GPU

Idle GPU, kernel uses it

User application uses GPU, back off

GPU idle again, OS uses GPU



Expressing Policies using eBPF

```
1  policy cu_policy(offload_func_t dev_func, void
       *dev_args, offload_func_t cpu_func, void *
       cpu_args) {
2    static nvmlUtilization_t util;
3    if   5 ms elapsed since last check
4      // LAKE-remoted nvml API
5      nvmlGetUtilization(dev, &util)
6    // compute GPU utilization moving average
7    int exec_rate = mov_avg(util.gpu);
8    // batch size for profitability threshold
9    int batch_sz = get_batch_size(def_args)
10   if (exec_rate < exec_threshold
11       && batch_sz >= batch_threshold)
12     return dev_func(dev_args);
13   else
14     return cpu_func(dev_args); }
```

LAKE manages contention and variable profitability through custom policies (eBPF)

What if the ML model produces bad solutions?

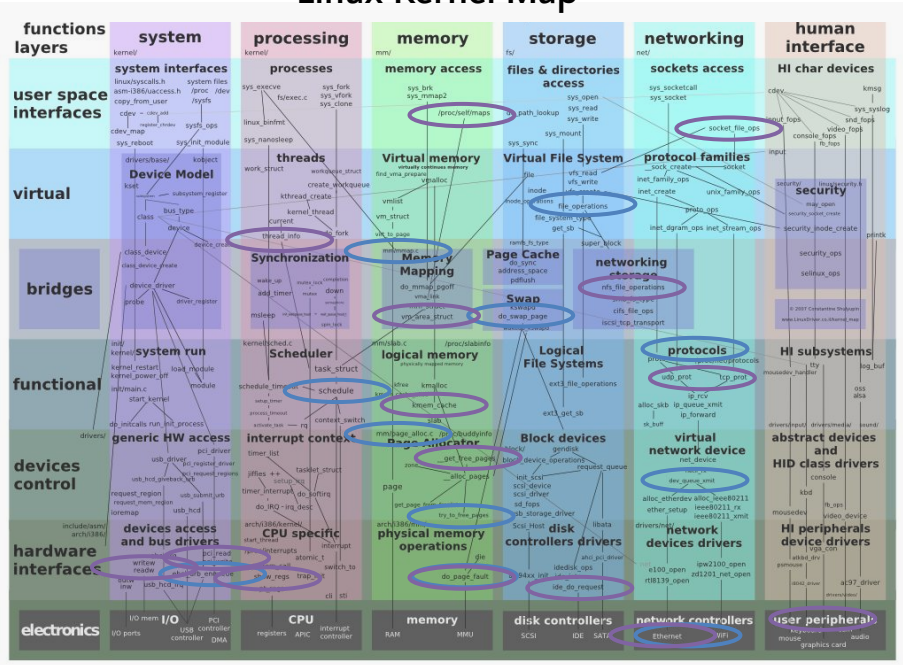
This mechanism can be augmented to support ML guardrails in LDOS

Example: poll utilization to handle contention

Example: use GPU if batch is large enough

Challenge: Poor OS Structure

Linux Kernel Map



Source: https://makelinux.github.io/kernel/map/

Issues with modern OSes:

Features and policies dispersed

Current structure a major obstacle to meeting requirements

Capturing Feature Vectors

```
// feature vector struct

struct feature_vector_t {
   int    io_size
   int    elapsed
}

//feature vectors array

struct features_t {
  int n_fvs
  struct fv_t* my_fvs
  …
}
```

```
submit_bio():
   bio->start_ts = now()
   // create new feature vector
   new_fv = allocate_fv()
   new_fv.io_size = bio->io_size

   //store feature vector
   store_fv(new_fv)
```

```
bio_end():
   fv = find_fv()
   ts_end = now()
   fv.elapsed = ts_end - bio->start_ts
   //store updated feature vector
   store_fv(my_struct, new_fv)
```

Locking disciplines:

- Careless locking can cause deadlocks
- Double acquire spinlock
- Sleep/wait while holding a spinlock
- Disabling/enabling interrupt context

LAKE's Feature Registry

LAKE's feature registry facilitates feature capture and management

submit_bio():

bio->start_ts = now()

begin_fv_capture (now())

capture_feature (
"io_size", bio->io_size)

bio_end():

elaps = now() - bio->start_ts

capture_feature (**"elapsed"**, elaps)

commit_fv_capture()

API	Description
`create_registry(name, sys, schema, window)`	Creates feature registry with capacity
`destroy_registry(name, sys)`	Destroys a feature registry
`create_model(name, sys, path)`	Create a new ML model, saved at path
`update_model(name, sys, path)`	Commit a changed model to the file system
`load_model(name, sys, path)`	Load a model from path into memory
`delete_model(name, sys, path)`	Delete a model from the file system and memory
`register_classifier(name, sys, fn, arch)`	Provide a function pointer for classifiers/inference Note: arch specifies CPU / GPU / XPU
`register_policy(name, sys, fn)`	Provide an eBPF policy for contention/batching (§4.3)
`score_features(name, sys, fvs, num)`	Run inference on a batch, return batch results
`get_features(name, sys, ts)`	Batch retrieves all feature vectors older than ts
`begin_fv_capture(name, sys, ts)`	Starts the creation of a new feature vector. Subsequent calls to capture_feature for name/subsystem will add/overwrite the current value of that feature
`capture_feature(name, sys, key, val, sz)`	Sets feature with key, val on the current vector
`capture_feature_incr(name, sys, key, incrval, sz)`	Update a feature with key by incrementing
`commit_fv_capture(name, sys, ts)`	Commits the current feature vector to the registry.
`truncate_features(name, sys, ts)`	Removes all feature vectors older than ts

Evaluation: I/O Scheduling

Setup: server with three NVMes with replicated data for redundancy

Goal: classify I/Os issued to an NVMe as fast or slow

Actions:

Fast → No-op

Slow → Re-issue to different device

How? neural network with two layers, 256 and 2 neurons, respectively.
We simulate **more complex models** by adding **+1** and **+2** layers to the NN

Testbed: 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes

Evaluation: I/O Scheduling

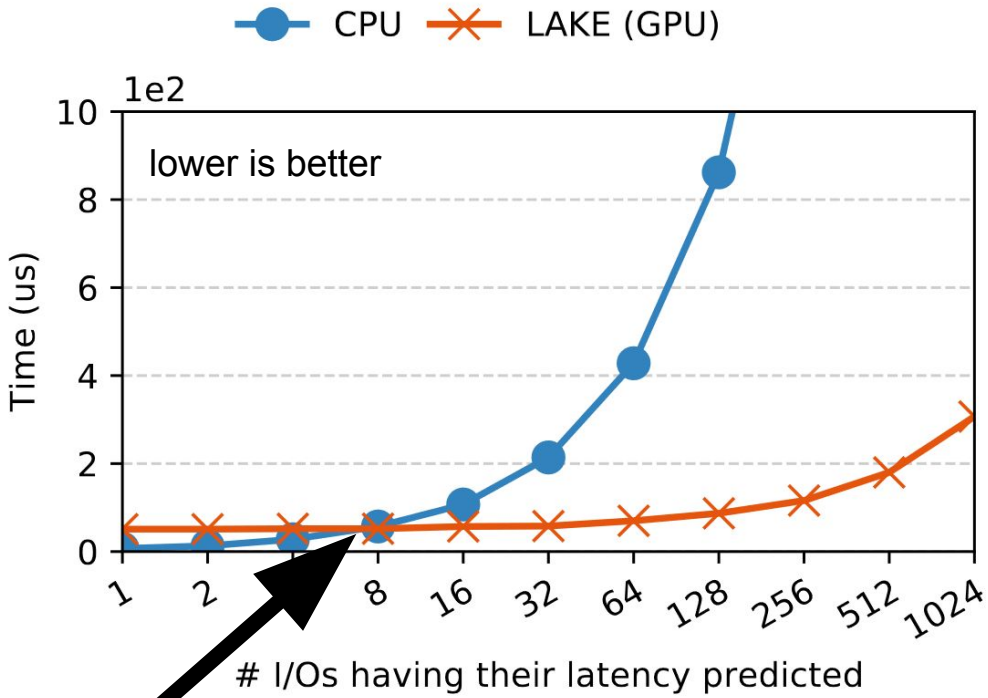
X axis is the number of I/Os we are classifying in **one batch.**

Y is the **total time to perform inference**, the lower the better

At what point does using a GPU become beneficial?

(we call this the **crossover point**)

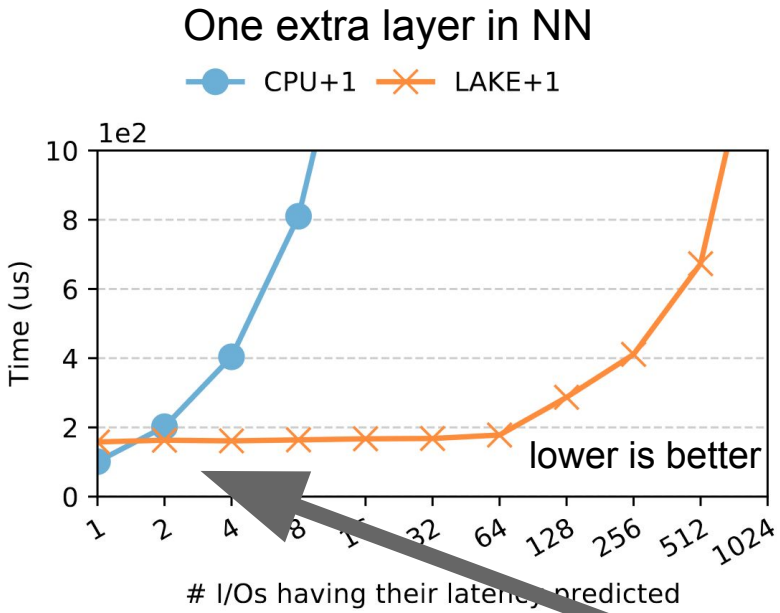
This is a small model, NN with 2 layers

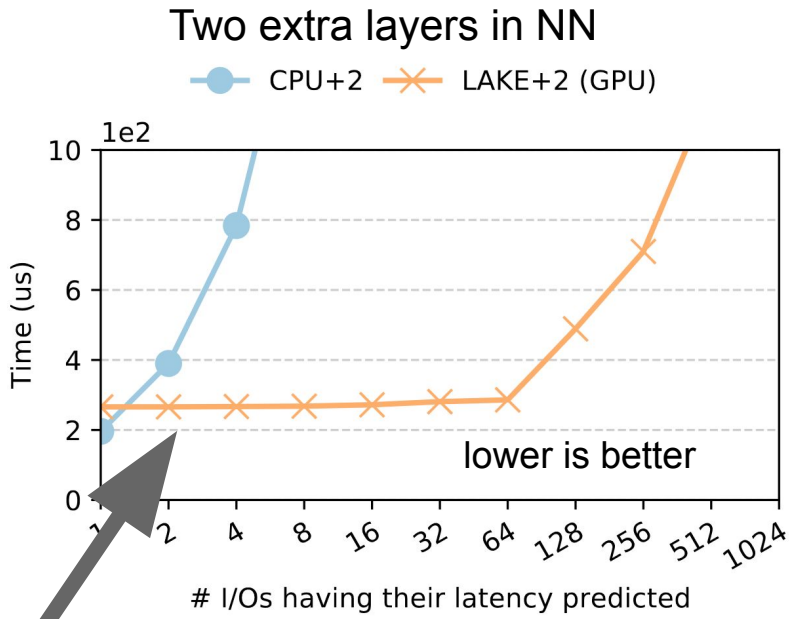


With 8 or more inputs, using a GPU is profitable

Testbed: 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes

Evaluation: I/O Scheduling





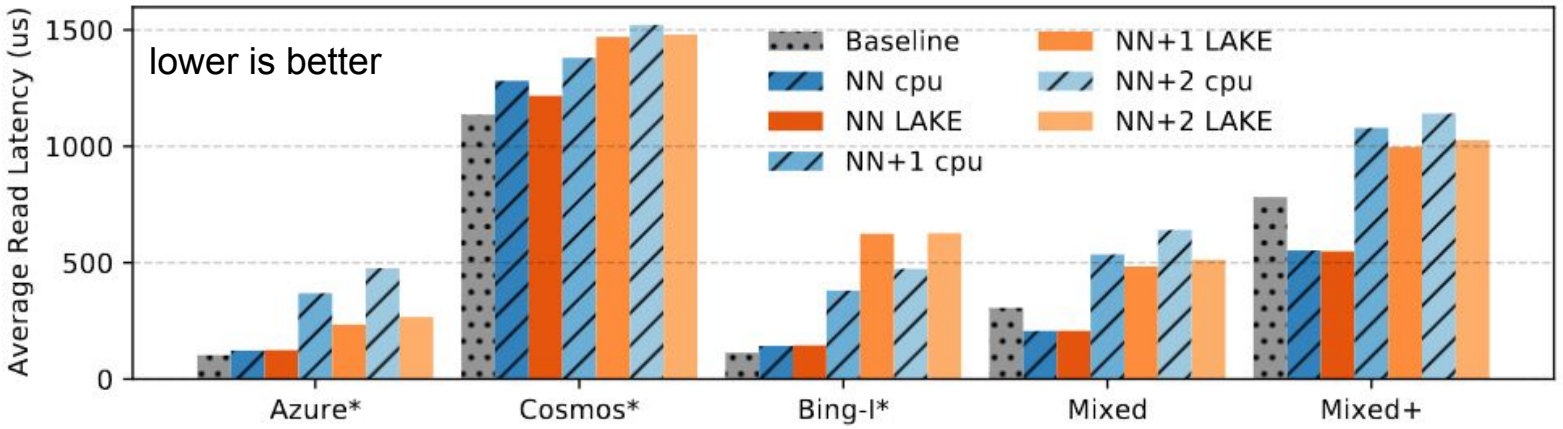
Profitability increases faster for:

- More complex models
- More inputs

For larger models, GPU is faster with just two inputs

Testbed: 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes

Evaluation: I/O Scheduling



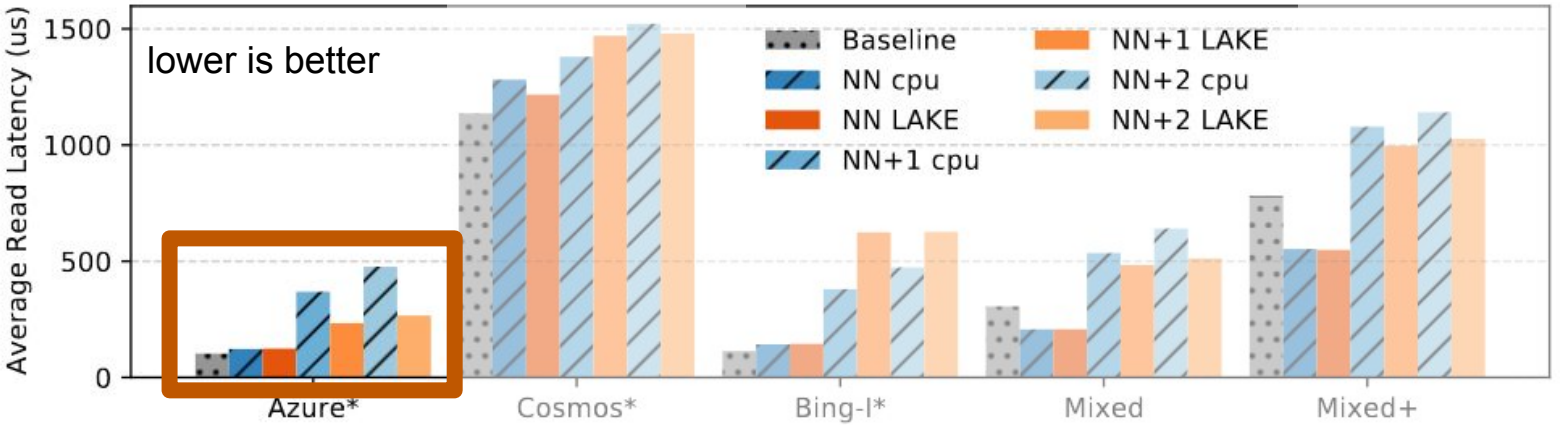
Workload properties:

Trace Name	Avg IOPS	Avg Read/Write I/O size (KB)	Min/Max Arrival Time (us)
Azure	26k	30/19	0/324
Bing-I	4.8k	73/59	0/1.8k
Cosmos	2.5k	657/609	0/1.6k

Testbed: 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes

Evaluation: I/O Scheduling

Trace Name	Avg IOPS	Avg Read/Write I/O size (KB)	Min/Max Arrival Time (us)
Azure	26k	30/19	0/324
Bing-I	4.8k	73/59	0/1.8k
Cosmos	2.5k	657/609	0/1.6k



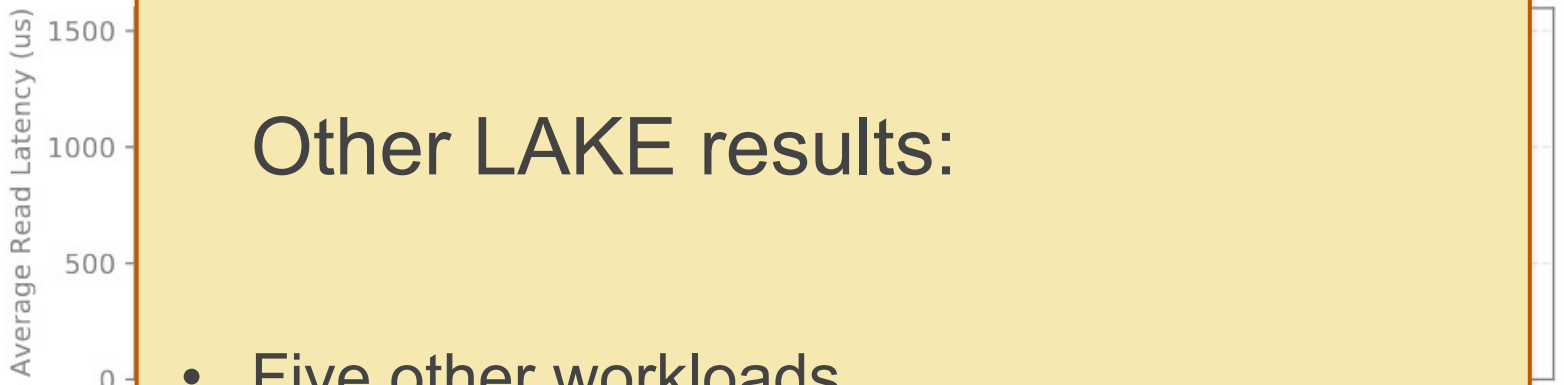
High IOPS (Azure) -> higher batch sizes, great for GPUs

Policy uses the GPU when it observes an I/O burst

Testbed: 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes

Evaluation: I/O Scheduling

Trace Name	Avg IOPS	Avg Read/Write I/O size (KB)	Min/Max Arrival Time (us)
Azure	26k	30/19	0/324
Bing-I	4.8k	73/59	0/1.8k
			0/1.6k



Other LAKE results:

- Five other workloads
- Non-ML: eCryptfs
- CPU utilization

Small I/Os

equent,
CPU

on Gold 6226R
M, two
three Samsung
980 Pro 1TB (PCIe 4.0) NVMes

LAKE: what did we learn?

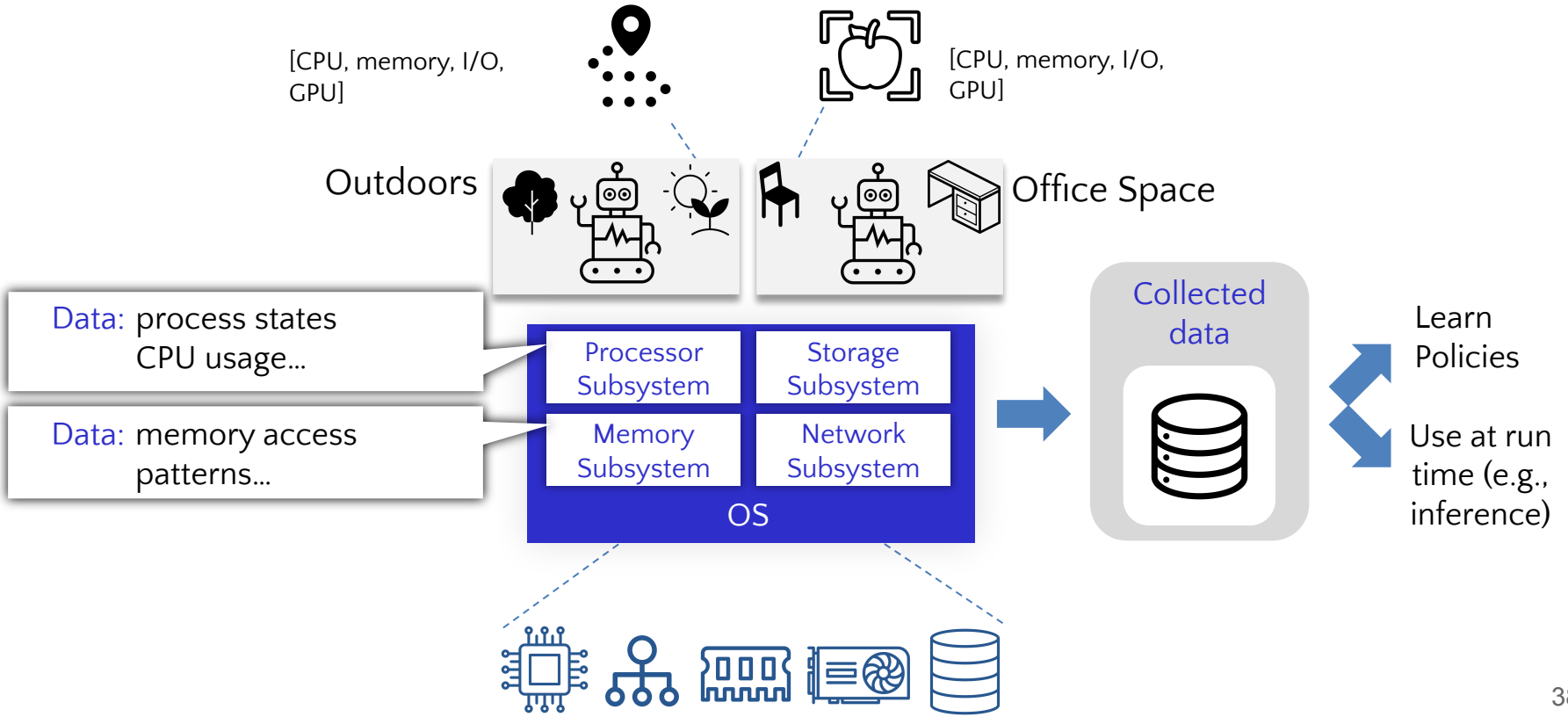
- ***Expected to learn***:
  - ML is super-applicable and works great in the kernel
- ***Actually learned*** about challenges:
  - Robustness
  - Data is hard to come by
  - Choosing learning techniques
  - OS+ML may need *different* acceleration
  - Explainability/Misbehavior/pathologies
  - Development cycle mismatches
  - OS structure

Robustness

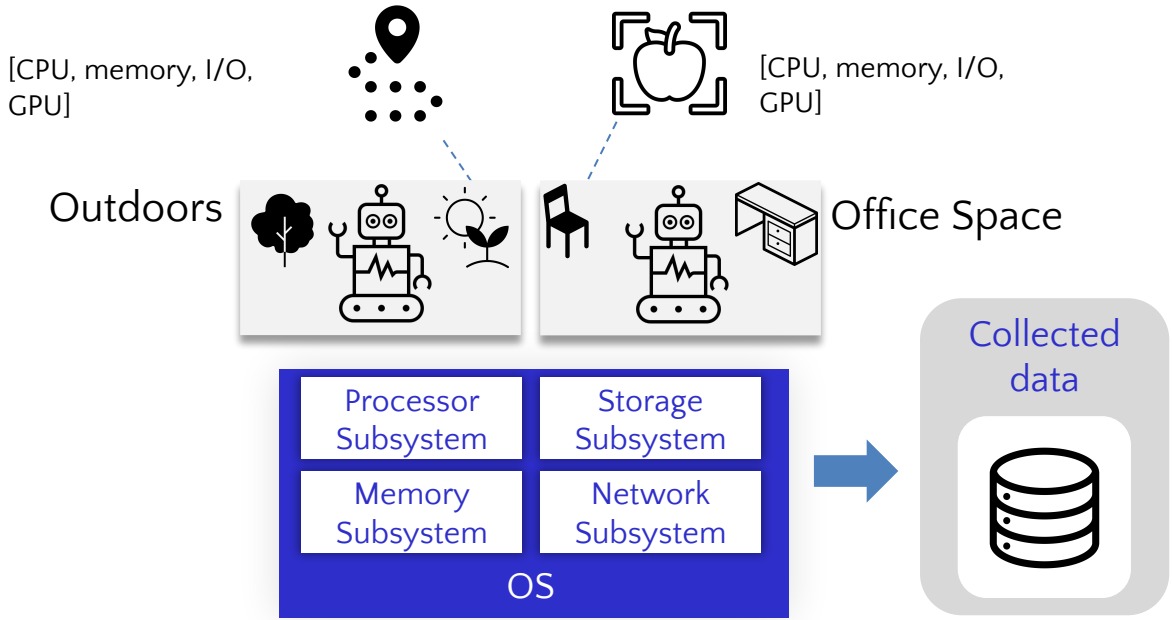
	API?	Contention Control?	Complex models?	End-to-end evaluation?	Repro-ducible?
KML [1]	✓	*	X	✓	...
SmartOS [2]	X	X	X	✓	...
LinnOS [3]	X	X	X	✓	...
Lynx [4]	✓	✓	✓	✓	...
Kleio [5]	X	X	*	X	...
LAKE	✓	✓	✓	✓	...

- Results difficult to reproduce
- Results very sensitive to env. / exp. design
- ***Not a criticism of previous work:***
- ***These challenges are fundamental***

Data Challenge

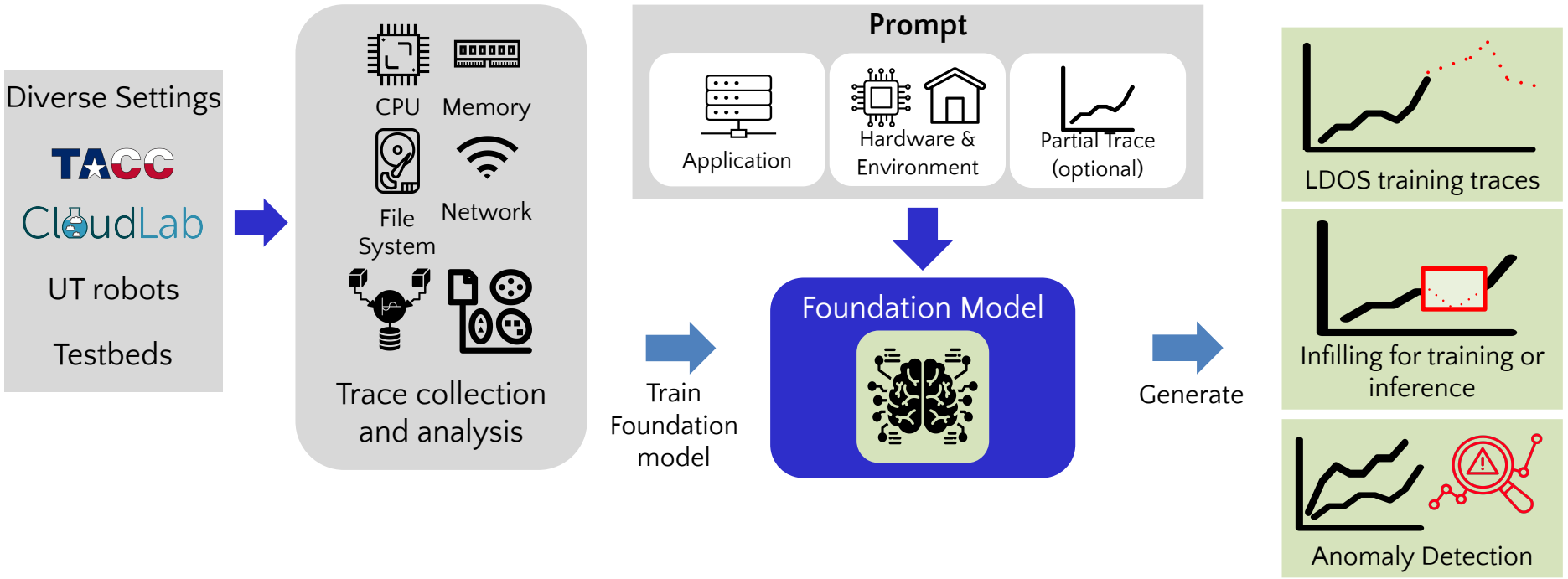


Data Challenge



Data collection overhead	Privacy and regulatory constraints	Unseen applications and environments

LDOS Solution: Foundation Model for OS Data



Learning Techniques for ML in OS

Supervised learning techniques

- Require ground truth
- Require lots of data
- Hard to improve over existing heuristics
- Existing sources of ground truth are heuristics
- Ground truth data improving over heuristics: not scalable

Unsupervised techniques

- Can improve over heuristics (e.g. RL)
- Computationally expensive
- Can be slow to converge on best policy

ML, OS, Hardware Acceleration

LAKE design adapts to current hardware

- GPUs/TPUs designed for user-level, OS-independence
- GPUs/TPUs designed for high-throughput batch compute
- Kernels care a lot about latency
- In-kernel ML still dominated by parallel compute (e.g. GEMM)

Conjecture: Hardware will evolve if ML+OS benefits

Consequence: near term challenge

- LDOS will likely focus on CPU-amenable ML
- Evaluating benefits will be tough for many ML techniques

Addressing Challenges with LDOS

Robustness	○	**Verified Learning**
Data is hard to come by	○	**Foundation Model for OS**
Explainability/Misbehavior	○	**Guardrails / Adaption**
Development cycle mismatches	○	**KernMLOps**
OS structure	○	**Clean-slate Design**
Choosing learning techniques	○	**<>**
OS+ML: *different* acceleration	○	**<>**

Conclusion

LDOS

- Clean-slate OS
- ML-driven resource management policy architecture
- Cross-cutting : generative AI, ML, formal methods, and systems

Many exciting research directions!

We welcome collaborators!
https://ldos.utexas.edu

Questions?