The Modernization of the Windows Desktop

From Legacy Monoliths to Cloud-Native Composition

Executive Summary

Legacy Windows applications have been the backbone of businesses for years. However as technology advances these applications face aging challenges. These are software programs developed using older technologies such as Visual Basic, C++ or .NET, often running on outdated operating systems.

This guide details the technical challenge and a modernization roadmap to remediate the situation and ensure applications form a dynamic component of an organization's distributed and virtual digital workplace strategy.

The Legacy Estate and the Crisis of Technical Debt

The enterprise technology landscape stands at a precipice of transformation, driven by the convergence of remote work demands, cybersecurity imperatives, and the migration to cloud-hosted infrastructure.

Yet, for thousands of organizations globally, this forward momentum is anchored by a massive, inertia-heavy substrate: the legacy Windows application estate.

These applications, often numbering in the thousands within a single enterprise, constitute the operational backbone of global business, yet they were engineered for an era of computing that is rapidly becoming obsolete.

The challenge facing modern IT architects is not merely one of "installing software," but of fundamentally re-architecting the relationship between the application, the operating system, and the user identity. To understand the magnitude of this challenge, one must first dissect the anatomy of the legacy estate and the "technical debt" it represents.

The Architecture of Entrenchment: Win32, COM, and the Registry

For over two decades, the "Win32" API (Application Programming Interface) has been the dominant standard for Windows software development.

Applications built on this framework—along with those utilizing the Component Object Model (COM) and early versions of the.NET Framework—were designed with a presumption of permanence and singularity.

In the traditional "Classic IT" model, an application installer (typically an MSI or Setup.exe) functions as a deep integration agent. It does not merely place files on a disk; it weaves the application into the very fabric of the operating system.

When a legacy application installs, it performs a series of invasive operations. It copies executables and libraries to C:\Program Files, often scattering shared DLLs (Dynamic Link Libraries) into C:\Windows\System32.

Simultaneously, it writes thousands of configuration entries into the Windows Registry—a hierarchical, monolithic database that stores settings for the OS and all installed software. This architecture creates a tight coupling between the application and the specific machine it resides on. The application expects to find its dependencies in

specific absolute paths; it expects its registry keys to persist in specific hives (HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER); and, frequently, it expects to run with elevated privileges that allow it to modify these system-critical areas.

This tight coupling is the root cause of "WinRot," a phenomenon where the operating system's performance and stability degrade over time.

As applications are installed, patched, and removed, they leave behind "orphaned" artifacts—registry keys that point to non-existent files, DLLs that are no longer referenced but remain loaded in memory, and configuration files that clutter the file system.

The Windows Registry, in particular, is prone to "bloat." Research indicates that excessive registry growth can lead to significant performance penalties, including slow boot times, delayed user logons, and even system resource exhaustion errors (such as 0x800705AA), where the kernel lacks sufficient paged pool memory to initialize new processes.

Furthermore, the reliance on shared components leads to the infamous "DLL Hell," where two applications require different versions of the same shared library. In a monolithic environment, installing Application B might overwrite a DLL required by Application A, causing Application A to crash. This fragility forces IT organizations to engage in rigorous, time-consuming regression testing every time a core component or OS update is deployed, effectively freezing the environment in a state of fear-driven stasis.

The Security Implications of "Admin Mode"

A critical dimension of this technical debt is security. Many legacy applications were developed in an era before the modern threat landscape existed, often adhering to the assumption that the user would have local administrative rights.

These applications frequently attempt to write configuration updates to their own installation directories in C:\Program Files or modify global registry keys in HKLM during runtime. In a modern "Zero Trust" environment, where users operate with Standard User privileges to mitigate the blast radius of malware, these operations are blocked by the OS, causing the application to crash or malfunction.

To mitigate this, IT administrators have historically resorted to dangerous compromises: relaxing folder permissions, creating "shims" that bypass security checks, or, in the worst cases, granting users local admin rights. These workarounds accumulate as

"security debt," leaving the enterprise vulnerable to lateral movement attacks. The modernization imperative, therefore, is not just about functionality; it is about bringing these unruly applications into compliance with the Principle of Least Privilege without rewriting their source code—a source code that, in many cases, has been lost or is owned by vendors who no longer support the product.

The "Golden Image" Operational Dead End

The operational manifestation of this legacy architecture is the "Golden Image." In traditional Virtual Desktop Infrastructure (VDI) or physical desktop management (SCCM), administrators create a monolithic master disk image containing the operating system, security agents, and a baseline set of applications.

This image is then cloned to thousands of endpoints. While this model provided consistency in the mid-2000s, it has broken down under the weight of modern complexity.

As different departments require different application sets, the number of Golden Images explodes—a phenomenon known as "Image Sprawl."

An organization might maintain an "HR Image," a "Finance Image," and an "Engineering Image," each requiring separate patching cycles for Windows updates, antivirus definitions, and application upgrades. If a critical vulnerability is discovered in Adobe Reader, for example, the administrator must crack open every single image, patch the application, reseal the image, and redeploy it.

This process is slow, labor-intensive, and prone to human error. It essentially turns the IT department into a digital assembly line that can never stop moving but rarely makes forward progress.

The transition to cloud-native endpoints, such as Azure Virtual Desktop (AVD) and Windows 365, renders this model untenable. In the cloud, the paradigm shifts from "Device Management" to "Service Delivery." The operating system is ephemeral; it may be destroyed and recreated daily or even significantly faster in non-persistent scenarios. Applications that rely on machine-specific state or complex, slow installation routines effectively block the adoption of these agile, scalable cloud platforms. The industry is thus searching for a way to decouple the application from the OS, transforming the application from a "parasitic" installation into a portable, modular asset.

The Catalyst for Change: Cloud PCs

and Azure Virtual Desktop

The impetus for modernizing Windows application packaging is inextricably linked to the rise of Desktop-as-a-Service (DaaS), specifically Microsoft's Azure Virtual Desktop (AVD) and Windows 365 Cloud PCs. These platforms represent a fundamental architectural departure from traditional on-premises VDI, introducing new operating system capabilities and economic models that necessitate a modernized application strategy.

The Multi-Session Revolution

Historically, Windows came in two distinct flavors: Windows Server (which supported multiple concurrent user sessions via Remote Desktop Services) and Windows Client (Windows 10/11, which was single-user).

Users preferred the Client OS for its familiarity and app compatibility, but it was expensive in the cloud because each user required their own dedicated Virtual Machine (VM). Windows Server was cost-effective (many users on one VM) but suffered from poor application compatibility, as many Win32 apps check the OS version and refuse to run on a Server OS.

Azure Virtual Desktop introduced Windows 10 and 11 Enterprise Multi-session, a hybrid OS available only in Azure. It combines the user experience and compatibility of the Client OS with the multi-user scalability of the Server OS. This allows organizations to stack 10, 20, or even 50 users on a single robust VM, dramatically reducing cloud infrastructure costs. However, this density introduces a critical constraint: State Separation.

In a multi-session environment, multiple users share the same C: drive, the same kernel, and the same global registry hives. If a legacy application behaves poorly—for instance, by locking a global configuration file for exclusive write access—it will crash for User B as soon as User A launches it.

If an application consumes excessive CPU cycles during a "self-repair" or "first-run" initialization, it degrades the experience for every other user on that node. Consequently, the "sloppy" behaviors tolerated on physical PCs become showstopping failures in the multi-session cloud. Applications must be strictly isolated from one another and from the underlying OS to function reliably in this shared environment.

The Imperative of Statelessness and Agility

Cloud PCs thrive on "statelessness." In a modern AVD architecture, the Session Host (the VM providing the desktop) is treated as a disposable commodity. To maintain security and stability, these hosts are frequently reimaged or replaced with fresh deployments from a gallery image. In this "cattle, not pets" model, there is no time to run a 45-minute installation script for a heavy CAD application every time a user logs in or a new host spins up.

Legacy installation methods (MSI/EXE) are inherently stateful and slow. They mutate the OS. To achieve the agility required for the cloud—where resources scale up and down dynamically based on demand—applications must be delivered instantly. This requirement has driven the industry away from "installing" applications and toward "attaching" them.

The goal is a composable desktop where the OS, the user profile (via technologies like FSLogix), and the applications are three distinct layers that are assembled in real-time at the moment of user logon.

This architecture minimizes the base image size, eliminates application conflicts, and allows for "zero-downtime" updates, as updating an app simply involves swapping out the attached virtual disk rather than reinstalling software on thousands of machines.

MSIX: The Containerization of the Desktop

To address the twin challenges of WinRot (on physical devices) and state separation (in the cloud), Microsoft introduced MSIX, a unified packaging format designed to supersede the fragmented ecosystem of MSI, EXE, and App-V. MSIX is not merely a new installer wrapper; it is a containerization technology that fundamentally alters the runtime environment of the application.

Architectural Core: The AppxManifest and Virtual Resources

At its heart, an MSIX package is a zipped archive containing the application binaries and a robust XML configuration file named AppxManifest.xml. Unlike an MSI database which contains procedural instructions on how to install (copy file A to path B), the MSIX manifest is declarative. It describes what the application requires—its identity, its capabilities, and its integration points (extensions) with the OS.

When an MSIX application is launched, it does not run directly against the host OS in

the traditional sense. Instead, the OS spins up a lightweight container.

- Virtual File System (VFS): The application's file operations are intercepted. When the app attempts to write to its installation directory (e.g., VFS\ProgramFilesX64\Vendor\App), the OS redirects this request to the package's isolated data store. This ensures that the application can never pollute the host's System32 or Program Files directories. It also means that uninstalling the app is as simple as deleting the container, removing 100% of the files instantly and eliminating WinRot.
- Virtual Registry (VREG): Similarly, registry operations are virtualized. The package contains a registry.dat file that merges with the user's registry view at runtime. The app sees a unified registry, but any keys it creates or modifies are trapped within its private hive. This prevents the "Registry Bloat" that plagues legacy systems.

Security, Identity, and Integrity

MSIX enforces a modern security model that was optional or non-existent in the Win32 world.

- Package Identity: Every MSIX app has a strong identity (Name, Publisher, Version). This allows the OS to manage it as a distinct object rather than a loose collection of files. Windows features like Notifications, Live Tiles, and Background Tasks require this identity to function.
- Code Signing: Perhaps the most significant hurdle for legacy migration is the requirement for digital signing. No MSIX package can be installed unless it is signed with a trusted certificate. This effectively kills the distribution of anonymous or tampered software. Organizations must implement a Public Key Infrastructure (PKI) strategy, using either public certificates (from vendors like DigiCert) or internal enterprise certificates trusted via Group Policy.
- Timestamping: Best practices dictate that all signatures must be timestamped. A digital signature is valid only as long as the certificate is valid. If a certificate expires, the app will stop working unless it was timestamped at the time of signing. A timestamp proves that the certificate was valid when the signing occurred, allowing the signature to remain trusted indefinitely even after the certificate's expiration date.

Network Efficiency and Differential Updates

One of the most compelling advantages of MSIX for cloud and remote scenarios is its

update mechanism. In the MSI world, updating a 1GB application often required downloading a new 1GB installer and running a full uninstallation/reinstallation sequence. MSIX utilizes a block-level differential update engine. The OS maintains a "block map" of the installed application. When an update is published, the OS compares the new package's block map to the existing one and downloads only the 64KB blocks that have changed.

For an application where only a few DLLs or executables have been patched, a 1GB update might result in only 5-10MB of network traffic. In an enterprise with 10,000 endpoints, this reduction in bandwidth consumption is massive, accelerating deployment times and reducing congestion on corporate networks and VPNs.

The "Success Rate" Reality and Limitations

Despite these benefits, the transition to MSIX has been slow and fraught with difficulty. The strict isolation that provides security also breaks compatibility. Industry analysis suggests that only 30% to 60% of legacy Win32 applications work natively when converted to MSIX without modification.

Key Technical Blockers:

- Drivers: MSIX containers run in user mode. They cannot install kernel-mode drivers or filesystem filters. Applications that rely on hardware dongles, specialized VPN clients, or deep antivirus hooks cannot be converted to pure MSIX.
- Services: While Microsoft added support for Windows Services in later builds of Windows 10/11, these services run isolated within the package container. They cannot easily interact with other system services or applications outside the container, breaking complex multi-tier enterprise apps.
- Current Working Directory (CWD): Legacy apps often assume their CWD is their executable's folder and look for config.ini locally. When launched via the MSIX container, the CWD may default to C:\Windows\System32, causing the app to fail to find its configuration files.
- Command Line Arguments: A traditional shortcut might launch app.exe /mode:admin. MSIX shortcuts point to the container entry point, and passing arbitrary arguments was historically restricted (though this is improving, it remains a common friction point).

Remediation Engineering: The Package

Support Framework

To bridge the chasm between the messy reality of legacy applications and the strict rules of the MSIX container, Microsoft released the Package Support Framework (PSF). The PSF is arguably the most critical tool in the modernization arsenal, enabling "fixups" (shims) that allow legacy apps to run in containers without access to the original source code.

The Detours Technology and Injection Mechanism

The PSF leverages Detours, a technology developed by Microsoft Research for instrumenting arbitrary Win32 functions on x86, x64, and ARM machines. In the context of the PSF, Detours is used to perform "API Hooking."

When a PSF-enabled MSIX app launches, it does not run the vendor's executable directly. Instead, the entry point in the AppxManifest.xml is set to PsfLauncher.exe. This launcher bootstraps the environment and injects a specialized DLL (the "Fixup" DLL) into the application's memory space. This injected DLL sits between the application and the Windows OS. When the application calls a Windows API function—such as CreateFile, RegOpenKey, or GetCurrentDirectory—the fixup intercepts the call. It can then modify the parameters, redirect the request to a different location, or handle the error gracefully before passing control back to the application or the OS. This allows the application to "think" it is running in a legacy environment while actually complying with modern rules.

Core Fixups and Configuration

The behavior of the PSF is controlled by a config.json file placed in the package root. This file tells PsfLauncher.exe which application executable to run and which fixups to apply.

File Redirection Fixup (FileRedirectionFixup.dll)

This is the most commonly used fixup. Legacy apps often crash because they try to write logs or update auto-update binaries in their installation folder (e.g., C:\Program Files\VendorApp). In MSIX, the package volume is immutable and read-only.

- Mechanism: The fixup hooks CreateFile and WriteFile. If the app attempts to write to a protected path defined in the config.json, the fixup redirects the write operation to a writable location in the user's profile, typically

C:\Users\%USERNAME%\AppData\Local\Packages\\LocalCache.

- Result: The application succeeds in writing the file (to the new location) and reading it back, completely unaware that the file is not where it expects it to be. This resolves "Access Denied" crashes instantly.

Dynamic Library Fixup (DynamicLibraryFixup.dll)

Legacy apps often fail to load DLLs because they rely on the PATH environment variable or the current directory, which may be obscured by the containerization. This fixup intercepts LoadLibrary calls and explicitly guides the application to the correct path within the package VFS, resolving "DLL Not Found" errors.

Scripting and Pre-Launch Actions

Many enterprise deployments utilize PowerShell scripts to map network drives, check for prerequisites, or clean up old data before an application launches. Native MSIX does not support "Pre-Install" or "Pre-Launch" scripts in the traditional sense.

- PSF Solution: The config.json can be configured with a startScript (runs before the app) and endScript (runs after the app terminates). The PSF Launcher executes these PowerShell scripts within the container context, restoring the automation capabilities that admins relied on with SCCM.

Implementing PSF: The Tooling Gap

Implementing the PSF manually is a developer-centric task requiring knowledge of JSON, XML, and directory structures. It is error-prone and tedious. Consequently, the commercial ecosystem has evolved to automate this. Tools like Advanced Installer and Flexera AdminStudio now feature "PSF Wizards." These tools can analyze an application during the conversion process (trace scanning), detect that it tries to write to its installation folder, and automatically inject the FileRedirectionFixup and generate the correct config.json without the packager needing to write a single line of code. This automation increases the MSIX compatibility success rate from ~30% to over 90% for non-driver-based applications.

Dynamic Delivery: App Attach and the Composability Layer

While MSIX solves the packaging and isolation problem, App Attach solves the delivery and management problem, particularly for Azure Virtual Desktop. It represents the

realization of the "stateless desktop" vision.

Architecture: Decoupling Data from Compute

In a traditional deployment, the application binaries are copied to the C: drive of the VM. In App Attach, the application package (the MSIX expanded content) is stored inside a virtual disk file—originally VHD or VHDX, and more recently CIM (Composite Image Filesystem). These disk files are stored on a central file share (typically Azure Files or Azure NetApp Files) separate from the Session Host VMs.

When a user logs into AVD:

1. The AVD Agent reads the user's entitlements.
2. It locates the VHD/CIM files for the assigned apps on the file share.
3. It mounts (attaches) these disks to the Session Host VM.
4. It registers the app with the OS.
   To the user, the app icon appears on the Start Menu instantly. When they click it, the app launches. However, the app is running from the network share, effectively streaming its execution data. The OS disk remains clean. This allows IT to update an application by simply uploading a new disk image to the share and updating the assignment pointer. The next time the user logs in, they get the new version. No re-imaging, no maintenance windows, and no "uninstall" scripts required.

Performance: The Shift from VHDX to CimFS

The choice of disk format is the single most critical performance factor in App Attach.

- VHD/VHDX: These are monolithic disk image files. Mounting a VHDX requires the OS to read the file header and establish a read/write lock (even if mounted read-only). In a multi-session environment where 20 users might log in within a few minutes (the "boot storm"), mounting dozens of VHDX files causes significant CPU spikes on the Session Host, degrading performance for everyone.
- CimFS (Composite Image Filesystem): Introduced specifically for this use case, CimFS is a collection of files (a .cim metadata file and several data files). It is designed to be a read-only, memory-mapped filesystem.
    - Metrics: Benchmark testing reveals that mounting a CimFS image is drastically faster than VHDX (averaging 255ms vs 356ms) and, more importantly, consumes significantly less CPU and RAM.
    - Recommendation: For all modern AVD deployments (Windows 11), CimFS

is the mandatory standard. It enables higher user density per host, directly reducing Azure compute costs.

Storage Fabric and RBAC

The reliance on network-attached storage means the file share performance is paramount.

- Latency: The storage account must reside in the same Azure Region as the Session Hosts. Separation (e.g., Storage in West US, VMs in East US) will result in unusable application latency.
- Permissions: App Attach utilizes the machine identity for mounting. The Session Host computer accounts must have Reader access to the file share. In Azure Files, this is achieved by assigning the Storage File Data SMB Share Reader RBAC role to the AVD Service Principal and the Session Host computer group. This differs from traditional file shares where permissions are typically User-centric.

Evolution: MSIX App Attach vs. App Attach (v2)

A nuanced but important shift is occurring in Microsoft's terminology and architecture.

- MSIX App Attach (Classic): This referred specifically to the mechanism of delivering MSIX packages via VHD.
- App Attach (Preview/v2): Microsoft is broadening the scope. The new "App Attach" feature in the Azure portal is designed to be a universal container delivery platform. It supports not just MSIX, but also App-V packages and potentially other partner formats (like Liquidware) directly. This signals Microsoft's recognition that while MSIX is the future, the legacy tail (App-V) is long, and a unified delivery pane is required.

The Hybrid Alternative: Layering and Liquidware FlexApp

Despite the power of MSIX and PSF, a subset of the application estate—often the most critical and complex 20%—remains stubborn. Applications with kernel-mode drivers, deeply embedded middleware, or complex cross-process dependencies often fail even with extensive PSF remediation. For these, Application Layering offers a robust alternative.

Filter Drivers vs. Containers

While MSIX uses containerization (user-mode isolation), Liquidware FlexApp uses filesystem and registry filter drivers.

- Mechanism: FlexApp injects the application into the OS at a lower level. When the OS looks at the file system, the FlexApp filter driver "merges" the application layer with the base OS view. The OS "sees" the files as if they were natively installed on C:, including drivers and services.
- Compatibility: Because it operates at the kernel filter level rather than a user-mode container, FlexApp boasts a compatibility rate exceeding 90%. It can successfully deliver applications that MSIX rejects, such as those requiring specialized device drivers or complex COM+ interactions.

Strategic Integration

Liquidware has integrated FlexApp directly with the AVD App Attach control plane. This allows administrators to use the native AVD interface to assign FlexApp packages alongside MSIX packages.

- The Hybrid Strategy: The most mature modernization strategy is not "MSIX or nothing." It is a tiered approach:
    1. Tier 1 (Native MSIX): Modern apps, simple tools, and in-house developed software. Delivered via App Attach.
    2. Tier 2 (Remediated MSIX): Legacy Win32 apps fixed with PSF. Delivered via App Attach.
    3. Tier 3 (FlexApp Layering): Complex legacy apps, apps with drivers, and remaining App-V packages. Delivered via FlexApp (integrated into App Attach).
    4. Tier 4 (Image Based): Security agents and heavy compilers that must be in the base image.
       This hybrid model ensures 100% coverage of the estate while maximizing the benefits of modernization.

The Modernization Toolchain: Vendor Analysis

The transformation of thousands of applications cannot be achieved manually. A specialized ecosystem of vendors has emerged to automate the discovery, conversion,

testing, and management of this process.

Table: Vendor Capabilities Comparison

Vendor	Primary Focus	Key Capability	Best For...
Flexera (AdminStudio)	Portfolio Management	Package Feed Module: Auto-downloads and wraps thousands of public vendor apps.	Large Enterprises (1000+ apps) needing bulk automation and inventory tracking.
Advanced Installer	Packaging Engineering	PSF Integration: Best GUI for visualizing and injecting PSF fixups. Powerful MSIX Editor.	Packaging Engineers requiring granular control and debugging of failed conversions.
Rimo3	Automated Testing	Intelligent Smoke Testing: Automates the Install/Launch/Test cycle on AVD images.	QA Teams and Architects validating OS migrations (e.g., Win10 -> Win11).
Liquidware	Application Delivery	FlexApp Layering: High compatibility layering for complex apps.	Hybrid environments needing to support difficult apps that fail MSIX conversion.

Juriba	Migration Logistics	App Readiness: Workflow orchestration, scheduling, and user communication.	Project Managers overseeing the end-to-end logistics of an OS or Cloud migration.
Master Packager	Technical Editing	Table Editor: Lightweight, fast, low-cost MSI/MSIX table manipulation.	DevOps and technical packagers wanting a "no-nonsense" editor without the bloat.
Access IT Automation	API Automation	Capture: API-driven packaging factory for CI/CD pipelines.	DevOps-centric organizations integrating packaging into software build pipelines.

Vendor Spotlights

Rimo3: The Validation Engine

Migration projects often stall at the "User Acceptance Testing" (UAT) phase. Manually testing 500 apps on a new Windows 11 build takes months. Rimo3 solves this by automating the validation. It ingests an application (MSI, EXE, or MSIX), spins up a headless AVD Session Host (configured with the target OS), installs the app, and performs a "smoke test" (launching the app, checking for crashes, verifying UI rendering). It provides a "Pass/Fail" report and performance metrics. This allows IT to know before migration which apps will break, shifting the discovery of technical debt to the left.

Advanced Installer: The Fixup Factory

When Rimo3 reports a failure, Advanced Installer is the remediation workbench. Its deep integration with the Package Support Framework allows engineers to visually trace the application's behavior. If the trace shows the app trying to read a config file

from the wrong directory, the engineer can right-click and "Add File Redirection Fixup," and the tool handles the complex JSON and DLL injection. This lowers the barrier to entry for MSIX, allowing admins without C++ skills to leverage advanced compatibility shims.

Flexera AdminStudio: The Enterprise Library

For organizations managing massive portfolios, AdminStudio acts as the central repository. Its "Backlog" management allows teams to prioritize applications based on usage data (integration with SCCM/Intune). Its "Package Feed Module" is a standout feature, providing verified silent installation commands for thousands of common applications (Adobe, Chrome, Zoom), dramatically reducing the research time for packaging teams.

Juriba: The Command Center

While others focus on the package, Juriba focuses on the project. It creates a "single pane of glass" for the migration status. It tracks which apps are ready, which users use them, and automates the communication to those users (e.g., sending an email: "Your apps are ready, your migration is scheduled for Tuesday"). This orchestration is vital for avoiding the chaos of large-scale rollouts.

Strategic Recommendations and Conclusion

The modernization of Windows application packaging is not a mere technical upgrade; it is a strategic necessity for the adoption of cloud-native computing. The legacy model of monolithic images and fragile, stateful installations is incompatible with the agility, security, and statelessness required by Azure Virtual Desktop and Windows 365.

To navigate this transformation successfully, organizations must adopt a disciplined, data-driven approach:

1. Rationalize Before You Modernize: Do not migrate "junk." Use inventory data to identify and retire unused applications. The fastest way to package an app is to delete it.
2. Adopt a "Factory" Mindset: Move away from ad-hoc manual packaging. Implement a toolchain that includes automated testing (Rimo3) and standardized conversion workflows (AdminStudio/Advanced Installer).
3. Prioritize MSIX and CIM: Make MSIX the default standard for all new packaging.

Enforce the use of CimFS for all App Attach payloads to maximize AVD performance and user density.

4. Master the PSF: Invest in training or tooling that unlocks the Package Support Framework. It is the key to salvaging the investment in legacy Win32 software.
5. Embrace Hybrid Delivery: Accept that 100% MSIX conversion is a theoretical ideal, not a pragmatic reality. Integrate a layering solution like Liquidware FlexApp to handle the complex edge cases without stalling the broader migration.

By decoupling the application from the operating system, organizations do not just solve the immediate problems of WinRot and compatibility; they future-proof their estate, enabling a workspace that is secure, portable, and ready for whatever platform follows Windows 11.