RAVEN

ADR VS EDR

EDR wasn't built for servers.

ADR was.

A full comparison brochure on why endpoint-first security fails to protect modern application runtimes

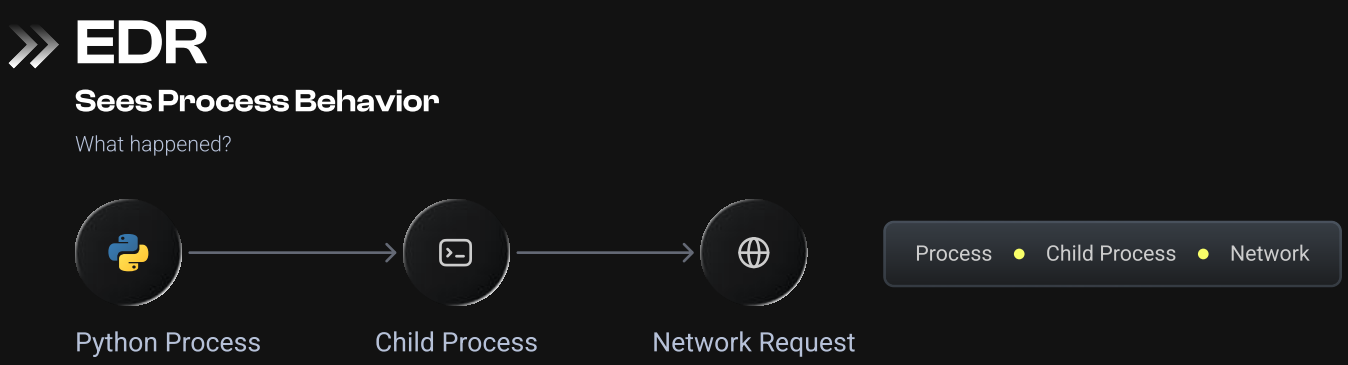
1. Executive Summary

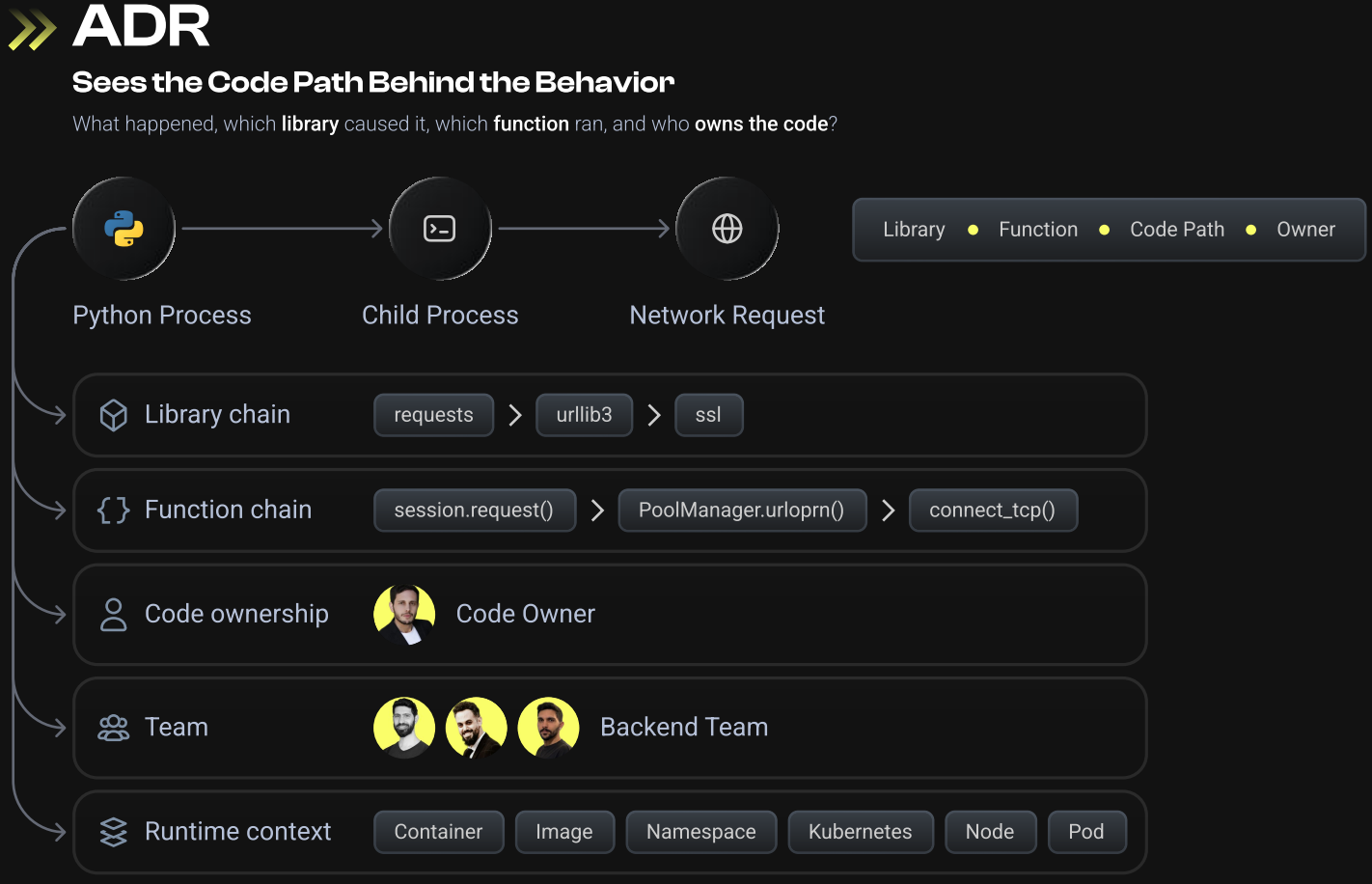
Traditional EDR and CWPP runtime sensors are valuable for infrastructure and workload security. They can observe process execution, network activity, file activity, system calls, and container/workload context. This is important, but it is not the same as application-runtime visibility.

ADR also sees process, network, syscall, container, image, pod, node, and workload context. The difference is that ADR ties these runtime events back to the exact application runtime, code, library, function, and call chain that caused the behavior.

This distinction matters most for CISOs responsible for software factories: organizations that build, ship, and operate code at high velocity. For those organizations, the security question is not only "what did the workload do?" but "which part of our application or open-source dependency caused it, is it expected, and what should the developer fix?"

EDR / CWPP sees the process. ADR sees the application runtime and connects process/network/syscall activity to the specific library/function chain responsible for it.





2. The Core Difference: EDR Runtime View vs. ADR Runtime View

A traditional EDR/CWPP runtime sensor may correctly detect that a process created a child process, executed a command, opened a network connection, or performed a syscall. ADR detects these same categories of runtime behavior as well. The core difference is attribution and depth.

ADR connects the runtime event back to the exact library, function path, syscall, container, image, pod, service, and policy context. This provides a direct root cause for both security teams and developers.

Category	Traditional EDR/CWPP Runtime View	Runtime ADR
Main Question Answered	✗ What did the process, host, or workload do?	✓ Which library/function chain inside the application caused the behavior?
Process, Network, And Syscall Visibility	✗ Detects process execution, network activity, file activity, and system calls at the workload/host level.	✓ Also sees process, network, syscall, container, image, pod, node, and workload context - and ties those events back to the responsible library/function chain.
Application-Runtime Visibility	✗ Does not provide deep language-aware visibility inside the application runtime; typically cannot identify the specific Python/Java/Node/Go/.NET library and function chain that caused the event.	✓ Provides runtime code visibility and application-runtime visibility at the library, function, syscall, and call-chain level across 10+ languages.
Open-Source Library Context	✗ May know that an image or workload contains vulnerable packages, but runtime event attribution to a specific library/function chain is limited.	✓ Shows the specific OSS package/library, version, runtime path, and vulnerable function context when applicable.
Command Execution	✗ Detects process creation or command execution.	✓ Attributes command execution to the responsible library/function chain and policy context.
Network Activity	✗ Detects outbound connections or suspicious network behavior.	✓ Attributes network behavior to the specific library/function chain that initiated it.
Vulnerability Context	✗ Usually package, image, or workload-level context.	✓ Runtime SCA context: present on disk, loaded into memory, executed by CPU, and vulnerable function executed.
False-Positive Reduction	✗ Can be ambiguous because legitimate application-runtime behavior may look suspicious at the process level.	✓ Reduces ambiguity by showing exactly which library caused the event and whether it is expected or exploit-driven.
Prevention Model	✗ Often broader process, workload, host, file, or network policy action.	✓ Surgical syscall-level prevention from the violating library/function chain, without killing the entire application process.
Developer Actionability	✗ May require manual investigation from process-level evidence.	✓ Points to package, library, function, image, service, policy, and fix path.

3. Runtime Code Visibility Across 10+ Languages

The difference is not that ADR merely recognizes a process named python, java, node, or dotnet. A traditional runtime sensor can often see a runtime process at the OS level. ADR goes deeper: it maps runtime activity back to application libraries and functions across major language runtimes.

Supported runtimes include Java, Node.js, Python, Go, .NET, C/C++, Ruby, PHP, Kotlin, and Scala.

This language-aware runtime visibility is what makes ADR useful for security leaders responsible for software development organizations. ADR connects infrastructure events to code ownership and developer remediation, not only to workload investigation.

4. Why Granularity Matters: Legitimate Library Behavior vs. Exploit-Driven Behavior

Some runtime behaviors look suspicious from the outside but are legitimate when viewed with application context. For example, a Python library may run a feature check at the CPU level, such as checking SVE support, and trigger fork/exec behavior as part of legitimate runtime behavior. A regular EDR may alert on the process activity and create a long SOC investigation because it does not know which library and function path caused it.

ADR can shorten that investigation by showing the runtime chain that caused the action. Instead of asking analysts to manually reverse-engineer the process tree, ADR can show the relevant library/function chain and whether the behavior matches expected library behavior or exploit-driven misuse.

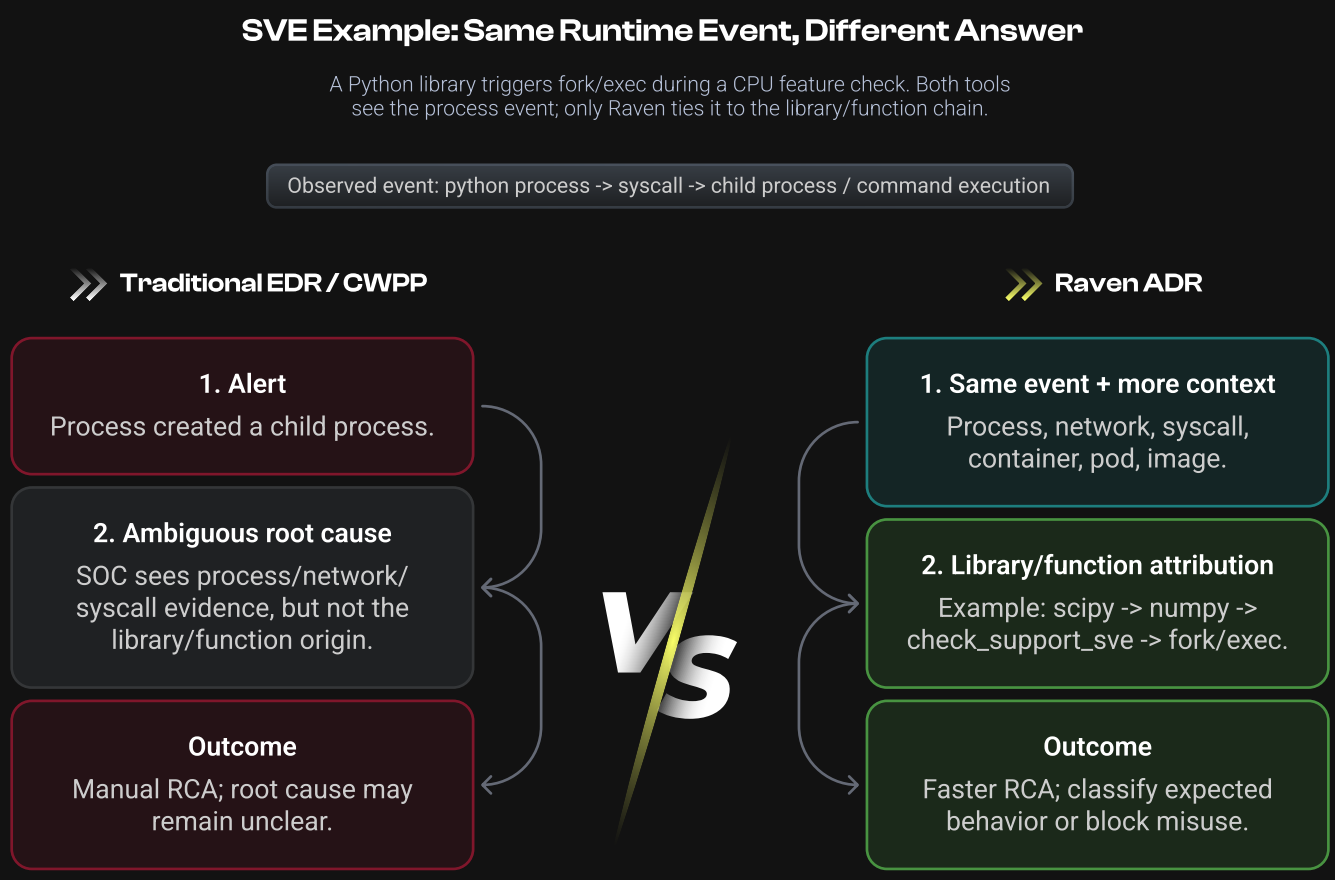


Figure 1 - Visual example: traditional process-level investigation vs. ADR library/function-level root cause attribution.

5. Visual Example: EDR View vs. ADR View

The following example illustrates the difference between a process/workload-centric EDR view and an application-runtime ADR view.

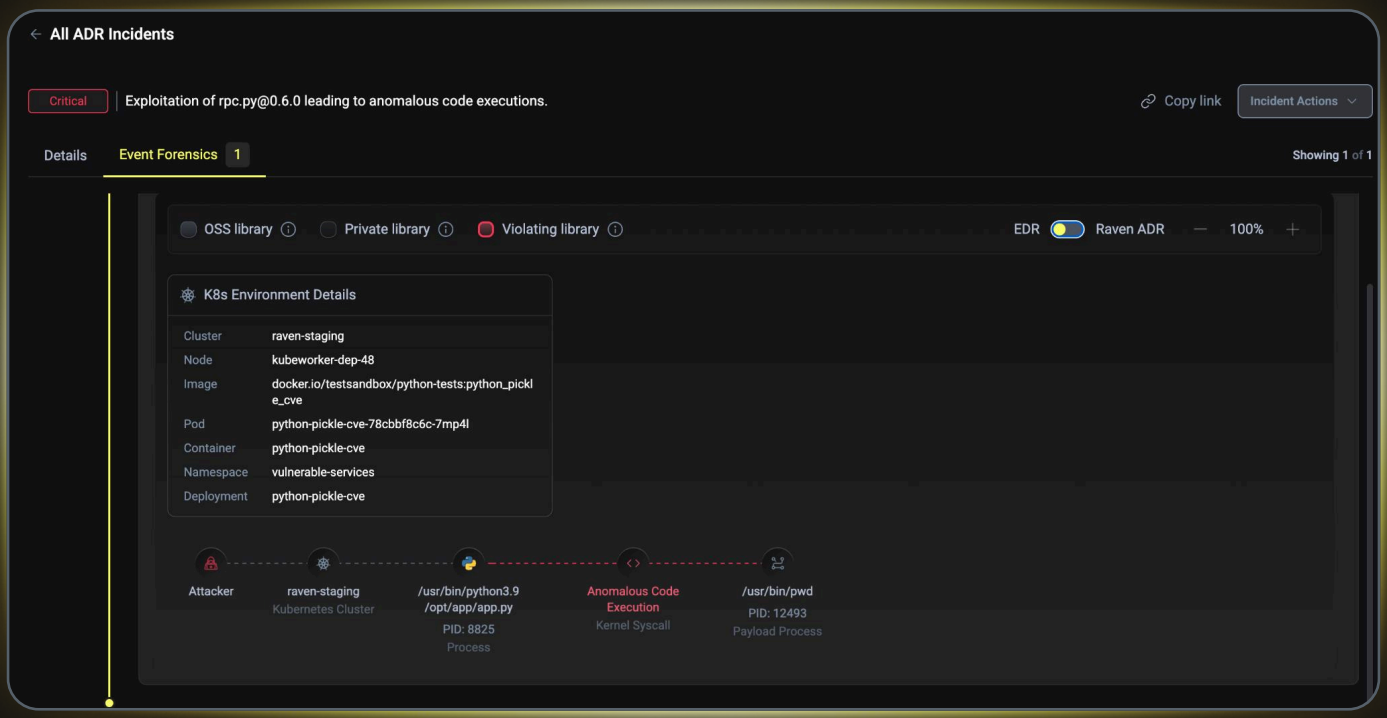


Figure 2 - Traditional EDR-style view: useful process/workload evidence, but limited application-runtime root cause attribution.

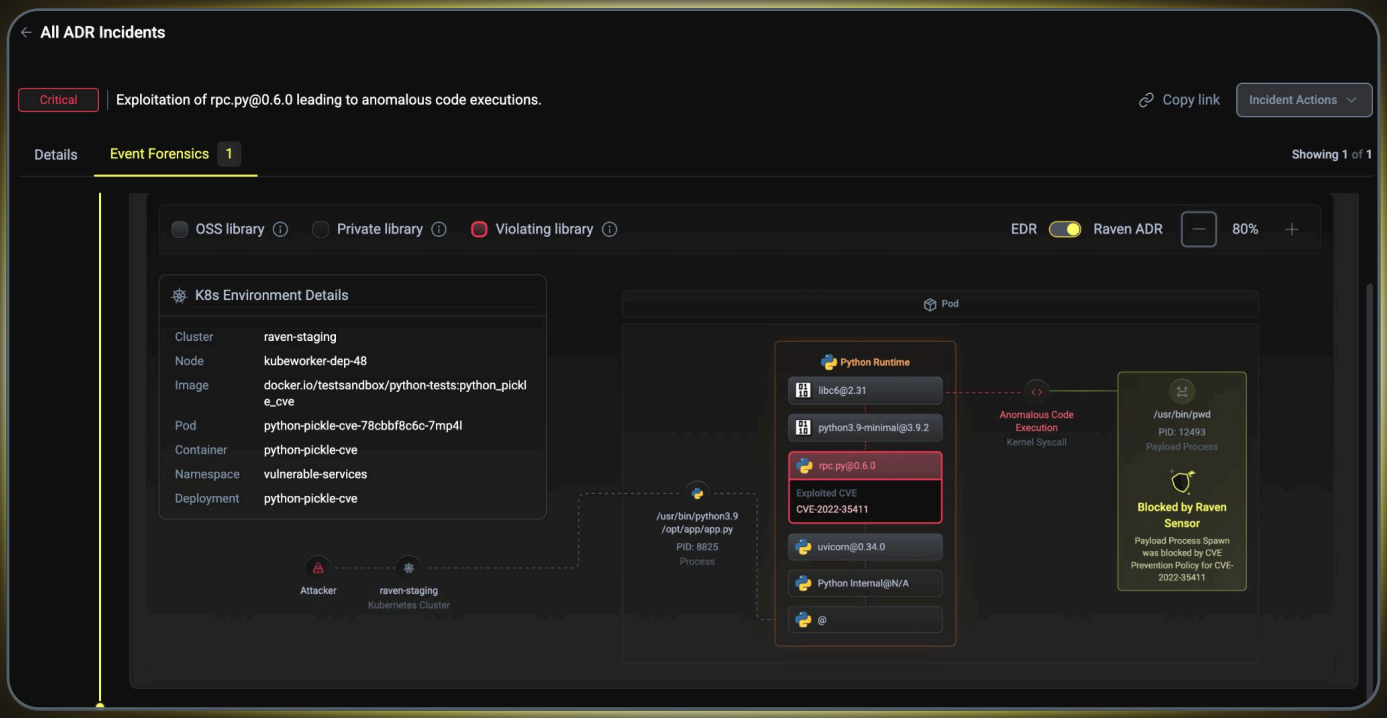


Figure 3 - ADR view: library/package, CVE, runtime path, syscall, payload process, Kubernetes context, and prevention result.

6. Runtime SCA: From Static Inventory to Runtime Truth

Traditional SCA and CNAPP vulnerability modules usually begin with static evidence: packages found in source code, manifests, container images, SBOMs, or workload inventory. That is useful, but it does not prove whether vulnerable code actually runs in production.

Runtime SCA adds runtime evidence to vulnerability prioritization. It can distinguish between vulnerable components that are merely present and vulnerable components that are loaded, executed, or reaching the vulnerable function.

Runtime SCA Capability	Traditional SCA / CNAPP Vulnerability View	Runtime SCA
Detect Vulnerable Packages	✗ Yes	✓ Yes
Identify Package In Image	✗ Yes	✓ Yes
Know Whether Library Is Loaded At Runtime	✗ No	✓ Yes
Know Whether Library Code Executed	✗ No	✓ Yes
Know Whether Vulnerable Function Executed	✗ No	✓ Yes
Prioritize By Actual Runtime Exposure	✗ No	✓ Yes
Connect Vulnerability To Runtime Behavior	✗ No	✓ Yes
Connect Runtime Event To Vulnerable Library/Function	✗ No	✓ Yes

Practical difference: traditional SCA tells the customer what vulnerable packages exist. Runtime SCA tells the customer which vulnerable libraries and vulnerable functions actually execute in production.

7. ADR + Runtime SCA Together

The advantage of ADR combined with Runtime SCA is the connection between runtime detection, runtime prevention, and runtime vulnerability prioritization.

1. ADR detects and prevents abnormal application behavior at the library/function/call-chain level.
2. Runtime SCA prioritizes vulnerabilities based on runtime evidence: disk, memory, CPU execution, and vulnerable-function execution.
3. Library-level ADR prevention blocks the specific dangerous syscall or runtime action caused by a violating library/function chain.
4. ADR gives developers remediation context: package, version, function, image, service, workload, and runtime behavior.

8. Runtime Gatekeeper and CI/CD Control

ADR can also extend runtime intelligence into the software delivery pipeline through Runtime Gatekeeper and CI/CD integration. This is important for software factories because prevention should not only happen after deployment; runtime knowledge should also influence what is allowed to move toward production.

A CI/CD integration can automate the transmission of image build metadata from image build pipelines into the runtime security platform. The CI step typically runs after docker build and before docker push, while the image is still available inside the CI pipeline. This allows runtime intelligence to be associated with build context without collecting sensitive CI data such as passwords.

Gatekeeper / CI Area	ADR Capability
Runtime Evidence	Uses production/runtime observations to understand which libraries and functions actually execute.
CI Image Metadata	Associates image build context with runtime findings and runtime intelligence.
Policy Control	Enables policy-driven decisions in the pipeline based on runtime risk, vulnerable-function context, and customer-defined rules.
Developer Workflow	Routes actionable findings to the teams that own the image, package, service, or dependency.
Security Outcome	Helps prevent risky images and dependencies from progressing without forcing teams to treat every static CVE as equal.

9. Prevention: Broad Blocking vs. Surgical Blocking

Traditional EDR/CWPP prevention often operates at broader levels, such as process, workload, network, file, malware, or host policies. ADR prevention is more surgical. ADR can block the specific syscall-level action originating from the violating library/function chain, without killing the entire process or stopping the application.

Scenario	Traditional EDR/CWPP Response	ADR Response
Python Process Attempts Child Process Creation	✗ Alert on suspicious process spawn; may block or kill process depending on policy.	✓ Identify the library/function chain that caused the syscall and block only the dangerous syscall.
Library Unexpectedly Opens Network Connection	✗ Alert on outbound connection.	✓ Attribute the connection to the specific library/function chain and block that action if policy requires.
Vulnerable Package Exists In Image	✗ Flag vulnerability.	✓ Determine whether the package is loaded, executed, and whether the vulnerable function is reached.
Suspicious Runtime Behavior Occurs	✗ Show process/workload context.	✓ Show process, library, function, syscall, container, image, pod, service, and policy context.

10. Buyer Fit: Infrastructure Security vs. Software Factory Security

Traditional EDR, CWPP, and CNAPP runtime sensors are strong infrastructure and cloud workload tools. They are useful for infrastructure security teams, cloud security teams, and SOC teams that need broad host, container, process, network, and cloud context.

ADR is designed for organizations where the CISO is responsible for defending a software factory: a large engineering organization that builds applications, ships containers, uses open-source dependencies, and operates high-velocity CI/CD pipelines.

Dimension	Traditional EDR/CWPP/CNAPP	ADR + Runtime SCA
Primary Owner	Infrastructure security, cloud security, SOC	CISO/Soc/AppSec/DevSecOps teams protecting applications and software delivery
Primary Object Protected	Host, workload, container, cloud asset	Application runtime, libraries, functions, dependencies, and code execution paths
Best Use Case	Broad infrastructure detection and workload response	Runtime exploit prevention, Runtime SCA, library-level root cause, developer remediation, CI Gatekeeper
Question Answered	Is this workload or process behaving suspiciously?	Which library/function caused it, is it expected, and what should we block or fix?

11. ADR vs. EDR: The General Runtime Security Gap

Traditional EDR/CWPP and cloud workload runtime sensors are strong at broad infrastructure and workload context. They commonly provide visibility into runtime activity such as processes, network connections, file activity, system calls, containers, hosts, and cloud assets. That view is useful, but it usually remains centered on the workload rather than the application runtime inside the workload.

The ADR differentiation is not simply process or network visibility. EDR already sees many of those signals. The differentiation is application-runtime code visibility: library, function, syscall, call-chain attribution, Runtime SCA, vulnerable-function execution, surgical prevention, and CI/CD policy control.

Layer	Traditional EDR/CWPP Runtime View	ADR Runtime View
Infrastructure/Workload Context	Strong cloud workload, process, network, file, syscall, and cloud graph context.	Also sees workload, process, network, syscall, Kubernetes, image, pod, node, and service context.
Application-Runtime Attribution	Not positioned as deep language-aware library/function/call-chain visibility inside the application runtime.	Core capability: maps runtime behavior to library, function, syscall, and call chain across 10+ languages.
Runtime SCA	Can validate some runtime exposure, such as loaded vulnerabilities, according to public traditional EDR/CWPP materials.	Adds disk/memory/CPU/vulnerable-function execution context and ties runtime events back to the vulnerable library/function.
Prevention	Runtime blocking and response policies at workload/security-rule level.	Surgical syscall-level prevention from the specific violating library/function chain.
CI/CD Control	Cloud security and container lifecycle context.	Runtime Gatekeeper and CI integration use image build metadata and runtime intelligence to control pipeline policy decisions.

12. Summary

Traditional EDR and CWPP runtime sensors provide important workload, process, network, and cloud context. These tools are valuable, especially for infrastructure and SOC workflows.

ADR fills a different and deeper gap: application-runtime security. ADR connects runtime behavior to the exact library and function path that caused it, prioritizes vulnerabilities based on runtime execution, and can surgically block only the dangerous syscall or action from the violating library/function chain.