

Unikernels for Serverless Computing

Performance, Challenges, and Research Directions

SUBMITTED BY
Aygul Nuryyeva

INSTRUCTOR
Nurmyrat Allayarov

UNIVERSITY
International University for the Humanities and Development

DATE
November 2026

Abstract

Serverless computing, or Function-as-a-Service (FaaS), lets developers deploy short-lived functions without managing servers, but it exposes a well-known performance problem: the *cold start*, the delay incurred when a platform must create a fresh execution environment before a function can run. This paper reviews unikernels—single-purpose, single-address-space virtual machine images that link an application with only the operating-system libraries it needs—as an architectural response to that problem. Drawing on peer-reviewed systems research and primary project documentation, the paper explains how serverless cold starts arise, how unikernel architectures differ from virtual machines, containers and microVMs, and what the published evidence reports. Reported results include microVM start-up of about 125 ms with under 5 MiB of memory overhead [1], unikernel boot in 2.3 ms on a modified Xen toolstack [2], and Unikraft images of roughly 1 MB that need under 10 MB of RAM and boot in 3–40 ms including the monitor [3]. The paper also argues that unikernels address only part of the cold-start path and must be combined with snapshotting and runtime optimisation. Four open problems are analysed: toolchain maturity, application compatibility, debugging, and fine-grained latency attribution. The paper closes with methodological recommendations for standardised benchmarking and reproducible, open research artifacts. The study is a literature-based analytical review; it reports no new measurements, and all quantitative figures are attributed to their original sources.

Keywords: unikernel; library operating system; serverless computing; Function-as-a-Service; cold start; microVM; Firecracker; Unikraft; virtualisation; reproducibility.

Table of Contents

1. Introduction 5

1.1 Background and motivation 5

1.2 Problem statement 6

1.3 Research questions 6

1.4 Objectives and contributions 6

1.5 Methodology and scope 6

1.6 Structure of the paper 7

2. Serverless Computing and the Cold-Start Problem 7

2.1 The Function-as-a-Service execution model 7

2.2 Anatomy of a cold start 7

2.3 Empirical evidence on cold starts and workloads 8

2.4 Existing mitigation strategies 9

2.5 Isolation technologies in brief 9

3. Unikernel Architecture 10

3.1 Origins and design principles 10

3.2 Architectural comparison 10

3.3 Build and boot pipeline 10

3.4 Survey of major systems 11

3.5 Monitors and hosting models 12

3.6 Linux-based and binary-compatible approaches 12

4. Published Performance Evidence 12

4.1 Boot time 13

4.2 Memory footprint, image size and density 13

4.3 Throughput and application performance 14

4.4 Interpreting the evidence: threats to comparability 14

5. Unikernels in Serverless Platforms 14

5.1 Reference architecture 14

5.2 Combining unikernels with snapshots 15

5.3 Development accessibility 15

5.4 Artifact transparency and reproducibility 16

5.5 Comparative assessment (author's synthesis) 16

6. Limitations and Research Gaps 16

6.1 Stability and production readiness 17

6.2 Application compatibility 17

6.3 Debugging complexity 17

6.4 Observability and performance attribution . . . 17
6.5 Isolation within a single address space . . . 17
6.6 State sharing and density . . . 18

7. Methodological Recommendations . . . 18

7.1 Standardised benchmarks and workloads . . . 18
7.2 Metric definitions and statistical practice . . . 18
7.3 A latency-attribution protocol . . . 18
7.4 Reproducibility checklist . . . 19
7.5 Clear architectural visualisation . . . 19

8. Discussion . . . 19

8.1 Threats to validity of this review . . . 19

9. Conclusion . . . 20

References . . . 21

Appendix A. Glossary . . . 23

Appendix B. Notes on Sources and Illustrations . . . 23

List of Figures and Tables

Figure 1. Conceptual layered virtualisation stack . . . 5
Table 1. Source selection and evidence rules . . . 6
Figure 2. Request path in a FaaS platform . . . 7
Figure 3. Stages of a cold start and the techniques that target them . . . 8
Table 2. Cold-start mitigation strategies . . . 9
Table 3. Comparison of isolation technologies . . . 9
Figure 4. Layered view of four isolation architectures . . . 10
Figure 5. Unikernel build and deployment pipeline . . . 11
Table 4. Selected unikernel and unikernel-like systems . . . 11
Figure 6. Milestones in library OS, virtualisation and serverless . . . 12
Figure 7. Reported boot and start-up times (log scale) . . . 13
Table 5. Quantitative results as stated by original authors . . . 13
Figure 8. Reference architecture of a unikernel-based serverless platform . . . 15
Table 6. Qualitative comparison matrix (author’s synthesis) . . . 16
Figure 9. From limitations to research directions . . . 16
Table 7. Proposed latency-attribution timeline . . . 18
Table 8. Reproducibility checklist . . . 19
Table 9. Glossary of terms . . . 23

1. Introduction

1.1 Background and motivation

Cloud computing has moved steadily from renting machines toward renting *functions*. Since Amazon Web Services introduced Lambda in 2014 [4], the serverless model has become a mainstream way to build event-driven applications, web back-ends, data-processing pipelines and machine-learning inference services. In this model the developer supplies code and the provider takes responsibility for provisioning, scaling and isolating the execution environment [5], [6]. Industry surveys of practice describe why, when and how organisations adopt the model [7], and the economic appeal is simple: resources are consumed, and billed, only while code actually runs.

That elasticity has a cost. Because idle environments are reclaimed to save resources, a request that arrives when no suitable environment exists must wait while the platform creates one. This delay—the *cold start*—adds latency that is irrelevant to long-running batch jobs but highly visible to interactive, latency-sensitive workloads. Providers therefore sit between two pressures: keeping environments alive wastes memory, while tearing them down quickly shifts the penalty onto users.

One line of research attacks the problem at its root by shrinking the environment itself. **Unikernels** compile an application together with only the operating-system functionality it uses into a single, specialised virtual machine image [8], [9]. Because such an image contains no general-purpose kernel, no shell and no unused drivers, it can be very small and can start very quickly. Recent systems such as Unikraft report images of roughly one megabyte that boot in a few milliseconds [3], which is attractive for a platform that must create thousands of short-lived sandboxes.



Figure 1. *Conceptual illustration of a layered virtualisation stack: a cloud-hosted workload (cube) above a system-software layer (gears), running on physical server hardware. Unikernels shrink the middle layers to the minimum an application needs. Illustration supplied by the author.*

1.2 Problem statement

Despite promising headline numbers, unikernels have not displaced containers or Linux-based microVMs in production serverless platforms. Firecracker, for example, has been used in production for AWS Lambda and AWS Fargate since 2018 [1], [4], whereas unikernel toolchains remain primarily research and niche-deployment technologies. The question this paper examines is therefore twofold: *how much* of the serverless cold-start problem can unikernels really solve, and *what must be resolved* before they can be adopted at scale?

1.3 Research questions

- **RQ1.** Which stages of a serverless cold start are affected by the choice of isolation technology, and which are not?
- **RQ2.** How do unikernel architectures differ from virtual machines, containers and microVMs in structure, isolation and start-up behaviour?
- **RQ3.** What quantitative evidence do peer-reviewed studies provide for unikernel start-up time, memory footprint and throughput, and how comparable is it?
- **RQ4.** What technical and methodological limitations currently prevent broad adoption, and which research directions address them?

1.4 Objectives and contributions

The paper has five objectives: (i) to explain the anatomy of a serverless cold start; (ii) to describe unikernel architecture and its main implementations; (iii) to collect and carefully qualify the published performance evidence; (iv) to analyse limitations in stability, compatibility, debugging and observability; and (v) to recommend practices for transparent, reproducible benchmarking. The main analytical contribution is a decomposition of cold-start latency (Section 2.2) that shows why a faster operating-system boot, taken alone, cannot remove the whole penalty.

1.5 Methodology and scope

This is a literature-based analytical review. Sources were selected from peer-reviewed venues in operating systems, virtualisation and cloud computing (SOSP, ASPLOS, EuroSys, NSDI, USENIX ATC, VEE, SoCC and HotCloud), supplemented by primary project documentation and artifact repositories where a project's own site is the authoritative description. Quantitative values are quoted only as stated by their original authors and are never recomputed or merged across studies. Where the paper offers its own synthesis—such as the qualitative comparison in Section 5.5—it is labelled as such. No new experiments were conducted.

Table 1. Source selection and evidence rules used in this review.

Criterion	Applied rule
Source type	Peer-reviewed conference or journal paper; or official project documentation for tooling descriptions.
Relevance	Addresses unikernels, library operating systems, lightweight virtualisation, or serverless cold-start behaviour.
Numbers	Quoted verbatim from the original source with its context (hardware, metric definition) noted; no cross-paper arithmetic.
Synthesis	Author's own judgements are marked as such and are not presented as measured results.

1.6 Structure of the paper

Section 2 introduces serverless computing and the cold-start problem. Section 3 describes unikernel architecture and surveys the main systems. Section 4 reviews published performance evidence. Section 5 examines how unikernels fit into serverless platforms, including development accessibility and artifact transparency. Section 6 analyses limitations and research gaps. Section 7 gives methodological recommendations. Section 8 discusses the research questions, and Section 9 concludes.

2. Serverless Computing and the Cold-Start Problem

2.1 The Function-as-a-Service execution model

In FaaS, an application is decomposed into functions that are triggered by events such as HTTP requests, queue messages or timers. A platform component receives each event, decides where to run the function, and either reuses an existing execution environment (a *warm* start) or creates a new one (a *cold* start). The environment is typically a sandbox that provides isolation between mutually distrusting tenants, because a single physical server hosts functions from many customers [5], [1].

Isolation is the central design constraint. Hardware virtualisation gives strong isolation but, in its traditional form, boots a full guest operating system. Containers start quickly but share the host kernel, which enlarges the attack surface. The research reviewed here can be read as a search for sandboxes that are both strongly isolated and cheap to create [1], [2], [10].

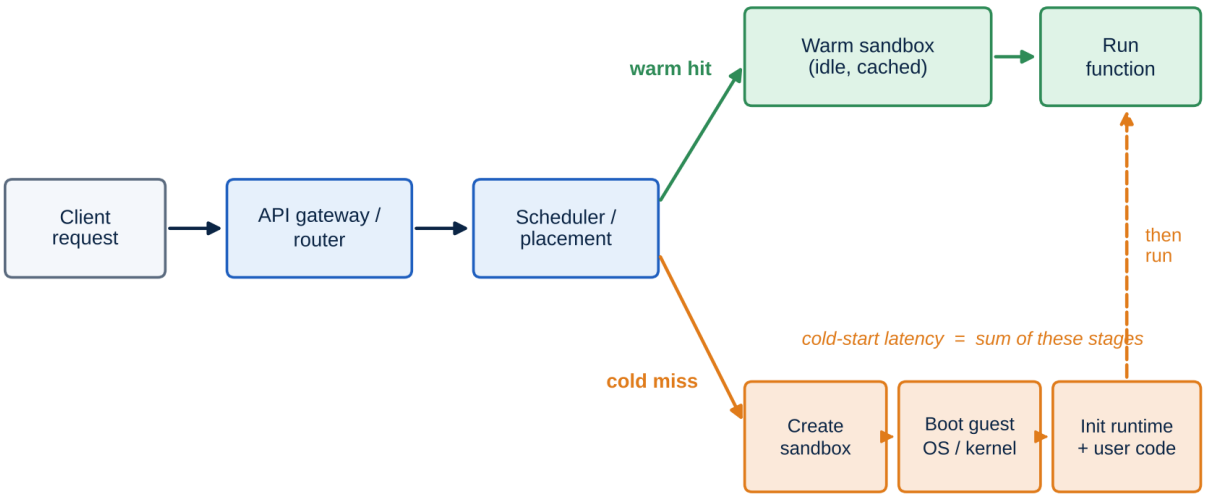


Figure 2. *Request path in a FaaS platform. A warm hit reuses an idle sandbox; a cold miss must create a sandbox, boot a guest system and initialise the runtime before the function can run.*

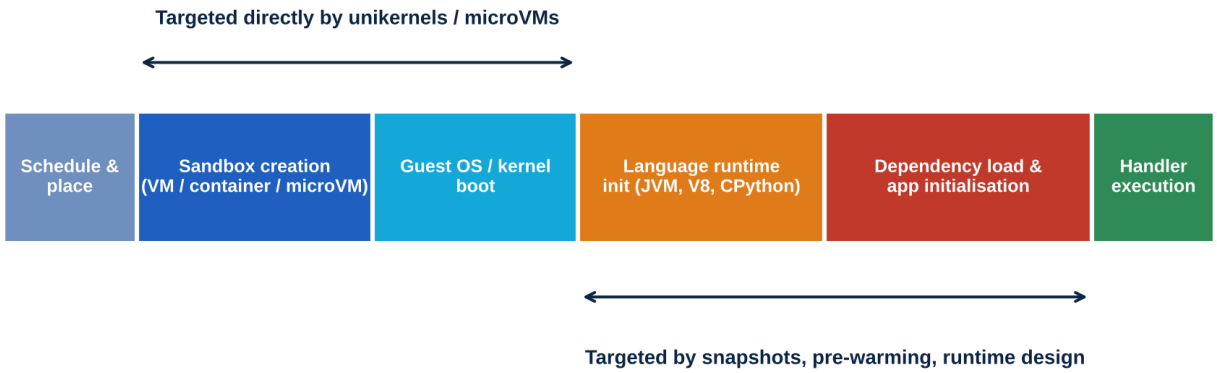
2.2 Anatomy of a cold start

A cold start is not a single delay but a pipeline of stages. Following the decomposition used implicitly in measurement and snapshotting studies [11], [12], [13], the end-to-end cold-start latency can be written as:

$$T_{cold} = T_{sched} + T_{sandbox} + T_{os} + T_{runtime} + T_{init}$$

where T_{sched} is request routing and placement, $T_{sandbox}$ is the creation of the container or virtual machine, T_{os} is guest operating-system boot, $T_{runtime}$ is language-runtime start-up (for example a JVM, the V8 engine or a Python interpreter), and T_{init} is loading dependencies and running user initialisation code. Handler execution

time is excluded because it is paid on warm starts too. Figure 3 shows the stages schematically.



Schematic only: segment widths are illustrative and are not measurements.

Figure 3. *Stages of a cold start and the techniques that target them. Segment widths are illustrative, not measured.*

This decomposition has a direct consequence for unikernels. A unikernel mainly reduces $T_{sandbox}$ and T_{os}: it removes general-purpose kernel boot and unused devices and drivers. It does not by itself shorten $T_{runtime}$ or T_{init}, which depend on the language runtime and on application code. As a purely *hypothetical* illustration (the numbers are not measurements), if guest boot were 150 ms and runtime plus application initialisation were 500 ms, a perfect, zero-cost operating-system boot would still leave 500 ms of cold-start latency. This is Amdahl's-law reasoning applied to start-up, and it explains why recent work combines lightweight sandboxes with snapshots [14], [15], [13].

2.3 Empirical evidence on cold starts and workloads

Measurement studies have shown that cold-start behaviour varies considerably across commercial platforms and configurations. Wang et al. instrumented several major providers and examined how instance creation, reuse and placement affect latency [11]. Shahrad et al. characterised the production workload of a large cloud provider's function service and found that invocation patterns are highly skewed: many functions are invoked infrequently, which makes them likely to encounter cold starts under simple keep-alive policies. They proposed a policy that adapts keep-alive and pre-warming windows to each application's observed invocation pattern [16].

These findings frame the value of faster sandboxes. If a platform cannot predict when an infrequently invoked function will be called, the cheapest way to avoid wasted memory without hurting latency is to make creating a new sandbox fast enough that keeping one idle is unnecessary.

2.4 Existing mitigation strategies

Table 2. Cold-start mitigation strategies and their trade-offs.

Strategy	Mechanism	Representative work	Main limitation
Keep-alive and pre-warming	Retain idle sandboxes; predict arrivals to start them early.	Adaptive policies [16]; caching-based keep-alive [17]	Consumes memory; prediction fails for rare or bursty functions.
Optimised containers	Lean container set-up, package caching, pools of pre-created resources.	SOCK [12]; SAND [18]; pause-container pools [19]	Shared host kernel; start-up still bounded by container and runtime set-up.
MicroVMs	Minimal device model and VMM; strong hardware isolation.	Firecracker [1]	Still boots a Linux guest kernel; ~125 ms class start-up.
Snapshot / restore	Resume from saved memory image of an initialised sandbox.	Catalyzer [15]; SEUSS [14]; vHive snapshots [13]	Snapshot storage and page-fault cost on restore; state-sharing concerns.
Unikernels	Remove the general-purpose OS; boot a specialised single-purpose image.	LightVM [2]; Unikraft [3]	Porting effort, tooling maturity, debugging (Section 6).

2.5 Isolation technologies in brief

Virtual machines. A hypervisor multiplexes hardware among guest operating systems, following the formal virtualisation requirements first set out by Popek and Goldberg [20]. Xen [21] and the Linux Kernel-based Virtual Machine (KVM) [22] are widely deployed hypervisors. The isolation boundary is narrow and well understood, but a conventional guest boots a complete operating system.

Containers. Containers use operating-system features such as namespaces and control groups to isolate processes while sharing one kernel [23]. They start in a fraction of a second and pack densely, but a kernel vulnerability can affect every tenant. Sandboxing layers such as gVisor interpose a user-space kernel to narrow the host-kernel interface, at the price of measurable overhead [10].

MicroVMs. Firecracker is a virtual machine monitor written in Rust that runs on KVM and exposes only a handful of emulated devices; its documentation lists a minimal set including virtio network, block and vsock devices, a serial console and a minimal keyboard controller [24], [4]. It is paired with a companion “jailer” process that adds Linux user-space isolation barriers [4]. Its authors report starting user-space code in as little as 125 ms and a memory overhead below 5 MiB per microVM [1], [24].

Table 3. Qualitative comparison of isolation technologies. Start-up figures are those reported in the cited sources.

Property	Virtual machine	Container	MicroVM	Unikernel
Isolation boundary	Hypervisor	Shared host kernel	Hypervisor (minimal VMM)	Hypervisor (thin monitor)
Guest kernel	Full general-purpose OS	None (host kernel)	Trimmed Linux kernel	Specialised LibOS in image
Typical image content	Complete OS + app	App + userland	Kernel + rootfs + app	App + needed libraries only
Attack surface	Large guest, narrow host interface	Large host-kernel interface	Small device model	Small device model, tiny guest
Start-up class (reported)	Seconds	Sub-second	≤ 125 ms [1]	Milliseconds [2], [3]

3. Unikernel Architecture

3.1 Origins and design principles

The intellectual roots of unikernels lie in *library operating systems*. The exokernel architecture argued that an operating system should securely multiplex hardware while leaving abstractions to application-level libraries [25]. MirageOS revived the idea for the cloud: applications written in OCaml are compiled together with the libraries they need into a bootable virtual machine image that runs directly on a hypervisor [8], [9], [26]. The resulting artefact was named a *unikernel*.

Three properties define the approach. First, a unikernel is **single-purpose**: one image runs one application. Second, it uses a **single address space**, so there is no separation between kernel and user mode inside the image and system calls become ordinary function calls. Third, it is **specialised at build time**: the build system links only the operating-system components the application actually uses. Together these yield small images, short boot paths and a reduced attack surface, because code that is never linked cannot be exploited [8], [3].

3.2 Architectural comparison

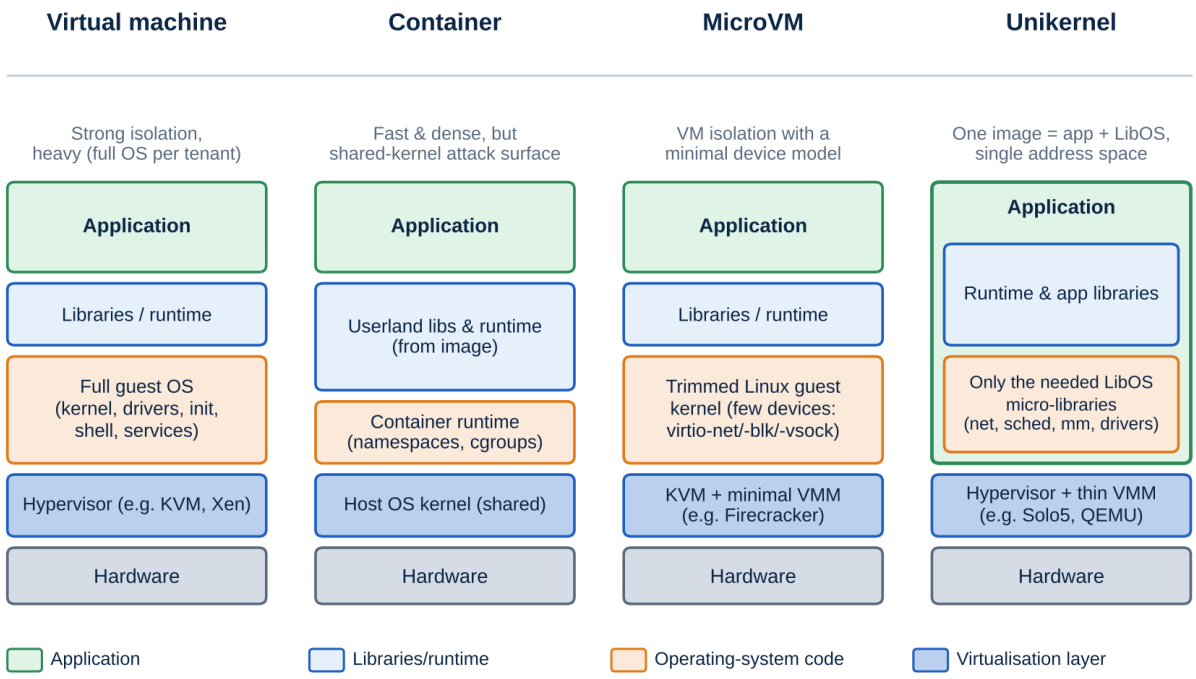


Figure 4. *Layered view of four isolation architectures. In a unikernel the application and the needed operating-system micro-libraries form one image running directly on a thin virtualisation layer.*

3.3 Build and boot pipeline

Building a unikernel differs from building a container image. In Unikraft, for instance, the operating system is decomposed into stand-alone micro-libraries—memory allocators, schedulers, network stacks and early boot code—that a developer selects through a configuration system; link-time optimisation and dead-code elimination then trim the result [3]. The output is a single bootable image targeting a chosen virtual machine monitor (Figure 5).

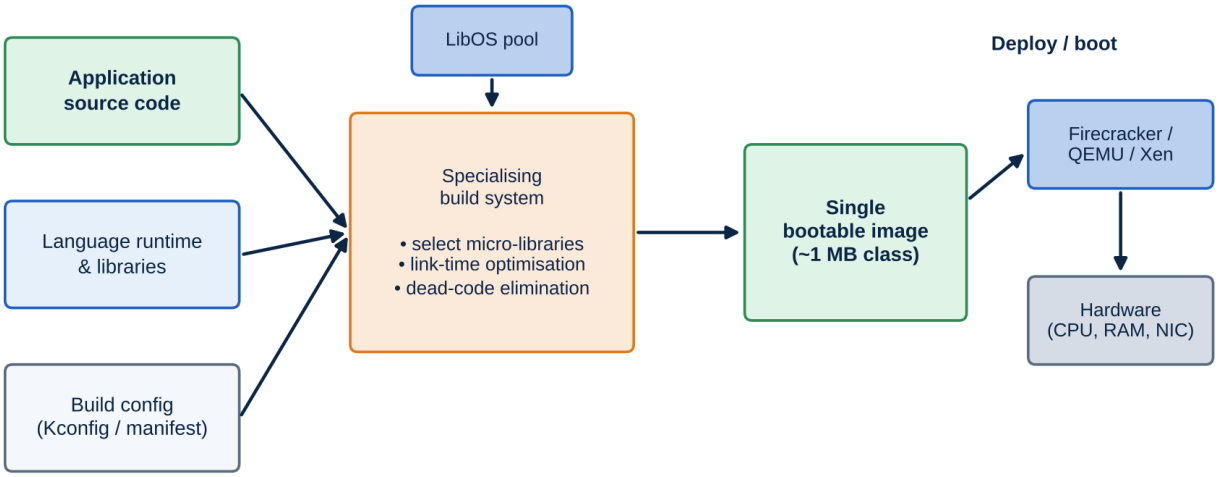


Figure 5. Typical unikernel build and deployment pipeline: application, runtime and configuration are combined with selected LibOS micro-libraries into one image that boots on a virtual machine monitor.

At boot, the monitor loads the image into guest memory and transfers control to its entry point. The image sets up memory management and interrupt handling, initialises the few virtual devices it needs (typically paravirtual network and block devices), and starts the application. There is no init system, no service manager and no shell. The Unikraft authors separate the time spent in the monitor from the time spent in the guest and report that the guest boots in around one millisecond on top of the monitor time [3]; the total therefore depends strongly on which monitor is used.

3.4 Survey of major systems

Table 4. Selected unikernel and unikernel-like systems.

System	Year	Approach	Notes on compatibility and use
MirageOS [8], [26]	2013	Type-safe OCaml library OS; targets Xen and other back-ends.	Application must be written for the framework (OCaml).
OSv [27]	2014	General-purpose unikernel OS for a single application per VM.	Designed to run unmodified Linux applications, including JVM workloads.
IncludeOS [28]	2015	Minimal C++ unikernel for cloud services.	C++ applications built against the library.
Rumprun / rump kernels [29]	2012–	Reuses NetBSD drivers via the “anykernel” design.	Supports many POSIX applications; basis for several later systems.
Solo5 / ukvm [30], [31]	2016	Minimal sandboxed execution interface and monitor for unikernels.	Provides a small host interface rather than a full OS.
LightVM (unikernels + Tinyx) [2]	2017	Redesigned Xen toolstack plus small VMs.	Focus on fast control plane as well as small guests.
HermiTux [32]	2019	Binary-compatible unikernel.	Runs unmodified Linux binaries without recompilation.
Lupine Linux [33], [34]	2020	Specialised Linux kernel configuration plus Kernel Mode Linux.	“Unikernel-like” yet still Linux; broad application support.
Unikraft [3], [35]	2021	Modular micro-library OS with a configuration system.	Targets existing applications; multiple monitors supported.
Nanos [36]	—	Unikernel kernel with tooling (OPS) for running common-language apps.	Single-process model; supports popular runtimes such as Node.js.
UKL [37]	2023	Links an application into the Linux kernel image.	Keeps Linux ecosystem; shortens kernel/user path.



Figure 6. *Milestones in library operating systems, lightweight virtualisation and serverless platforms discussed in this paper (years are publication or release years of the cited works).*

3.5 Monitors and hosting models

Where a unikernel runs matters as much as what it contains. Solo5 introduced a minimal interface between a unikernel and its host, together with a purpose-built monitor, so that the host-facing surface could be kept tiny [30]. The “unikernels as processes” work later showed that a unikernel can run as an ordinary host process confined by a very small set of permitted system calls, trading some of the hardware-isolation boundary for efficiency [38]. In production-style serverless settings the most prominent host is a microVM monitor: Lupine and Unikraft both report experiments on Firecracker, and the Lupine authors explicitly compare against a Firecracker microVM baseline [33], [3].

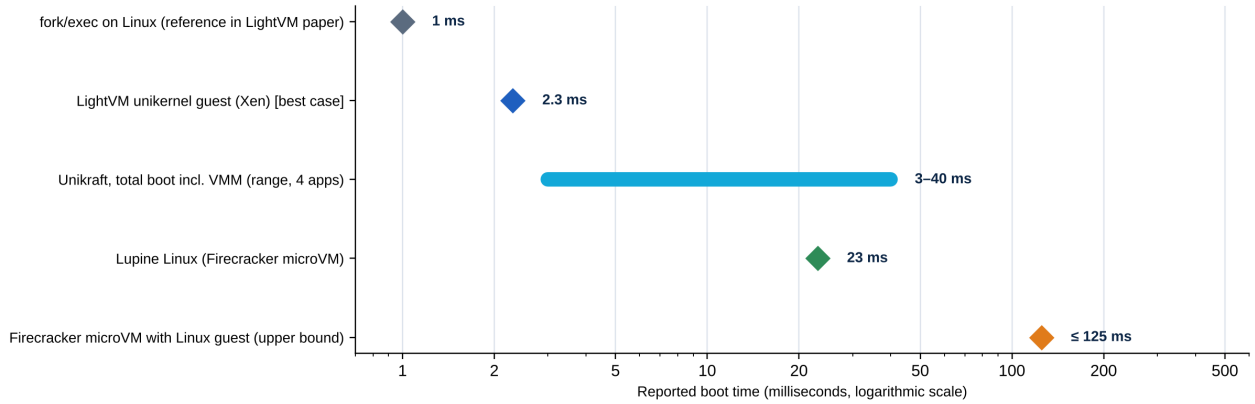
3.6 Linux-based and binary-compatible approaches

A recurring objection to classic unikernels is that they require applications to be rewritten or heavily ported. Three responses have emerged. *Binary compatibility* (HermiTux) runs existing Linux executables without recompilation [32]. *Linux specialisation* (Lupine) starts from the Linux kernel, strips it through configuration and, via Kernel Mode Linux, removes the privilege transition so that system calls become cheaper [33]. *Kernel integration* (UKL) links an application directly into the Linux kernel [37]. The Lupine authors’ own conclusion is instructive: much of the unikernel benefit comes from specialisation, which can be obtained without abandoning Linux [33], [34].

4. Published Performance Evidence

This section collects the quantitative claims made by the original authors. Because the studies use different hardware, monitors, applications and definitions of “boot time”, the figures are presented side by side for orientation, not as a ranking.

4.1 Boot time



Values are as reported in the cited papers, measured on different hardware, workloads and boot-time definitions. They illustrate orders of magnitude and are not a controlled head-to-head comparison.

Figure 7. *Boot or start-up times as reported by the original authors on a logarithmic scale. Different hardware, workloads and boot-time definitions apply; see Table 5.*

The spread is large. LightVM reports booting a unikernel guest in 2.3 ms, which it compares with 1 ms for fork/exec on Linux and describes as two orders of magnitude faster than Docker [2]. Unikraft reports a total boot time of 3–40 ms across its evaluated applications, including the time spent in the virtual machine monitor, with the guest itself needing roughly a millisecond [3]. Lupine Linux, a specialised Linux, reports 23 ms [33]. Firecracker's published figure is a bound rather than a typical value: user-space code can start in as little as 125 ms [1], [4]. The ordering is plausible given how much software each system boots, but the experiments are not identical, so the ordering should not be over-interpreted.

4.2 Memory footprint, image size and density

Table 5. Quantitative results as stated by the original authors (context differs between studies).

Source	Boot / start-up	Memory and size	Other reported results
Firecracker [1], [24]	User-space code starts in as little as 125 ms; up to 150 microVMs created per second per host.	Memory overhead below 5 MiB per microVM.	Used in production for Lambda and Fargate since 2018 [4].
LightVM [2]	Unikernel guest boots in 2.3 ms; fork/exec on Linux ≈ 1 ms.	Memory and CPU use comparable to processes.	Up to 8000 VMs on one host; migration and suspend/resume in tens of ms.
Unikraft [3]	Guest ≈ 1 ms on top of monitor time; total 3–40 ms.	Images around 1 MB; under 10 MB of RAM.	1.7×–2.7× performance improvement over Linux guests (nginx, SQLite, Redis).
Lupine Linux [33]	Boot 23 ms.	Image 4 MB; memory footprint 21 MB.	Up to 33% higher throughput than a Firecracker microVM baseline.

Memory density matters for serverless because a host must hold many concurrent sandboxes. A footprint of a few to a few tens of megabytes per sandbox, as reported above, makes it feasible to hold thousands of instances per server. LightVM's evaluation reaches up to 8000 guests on a single host [2].

4.3 Throughput and application performance

Specialisation can improve steady-state performance as well as start-up. Unikraft reports a 1.7×–2.7× improvement over Linux guests for nginx, SQLite and Redis [3], and Lupine reports up to 33% higher throughput than a Firecracker microVM [33]. The Lupine authors also report that, for realistic workloads, the extra benefit from removing the user/kernel boundary shrinks quickly, which they take as evidence that configuration specialisation, not the single address space itself, provides most of the gain [33].

4.4 Interpreting the evidence: threats to comparability

- **Boot-time definition.** Studies differ on when the clock starts and stops (monitor launch, guest entry, application ready), so numbers are not interchangeable; Unikraft explicitly separates guest and monitor time {unikraft}.
- **Application selection.** Many unikernels run only a limited set of applications, so comparisons may be restricted to workloads each system can run; the Lupine authors note that application choice was limited by what the unikernels could execute {lupine}.
- **Different baselines.** “Linux guest”, “Docker” and “microVM” baselines differ in kernel configuration, root file system and monitor, all of which affect results.
- **Absence of full cold-start measurement.** Boot time covers only part of the cold-start pipeline (Section 2.2); runtime and application initialisation are typically not included.

For these reasons, the defensible conclusion is that unikernels and unikernel-like systems consistently report start-up in the single-digit to tens of milliseconds and memory footprints of megabytes, which is one to two orders of magnitude below conventional virtual machines. Whether they win in an end-to-end serverless benchmark depends on the workload and remains underexplored.

5. Unikernels in Serverless Platforms

5.1 Reference architecture

Figure 8 sketches how unikernels would fit into a FaaS platform. The control plane—gateway, autoscaler, scheduler and image registry—decides which host should run a function. On each worker host a node agent starts a virtual machine monitor per function instance; the guest is a unikernel image rather than a general-purpose operating system. A host-local cache of images or snapshots supports fast restarts, and observability components collect traces and metrics from all layers.

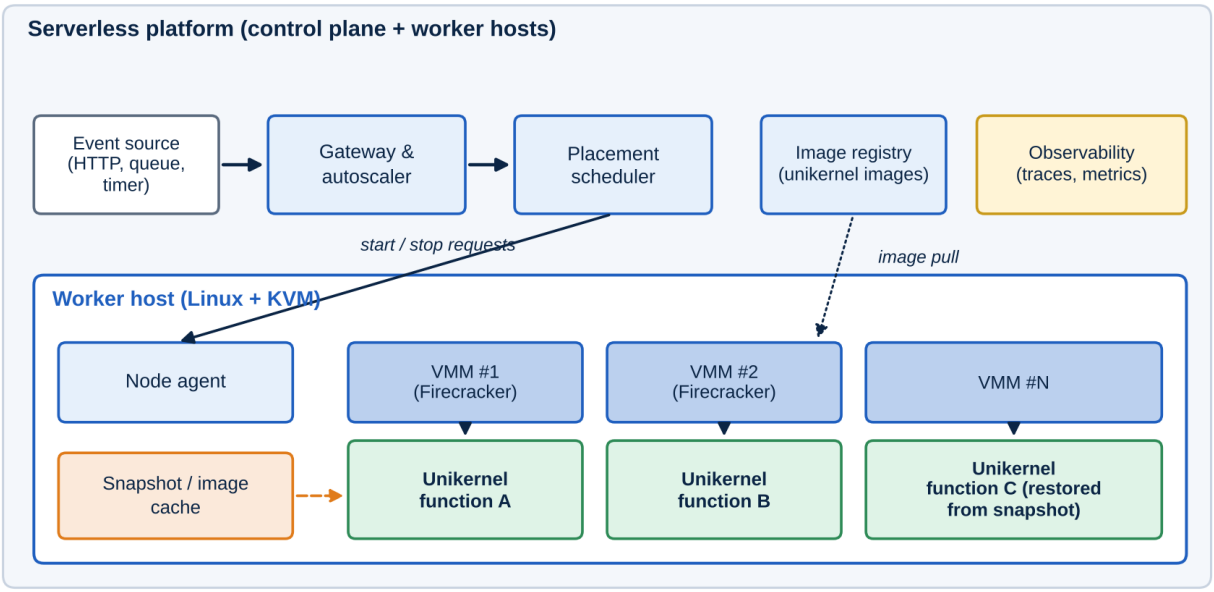


Figure 8. *Reference architecture of a serverless platform using unikernel images on KVM-based worker hosts. This is a conceptual design synthesised for this paper, not the architecture of a specific product.*

5.2 Combining unikernels with snapshots

Snapshotting complements lightweight guests by attacking the stages that boot-time reduction cannot reach. SEUSS captures snapshots of unikernel-based execution environments so that initialised runtimes can be restored instead of rebuilt [14]. Catalyzer reduces start-up by restoring pre-initialised state and sharing memory between instances, reporting sub-millisecond start for some workloads [15]. The vHive study of microVM snapshots found that restoring a snapshot is not free: much of the delay comes from page faults while the guest re-loads its working set, which motivated recording and prefetching that working set [13]. The lesson transfers directly to unikernels: a small guest image reduces the size of what must be restored, but the restore path must still be engineered.

5.3 Development accessibility

Adoption depends on whether ordinary developers can build and deploy unikernels. Projects such as Nanos aim to simplify unikernel development and deployment by supporting familiar languages and workflows, including Node.js, so that developers can reuse existing skills without detailed knowledge of low-level systems programming [36]. Unikraft takes a complementary route by providing a configuration-driven build system and aiming to run existing applications with limited porting effort [3], [35], while MirageOS requires developers to adopt OCaml [26].

The trade-off is between specialisation and convenience. Language-specific unikernels offer strong guarantees—type safety in OCaml, for example—but narrow the developer pool. POSIX-compatible systems widen the pool but must implement more of the Linux interface, which increases code size and the risk of behavioural differences [3], [33].

5.4 Artifact transparency and reproducibility

Research transparency is essential for validating experimental results. Accordingly, research artifacts, datasets, benchmarking configurations and analysis tools should be documented and made publicly accessible as open-source resources. Good practice already exists: the Unikraft authors published an artifact repository that maps each figure and table of the EuroSys paper to scripts, with estimated run times for each experiment [39], and the Lupine authors released their kernel configurations and scripts for running Firecracker experiments [34]. The ACM's artifact review and badging policy provides a shared vocabulary for artifacts that are available, functional, reusable and results-reproduced [40]. Such practices let the community independently verify reported improvements.

5.5 Comparative assessment (author's synthesis)

Table 6 condenses the preceding discussion into qualitative ratings. These ratings are the author's synthesis of the cited literature, *not* measurements, and are intended to structure the discussion rather than to rank products.

Table 6. Qualitative comparison matrix (author's synthesis of the literature reviewed).

Criterion	Container	Linux microVM	Classic unikernel	Linux-like unikernel
Start-up latency	Good	Moderate	Very good	Very good
Memory footprint	Very good	Moderate	Very good	Good
Isolation strength	Moderate	Very good	Very good	Very good
Application compatibility	Very good	Very good	Limited	Good
Debuggability and tooling	Very good	Good	Limited	Moderate
Production maturity	Very good	Very good [1]	Limited	Emerging

6. Limitations and Research Gaps

Despite considerable performance advantages, unikernels continue to face technical and practical challenges that limit adoption in production. Figure 9 maps each limitation to a research direction; the following subsections discuss them.

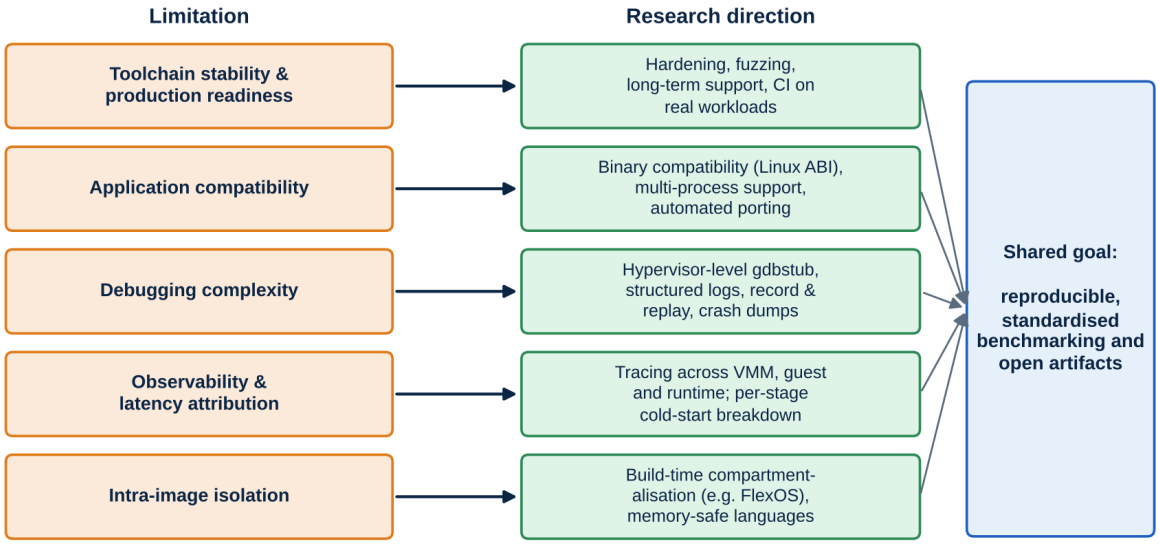


Figure 9. *Mapping from current limitations to research directions, all converging on reproducible and standardised evaluation.*

6.1 Stability and production readiness

Some unikernel toolchains remain less mature than established Linux-based microVM solutions in terms of production stability, operational reliability and ecosystem support. The contrast is visible in the record: Firecracker has run production workloads for AWS Lambda and Fargate since 2018 [4], [1], whereas most unikernel systems are research prototypes or maintained by small communities. Maturity includes long-term support, security patching, device-driver coverage and continuous testing across hardware. Further engineering effort is needed to make unikernel toolchains robust enough for large-scale deployment. The Lupine authors' recommendation to avoid re-implementing Linux's well-tested components reflects this concern [34].

6.2 Application compatibility

Porting complex, multi-process applications to a single-address-space library operating system can be challenging. Differences in system interfaces, process management and dependencies may require substantial modification; Unikraft's own framing of the problem is that unikernels are "infamous" for the engineering effort needed to port applications [3]. Some designs deliberately omit features such as *fork*; Nanos, for example, follows a single-process model [36]. Nevertheless, several projects are making progress. LibrettOS explores a multiserver-library OS design that can adapt dynamically [41], HermiTux runs unmodified Linux binaries [32], and Unikraft supports existing applications through modular libraries and a compatibility layer [3]. Remaining gaps include dynamic linking, multi-process workloads and the long tail of Linux system-call behaviour.

6.3 Debugging complexity

The absence of a conventional shell and standard operating-system debugging utilities in minimal unikernel images complicates fault diagnosis. Without *ssh*, *strace* or a package manager inside the image, developers need specialised mechanisms and external tools to investigate failures in production. Practical approaches include debugging through the hypervisor's debug stub, capturing crash dumps from guest memory, structured logging to a virtual serial or network channel, and building a richer "debug flavour" of the same image. Research is needed on record-and-replay, post-mortem analysis and developer tooling that preserves the small footprint of the production image.

6.4 Observability and performance attribution

Fine-grained latency attribution remains a significant challenge in serverless environments. Identifying the precise sources of execution delay requires advanced profiling, tracing and monitoring that can distinguish between application-level overhead, virtualisation cost and infrastructure latency. Studies of snapshot restoration show how easily time is mis-attributed: in vHive, a large share of restore time was found to be caused by page faults on the guest's working set rather than by the number of bytes in the snapshot [13]. Measurement across providers is further complicated by opaque platform internals [11].

A unikernel exacerbates the problem because there is no in-guest toolbox. Attribution therefore depends on instrumentation built into the monitor and the image: timestamps for monitor start, guest entry, runtime ready and first-request-served, collected against a common clock. Section 7.3 proposes a protocol for this.

6.5 Isolation within a single address space

A single address space improves performance but removes internal protection: a vulnerability in application code can compromise the whole image, including its networking stack, although the hypervisor boundary still protects the host and other tenants. FlexOS addresses this by making isolation between components a build-time choice, so the same code can be compartmentalised with different mechanisms depending on the threat model [42]. Memory-safe implementation languages, as in MirageOS's use of OCaml, offer another route [8]. Evaluating the cost of these protections in a serverless context is an open question.

6.6 State sharing and density

Containers on one host can share the kernel's page cache and common libraries, whereas many identical unikernels each carry their own copy of the image in memory. As a consequence, a fleet of small unikernels is not automatically more memory-efficient than the same number of containers; techniques such as copy-on-write cloning and snapshot sharing [15], [14] are needed to recover the density advantage. This is the author's analytical observation, and it should be tested empirically.

7. Methodological Recommendations

To ensure that innovations in this area have lasting impact on the systems research community, future work should prioritise three complementary objectives: clear architectural visualisations, transparent benchmarking based on standardised evaluation suites, and the publication of high-quality, reproducible open-source artifacts. This section turns those objectives into concrete practice.

7.1 Standardised benchmarks and workloads

Comparisons are most useful when they use shared workloads. SeBS is a benchmark suite for FaaS that defines workloads and a methodology for measuring cold and warm invocations across platforms [43]. vHive provides an experimentation framework for serverless systems built around microVM snapshots [13]. Realistic invocation patterns can be drawn from the published characterisation of a production function workload [16]. Studies of unikernels should evaluate at least one standardised serverless workload alongside application micro-benchmarks such as nginx, Redis and SQLite, which dominate current unikernel evaluations [3], [33].

7.2 Metric definitions and statistical practice

Every paper should state exactly which events delimit "start-up" (for example, monitor launch to first successful request), separate cold from warm invocations, and report distributions rather than single values: median and tail percentiles (p95, p99), the number of repetitions, and variability across runs. Hardware model, firmware, host kernel, hypervisor and monitor versions, guest configuration and file-system type should all be disclosed, because each affects boot time [33], [3].

7.3 A latency-attribution protocol

To address the attribution gap in Section 6.4, the following protocol is proposed as a starting point. It is a recommendation of this paper, not an established standard.

Table 7. Proposed event timeline for cold-start latency attribution.

Step	Event to timestamp	Where measured
1	Request arrives at gateway	Control plane
2	Placement decision made; start command issued	Scheduler / node agent
3	Monitor process created; guest memory prepared	Host (monitor)
4	Guest entry point reached	Monitor, using shared clock
5	Runtime initialised and application ready	Guest (early log / vsock message)
6	First response byte sent	Guest and gateway

Differences between consecutive timestamps give the cost of scheduling, sandbox creation, guest boot and runtime initialisation respectively, mapping directly onto the decomposition in Section 2.2. Using a single time base on one host avoids clock-skew artefacts; the same method can be applied to containers and microVMs so that comparisons are like-for-like.

7.4 Reproducibility checklist

Table 8. Reproducibility checklist for unikernel and serverless research artifacts.

Item	Recommended practice
Source and build	Public repository; pinned commits and dependency versions; scripted build of every image.
Experiment scripts	One script per figure or table, with expected run time, as in the Unikraft artifact [39].
Environment	Hardware, firmware, kernel, hypervisor and monitor versions recorded; container or VM image for the host environment where possible.
Data and traces	Raw measurements and traces archived with a persistent identifier; analysis notebooks included.
Badging	Seek artifact review and badges (available, functional, reusable, reproduced) [40].
Limitations	State known threats to validity and workloads where the system fails, not only where it succeeds.

7.5 Clear architectural visualisation

Architectural diagrams help readers see what is and is not included in an image. Good practice is to show (i) the layers present in each compared system, as in Figure 4; (ii) the build and boot pipeline, as in Figure 5; and (iii) the deployment context, including the monitor and host, as in Figure 8. Diagrams should be consistent across systems so that readers can compare like with like.

8. Discussion

RQ1 – Which cold-start stages depend on the isolation technology? The sandbox-creation and guest-boot stages are directly affected by the choice of container, microVM or unikernel; scheduling, language-runtime start-up and application initialisation are largely independent of it (Section 2.2). Unikernels therefore address a necessary but not sufficient part of the problem.

RQ2 – How do unikernels differ architecturally? A unikernel merges application and operating-system libraries into one single-address-space image specialised at build time, whereas a virtual machine boots a full guest OS, a container shares the host kernel, and a microVM boots a trimmed Linux kernel on a minimal monitor (Section 3, Figure 4).

RQ3 – What does the quantitative evidence show? Published studies report millisecond to tens-of-milliseconds boot, megabyte-scale images and memory, and throughput gains over Linux-guest baselines (Section 4). The evidence is consistent in direction but not directly comparable, and it rarely measures full cold-start latency.

RQ4 – What limits adoption? Toolchain maturity, application compatibility, debugging and observability are the main practical barriers, with isolation inside the single address space and memory-sharing across instances as additional research questions (Section 6).

8.1 Threats to validity of this review

This review is limited in four ways. It relies on published results and does not independently reproduce them. The selection of systems is representative rather than exhaustive. Technology evolves quickly, and project documentation may have changed since the cited versions. Finally, the qualitative ratings in Table 6 reflect the author’s judgement. Readers should consult the original sources before relying on any individual figure.

9. Conclusion

Unikernels offer a compelling architectural approach to mitigating the cold-start problem in serverless computing by substantially simplifying the virtualisation stack. Through optimised boot procedures, minimal device-driver footprints and aggressive memory reduction, research initiatives have demonstrated millisecond-level start-up that is difficult to achieve with general-purpose operating systems [2], [3], [33].

However, this paper has argued that a faster guest boot is only one component of cold-start latency. Language-runtime start-up and application initialisation remain, and the most promising designs combine small specialised guests with snapshot and restore techniques [14], [15], [13]. Several challenges must also be addressed before unikernels can achieve broader production adoption: improving toolchain stability, expanding application compatibility, simplifying debugging, and developing more sophisticated observability mechanisms.

Future research should prioritise (i) end-to-end cold-start benchmarks on standardised serverless workloads [43]; (ii) a consistent latency-attribution method across containers, microVMs and unikernels; (iii) evaluation of unikernel–snapshot combinations; (iv) the cost of intra-image isolation [42]; and (v) the publication of reproducible open-source artifacts [40], [39]. Addressing these gaps will help establish unikernels as a viable foundation for high-performance, resource-efficient and low-latency serverless computing infrastructures.

References

References are numbered in order of first citation. Entries are in IEEE style; DOIs are given where known. Project web pages were consulted as primary sources for tool descriptions and should be re-checked for later changes.

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, USA, 2020, pp. 419–434.

[2] F. Manco *et al.*, "My VM is lighter (and safer) than your container," in *Proc. 26th ACM SOSP*, Shanghai, China, 2017, pp. 218–233, doi: 10.1145/3132747.3132763.

[3] S. Kuenzer *et al.*, "Unikraft: Fast, specialized unikernels the easy way," in *Proc. 16th European Conf. Computer Systems (EuroSys)*, Online, 2021, pp. 376–394, doi: 10.1145/3447786.3456248.

[4] J. Barr, "Firecracker – Lightweight virtualization for serverless computing," *AWS News Blog*, Nov. 2018. [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing

[5] E. Jonas *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2019-3, Feb. 2019, arXiv:1902.03383.

[6] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[7] S. Eismann *et al.*, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[8] A. Madhavapeddy *et al.*, "Unikernels: Library operating systems for the cloud," in *Proc. 18th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, USA, 2013, pp. 461–472.

[9] A. Madhavapeddy and D. J. Scott, "Unikernels: The rise of the virtual library operating system," *Commun. ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[10] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gVisor case study," in *Proc. 11th USENIX HotCloud*, Renton, WA, USA, 2019.

[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC*, Boston, MA, USA, 2018, pp. 133–146.

[12] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX ATC*, Boston, MA, USA, 2018, pp. 57–70.

[13] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM ASPLOS*, 2021.

[14] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip redundant paths to make serverless fast," in *Proc. 15th EuroSys*, Heraklion, Greece, 2020.

[15] D. Du *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th ACM ASPLOS*, Lausanne, Switzerland, 2020.

[16] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX ATC*, 2020, pp. 205–218.

[17] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM ASPLOS*, 2021.

[18] I. E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. USENIX ATC*, Boston, MA, USA, 2018, pp. 923–935.

[19] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. 11th USENIX HotCloud*, Renton, WA, USA, 2019.

[20] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[21] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. 19th ACM SOSP*, Bolton Landing, NY, USA, 2003, pp. 164–177.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: The Linux virtual machine monitor," in *Proc. Ottawa Linux Symposium*, 2007, pp. 225–230.

[23] D. Merkel, “Docker: Lightweight Linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, Art. 2, 2014.

[24] Firecracker Project, “Secure and fast microVMs for serverless computing.” [Online]. Available: https://firecracker-microvm.github.io

[25] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr., “Exokernel: An operating system architecture for application-level resource management,” in *Proc. 15th ACM Symp. Operating Systems Principles (SOSP)*, Copper Mountain, CO, USA, 1995, pp. 251–266.

[26] MirageOS Project, “MirageOS: A library operating system for constructing unikernels.” [Online]. Available: https://mirage.io

[27] A. Kivity *et al.*, “OSv—Optimizing the operating system for virtual machines,” in *Proc. USENIX Annual Technical Conf. (ATC)*, Philadelphia, PA, USA, 2014, pp. 61–72.

[28] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, “IncludeOS: A minimal, resource efficient unikernel for cloud services,” in *Proc. IEEE 7th Int. Conf. Cloud Computing Technology and Science (CloudCom)*, Vancouver, Canada, 2015, pp. 250–257.

[29] A. Kantee, “Flexible operating system internals: The design and implementation of the anykernel and rump kernels,” Ph.D. dissertation, Dept. Comput. Sci. Eng., Aalto Univ., Espoo, Finland, 2012.

[30] D. Williams and R. Koller, “Unikernel monitors: Extending minimalism outside of the box,” in *Proc. 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Denver, CO, USA, 2016.

[31] Solo5 Project, “Solo5: A sandboxed execution environment for unikernels.” [Online]. Available: https://github.com/Solo5/solo5

[32] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, “A binary-compatible unikernel,” in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments (VEE)*, Providence, RI, USA, 2019.

[33] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, “A Linux in unikernel clothing,” in *Proc. 15th European Conf. Computer Systems (EuroSys)*, Heraklion, Greece, 2020, Art. 11, doi: 10.1145/3342195.3387526.

[34] H.-C. Kuo, “Lupine-Linux: Linux in unikernel clothes” (source code repository). [Online]. Available: https://github.com/hckuo/Lupine-Linux

[35] Unikraft Project (Linux Foundation), “Unikraft: A fast, secure and open-source unikernel development kit.” [Online]. Available: https://unikraft.org

[36] NanoVMs, Inc., “Nanos: A unikernel kernel” and “OPS” tooling. [Online]. Available: https://nanos.org and https://github.com/nanovms/nanos

[37] A. Raza *et al.*, “Unikernel Linux (UKL),” in *Proc. 18th European Conf. Computer Systems (EuroSys)*, Rome, Italy, 2023.

[38] D. Williams, R. Koller, M. Lucina, and N. Prakash, “Unikernels as processes,” in *Proc. ACM Symp. Cloud Computing (SoCC)*, Carlsbad, CA, USA, 2018.

[39] Unikraft Project, “Unikraft EuroSys’21 artifacts.” [Online]. Available: https://github.com/unikraft/eurosys21-artifacts

[40] Association for Computing Machinery, “Artifact review and badging, version 1.1,” Aug. 2020. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[41] R. Nikolaev, M. Sung, and B. Ravindran, “LibrettOS: A dynamically adaptable multiserver-library OS,” in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments (VEE)*, 2020.

[42] H. Lefeuvre *et al.*, “FlexOS: Towards flexible OS isolation,” in *Proc. 27th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[43] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “SeBS: A serverless benchmark suite for function-as-a-service computing,” in *Proc. 22nd Int. Middleware Conf.*, 2021.

Appendix A. Glossary

Table 9. Glossary of terms.

Term	Meaning
Cold start	Latency incurred when a platform must create a new execution environment before running a function.
Warm start	Invocation served by an already-running, idle execution environment.
FaaS	Function-as-a-Service: a serverless model in which short functions run in response to events.
Unikernel	A single-purpose, single-address-space virtual machine image that links an application with only the OS libraries it needs.
LibOS	Library operating system: OS functionality provided as libraries linked into the application.
MicroVM	A virtual machine with a minimal device model and monitor designed for fast start and high density.
VMM	Virtual machine monitor; software that creates and runs virtual machines (e.g. Firecracker, QEMU).
KVM	Kernel-based Virtual Machine: Linux's hardware-assisted virtualisation interface.
Hypervisor	Software layer that multiplexes hardware among virtual machines (e.g. Xen, KVM).
Snapshot / restore	Saving the memory and device state of an initialised sandbox and resuming it later.
LTO / DCE	Link-time optimisation / dead-code elimination, used to shrink images.
virtio	Standard family of paravirtual devices (network, block, vsock) used by guests under KVM.
Tail latency (p99)	The value below which 99% of observed latencies fall; used to characterise worst-case behaviour.
Artifact	The code, data and scripts that accompany a paper so others can reproduce its results.

Appendix B. Notes on Sources and Illustrations

Verification. The headline quantitative claims in Section 4 were cross-checked against the original paper abstracts or the project's own documentation: Firecracker (project site and AWS announcement), Unikraft (EuroSys 2021 abstract), LightVM (SOSP 2017 abstract) and Lupine Linux (EuroSys 2020 paper). All other statements are descriptions of the cited works.

Illustrations. Figure 1 is the illustration supplied by the author and is also used for the cover and the running page header. Figures 2–9 were drawn specifically for this paper. They are schematic: no figure presents original measurement data, and Figure 3 and the hypothetical example in Section 2.2 are explicitly illustrative.

Evidence limits. Where this paper states a limitation or proposes a method (the latency-attribution protocol in Section 7.3, the comparison matrix in Table 6, and the density observation in Section 6.6), that content is the author's own analysis and is identified as such in the text.