LYNX

Certification Complexity Is Not a VM Count

Why evidence boundaries, software commonality, and configuration variants matter more than the number of virtual machines in modern avionics



Virtualization has become an increasingly important architectural tool in modern avionics. It allows multiple operating environments to coexist on shared computing resources, enables consolidation of mixed-criticality workloads, and creates opportunities to separate applications from the hardware on which they execute. Yet as virtualization becomes more common in safety-critical systems, an understandable concern has emerged: does introducing more virtual machines also introduce more certification complexity?

The concern is legitimate. Every software component, execution environment, configuration, and interface can introduce requirements that must be understood, verified, and controlled. A poorly designed virtualized architecture may multiply software baselines, complicate traceability, increase regression testing, and create additional certification work. But the reverse conclusion, that fewer virtual machines necessarily produce a simpler certification effort, is equally questionable.

Certification complexity is not determined by how many virtual machines an architecture contains. It is determined by what must be assured, how responsibilities are allocated, which dependencies are shared, and how changes propagate through the system.

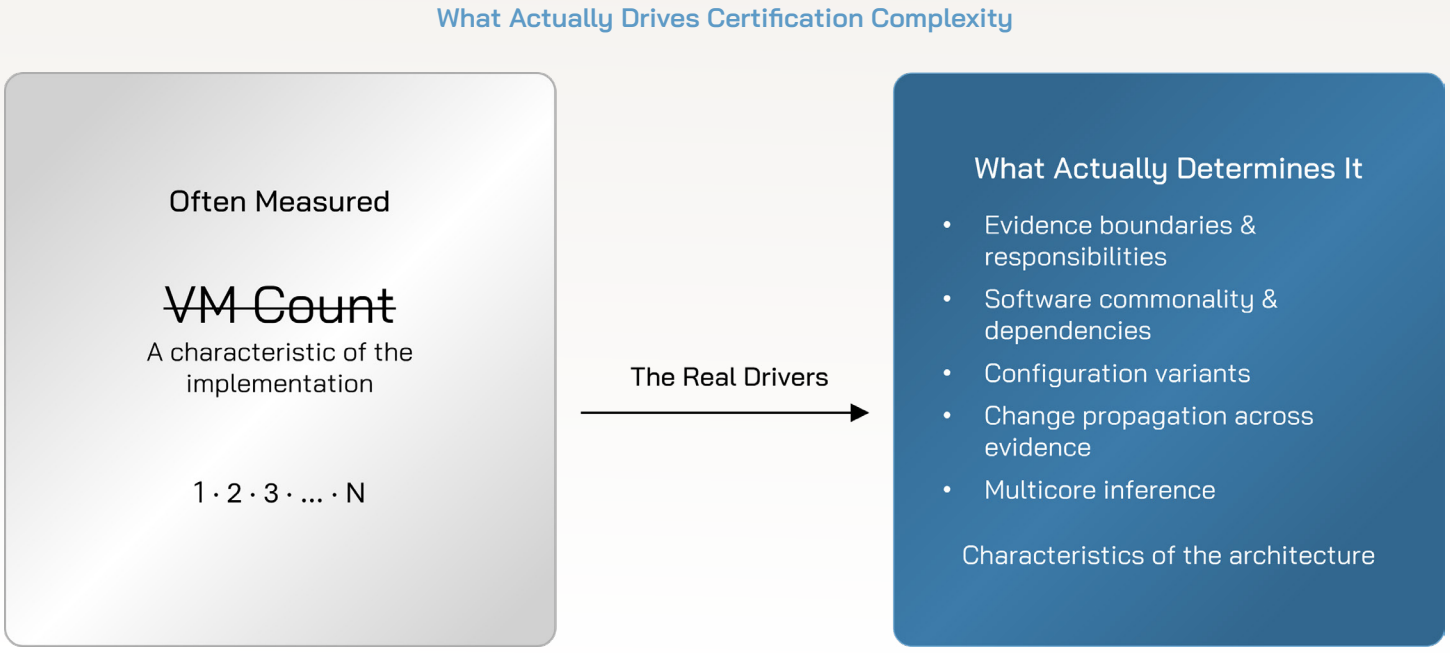


Figure 1. VM count is a property of the implementation; certification is driven by the architecture.

This distinction is becoming increasingly important as avionics architectures move toward multicore processors, heterogeneous computing, mixed-criticality consolidation, and eventually AI-enabled mission capabilities.

The question is no longer simply how many applications can share a processor. It is how an architecture can preserve justified confidence while those applications, their operating environments, and the underlying hardware evolve over decades of service.

The Difference Between Execution Boundaries and Certification Boundaries

A virtual machine establishes an execution environment. Depending on the underlying architecture, it can provide memory isolation, resource allocation, device access controls, and mechanisms for separating workloads. Those capabilities can contribute significantly to an assurance argument. However, an execution boundary is not automatically a certification boundary. A certification boundary is defined by the software and system responsibilities being assured, the applicable objectives, the interfaces and dependencies involved, and the evidence necessary to demonstrate compliance. It also depends on the roles and assumptions established for the system integrator, platform supplier, application developers, and certification applicant. This is particularly important in Integrated Modular Avionics (IMA), where RTCA DO-297 addresses the roles and responsibilities associated with integrating independently developed components into a common platform.

Consider two architectures. The first hosts ten applications within a common operating environment. The second hosts those applications in ten separately configured virtual machines. At first glance, the second architecture appears more complicated. It contains more execution environments and potentially more software images. But suppose the first architecture allows applications to share operating-system services, device drivers, memory-management mechanisms, and other dependencies. A modification to one shared component may require analysis across multiple applications because the effects of that modification are not necessarily confined to a single function.

Now suppose the second architecture establishes independently controlled execution environments with stable interfaces and explicitly defined resource ownership. If the platform can demonstrate the relevant isolation and non-interference properties, a modification to one application may have a more limited impact on the assurance arguments for other applications.

Neither outcome is automatic. A virtualized architecture can introduce substantial integration and verification work. A shared operating environment can also be engineered with strong partitioning and carefully managed dependencies. The important point is that counting virtual machines reveals very little about the actual scope of the certification argument.

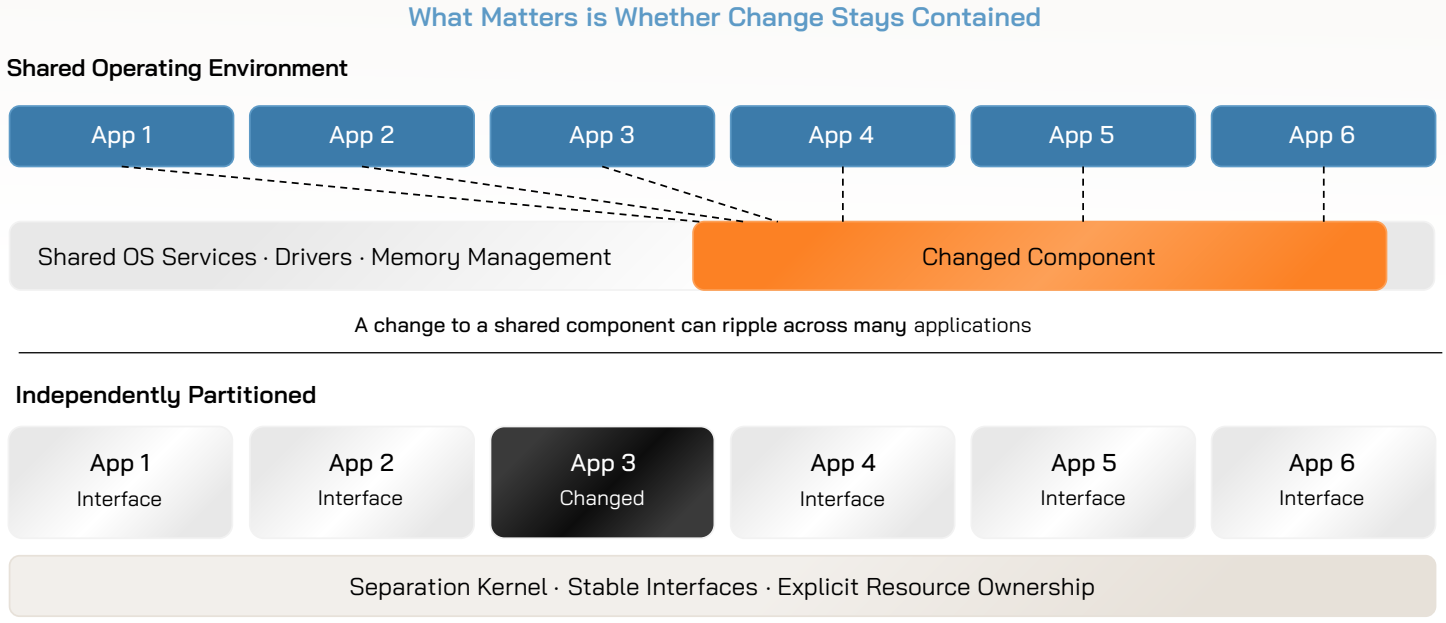


Figure 2. What matters is whether the change stays contained, not how many execution environments exist.

Ten virtual machines do not necessarily represent ten independent certification efforts, just as ten applications within one operating system do not necessarily represent one. What matters is where assurance responsibilities reside and whether the architecture can demonstrate that changes remain contained.

Commonality: The Overlooked Variable in Certification Economics

One of the strongest arguments for conventional platform architectures is software commonality.

If multiple applications depend on a common operating system, shared libraries, or platform service, the organization can potentially reuse a substantial body of development and verification evidence. This is an important advantage, particularly when common components have mature certification artifacts and clearly defined usage assumptions.

However, commonality introduces an architectural trade-off that is often overlooked.

A common component can reduce duplicated engineering effort while simultaneously increasing the number of applications affected when that component changes.

Consider a shared operating-system service used by several safety-critical applications. During initial development, the common implementation may reduce duplication and simplify verification. But when the service is modified, the integrator must evaluate whether the change affects any application relying on its behavior.

The impact may be small. It may also be substantial, depending on the nature of the modification, the interfaces involved, and the assumptions supporting the existing evidence. Commonality therefore creates both an opportunity and a dependency.

The same reasoning applies to unikernels and other lightweight operating environments. If each application is packaged with its own runtime, the organization must control the configuration identity and verification evidence associated with those images. If every image contains independently modified runtime components, configuration complexity can increase significantly.

But if the images are built from a controlled, common software baseline with well-defined variants, reproducible builds, explicit usage assumptions, and appropriate evidence reuse, the engineering problem is different. The existence of multiple images does not mean that every line of common software must necessarily be developed and verified independently for every image. Nor does common source code automatically entitle a program to reuse evidence across different configurations. The actual reuse argument depends on what is common, what has changed, and whether the prior evidence remains applicable. This suggests that avionics architects should distinguish between three forms of commonality.

Source commonality means multiple components originate from the same controlled software baseline.

Configuration commonality means the relevant build options, dependencies, interfaces, and operating assumptions remain equivalent.

Evidence commonality means previously generated assurance artifacts remain applicable to the configuration and intended use being evaluated.



LYNX

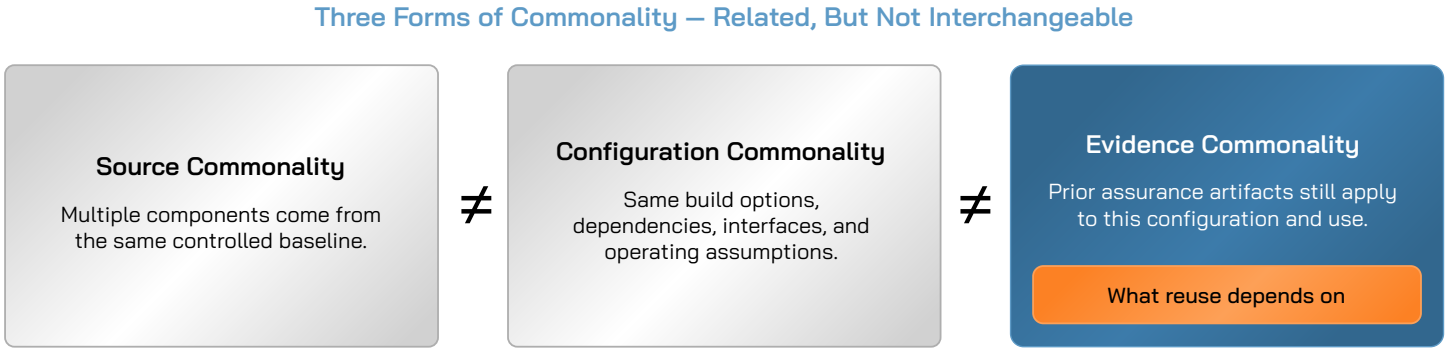


Figure 3. Source, configuration, and evidence commonality are related, but not interchangeable.

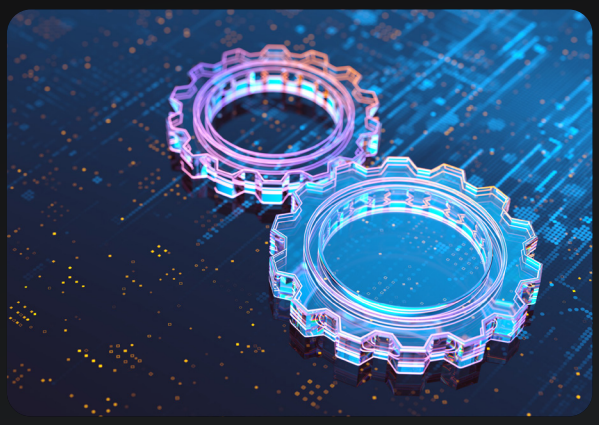
These forms of commonality are related, but they are not interchangeable. A common source baseline can produce materially different executable images. Different build configurations may expose different behaviors or dependencies. Evidence generated for one configuration may not fully address another.

Consequently, the central question is not whether software has been duplicated across virtual machines. It is whether the program has established a disciplined strategy for managing common software, configuration differences, and the evidence associated with each.

Configuration Variants: Where Complexity Can Really Multiply

Configuration management is frequently treated as an administrative discipline. In safety-critical software, it is much more than that. Configuration defines the precise software and hardware context to which an assurance argument applies.

A virtual machine image may incorporate an application, a runtime, device interfaces, build options, and other dependencies. Each variation can influence the evidence required to establish correct behavior. If a program allows these elements to diverge without control, certification complexity can grow rapidly.



But the same problem exists in conventional architectures. A shared operating system supporting different processor configurations, driver combinations, scheduling policies, memory layouts, or application sets also creates variants. Those variants may be less visible because they are packaged within a common platform, but their assurance implications do not disappear.

The relevant question is therefore not how many images exist. It is how many materially distinct configurations must be assured, and how effectively the architecture manages the differences between them.

A disciplined configuration strategy should identify which software elements are common, which parameters vary, which interfaces remain invariant, and which verification results depend on hardware or integration conditions.

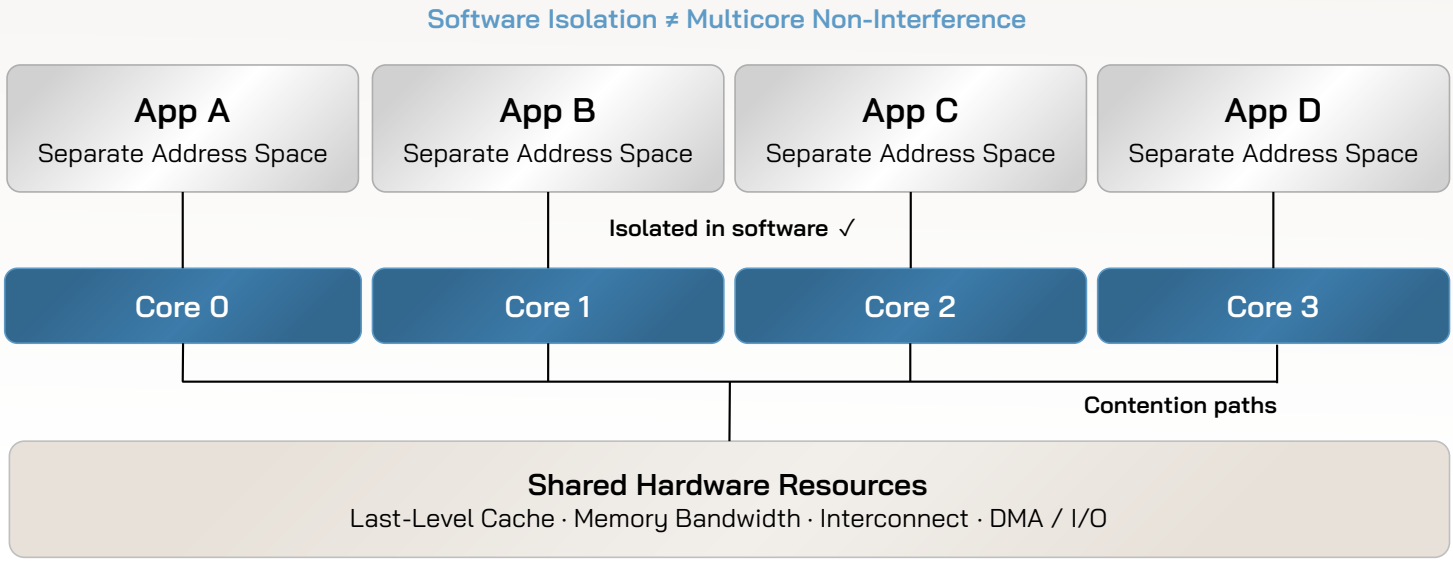
This becomes especially important in multicore systems. A configuration that behaves predictably on one processor may not exhibit equivalent timing behavior on another. Shared caches, memory controllers, interconnects, DMA activity, and other hardware resources can introduce interference paths that must be analyzed within the intended operational context.

Moving an application into a separate virtual machine does not automatically eliminate those interference paths. Equally, consolidating applications within one operating system does not automatically make them easier to characterize. The architecture must provide mechanisms and evidence appropriate to the system's actual interference risks.

This is why certification complexity should be evaluated at the intersection of software configuration, hardware behavior, and system integration — not inferred from the number of execution environments.

Multicore Makes Architectural Dependencies Impossible to Ignore

Multicore processors have fundamentally changed the assurance discussion because independently executing software can interact through hardware resources that are not obvious at the application level. Applications may have separate address spaces and still contend for shared caches, memory bandwidth, interconnects, or I/O resources.



Separate address spaces still contend for shared hardware.

Non-interference must be demonstrated against real hardware behavior — not inferred from software boundaries.

Figure 4. Software isolation does not establish multicore non-interference; shared hardware remains a contention path.

The challenge is not merely demonstrating that each application performs its intended function. It is establishing that interference from other workloads cannot invalidate the timing and behavioral assumptions on which safety depends. This is a system-level concern.

A well-designed separation architecture can provide important mechanisms for controlling resource ownership, partition scheduling, device access, and communication paths. These mechanisms can simplify aspects of the interference argument by making relationships more explicit.

However, software isolation alone cannot establish complete multicore non-interference. The analysis must address the relevant shared hardware resources and the assumptions under which the system operates.

This is an important distinction because it separates architectural claims from assurance evidence. An architecture may be designed to support strong partitioning. The certification program must still establish that the implemented configuration satisfies the applicable requirements.

The objective is not to claim that architecture eliminates verification. It is to design the system so that verification responsibilities are understandable, bounded where feasible, and repeatable as the platform evolves.

The Real Test: What Happens When Something Changes?

The initial certification effort is only part of the economic equation. Many avionics platforms remain operational for decades. During that time, applications change, processors are refreshed, cybersecurity vulnerabilities are addressed, new capabilities are introduced, and suppliers evolve their technologies. The cost of these changes can become a defining characteristic of the software architecture.

Imagine that a safety-critical application requires an update after the platform has entered service. In one architecture, the application depends on shared operating-system services and interfaces used by several other applications. The change-impact analysis must determine whether the modification affects those dependencies, their associated requirements, and previously established verification results.

In another architecture, the application executes within an independently controlled environment whose interfaces and resource allocations remain unchanged. If the applicable isolation assumptions remain valid, the change may be more readily confined to a smaller set of software components and assurance artifacts.

The distinction is not that one architecture has more virtual machines. It is that the two architectures may expose different change-propagation paths. This is where architectural design begins influencing lifecycle economics.

A useful certification strategy should therefore answer four questions whenever a change is introduced:

1. What changed, including source code, executable image, configuration, interfaces, tools, and hardware assumptions?
2. Which previously established requirements and assurance claims could be affected?
3. Which verification results remain applicable, and which activities must be repeated or extended?
4. Can the impact be demonstrated through objective evidence rather than assumed from the architecture diagram?

These questions apply equally to conventional RTOS architectures, virtualized IMA platforms, and unikernel-based systems. They also reveal why a VM count is an inadequate metric. The real measure is the extent to which a change propagates across assurance boundaries.

Evidence Should be Designed for Reuse, Not Reconstructed After Every Change

Evidence reuse is frequently discussed as though it were primarily a documentation problem. It begins with architecture.

When components have stable interfaces, controlled dependencies, clearly allocated responsibilities, and well-defined usage assumptions, it becomes easier to determine whether previously generated evidence remains relevant. Conversely, when dependencies are poorly understood, even small changes can require extensive investigation before engineers can establish which evidence remains applicable.

This is why reusable evidence should not be confused with reusable documents. The value of an assurance artifact lies in the claim it supports and the assumptions under which that claim remains valid. A test report, analysis, or verification result may remain useful across multiple configurations. But its applicability must be justified for each intended use. Reusing the file is not the same as reusing the assurance argument.

For modern IMA programs, the objective should be to establish an evidence architecture alongside the software architecture. That means defining how requirements, configurations, interfaces, verification results, platform assumptions, and change-impact decisions relate to one another. The result is not automatic certification reuse. It is a more disciplined foundation for determining what can legitimately be reused.

This is also where modern engineering automation and AI-assisted tools may eventually contribute significant value. Traceability analysis, configuration comparison, evidence completeness checking, and change-impact investigation are promising areas for automation. But automation does not replace the need for sound engineering assumptions. It makes those assumptions easier to manage and examine.

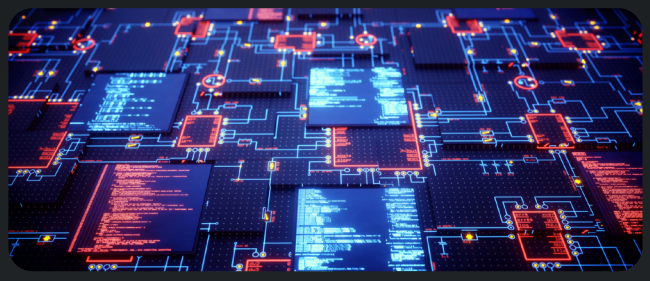
From Virtualization to Certification Architecture

The debate surrounding virtualization in avionics is sometimes framed as a choice between conventional integrated operating environments and collections of independently packaged virtual machines. That framing is too narrow. Both approaches can succeed. Both can fail. Both can produce architectures that are straightforward or difficult to certify.

The determining factors are architectural. How clearly are responsibilities allocated? How are shared resources controlled? Which software elements are genuinely common? How are configuration variants managed? How much evidence remains applicable after a change? How effectively can the system integrator establish that existing assurance claims remain valid?

These questions become even more consequential as mission systems incorporate increasingly heterogeneous workloads. A future avionics platform may need to accommodate safety-critical control software, advanced graphics, Linux-based services, AI inference, and specialized accelerator workloads. These components will not necessarily share the same operating environment, assurance objectives, or update cadence.

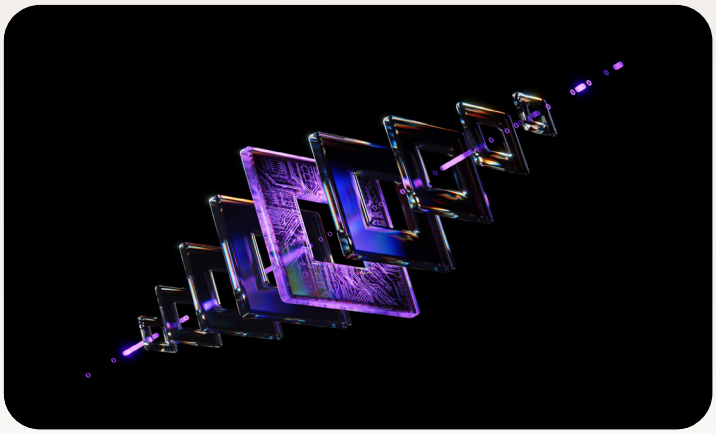
Trying to force them into one execution model may introduce unnecessary coupling. Separating them without a disciplined integration and evidence strategy may introduce unnecessary complexity. The objective is neither maximum consolidation nor maximum separation. It is an architecture that establishes the right boundaries for execution, assurance, and change. That is the deeper promise of composable mission-critical computing.



LYNX

The MOSA.ic™ Perspective: Certification by Architecture

At Lynx, this principle is central to the design philosophy behind MOSA.ic. Rather than treating virtualization as the objective, MOSA.ic uses a separation-kernel-based architecture to establish controlled execution environments, resource ownership, and partition boundaries. Different operating environments, including LynxElement® and LynxOS-178, can be selected according to the needs of the workloads they support. The architectural intent is to separate responsibilities that do not need to evolve together. A lightweight application environment need not inherit every service and dependency of a richer operating system. Conversely, applications requiring established POSIX functionality should not be forced into a minimal runtime merely for architectural uniformity.



The separation kernel provides a common foundation for controlled coexistence, while the system's configuration, integration strategy, and assurance activities establish how those mechanisms are used in a particular program. This approach does not remove the need to verify applications, characterize multicore interference, manage configuration variants, or satisfy system-level certification objectives. It provides an architectural basis for making those responsibilities explicit and for evaluating whether changes can be contained.

Lynx explores these principles in greater depth in its white paper, Certification That Compounds, which examines how architecture, evidence reuse, and controlled change can influence certification economics over a program's lifecycle. The underlying principle is straightforward: the value of an architecture should not be judged only by the effort required to establish the first certified baseline. It should also be judged by the effort required to sustain that baseline as the system evolves.

A Better Measurement of Certification Complexity

Perhaps it is time to change the question. Instead of asking how many virtual machines a system contains, architects and certification teams should ask how much of the system must be reconsidered when one component changes. That question brings the discussion back to the engineering realities that matter.

Execution boundaries matter because they can help control interactions. Commonality matters because it can enable evidence reuse while introducing shared dependencies. Configuration variants matter because they determine which assumptions and verification results remain applicable. Multicore interference matters because isolation must be demonstrated against the behavior of real hardware, not inferred from software boundaries alone.

And lifecycle architecture matters because certification is not merely a challenge encountered at initial deployment. Assurance must be maintained through the changes that follow. None of these concerns can be reduced to a VM count. The number of virtual machines is a characteristic of the implementation. The scope of change, the quality of the evidence, and the ability to preserve assurance are characteristics of the architecture.

For programs expected to evolve over decades, that is the distinction worth understanding.

Learn more

https://www.lynx.com/resource-library/lynx-mosa.ic-certification-whitepaper
https://www.lynx.com/resource-library/lynx-mosa.ic-technical-whitepaper



© 2026 Copyright Lynx

The information herein is subject to change at any time after the date of publication. Lynx does not guarantee the accuracy of the information herein beyond the date of publication. All third-party company and product names mentioned, and marks and logos used, are trademarks and/or registered trademarks of their respective owners.

Ready to Revolutionalize Your Mission-Critical Systems?

Contact Lynx to discuss how LYNX MOSA.ic can help establish clear execution and assurance boundaries, manage configuration changes, and support assurance throughout your system's lifecycle.

edge@lynx.com

www.lynx.com