A Modern View on Game Engine Architecture

1. Introduction

For more than three decades, game engines have evolved alongside the games they power. What began as reusable rendering code and utility libraries has gradually grown into increasingly sophisticated platforms containing rendering, physics, animation, scripting, networking, audio, user interfaces, editors and complete development ecosystems.

Despite tremendous advances in hardware and software engineering, the underlying architectural model has remained remarkably consistent. Most modern engines are still organized around collections of tightly integrated subsystems, where functionality is added by extending an already existing core. While this approach has proven highly successful, it also reflects assumptions that originated in a very different era of game development.

Today's projects differ fundamentally from those assumptions. Teams have become larger and increasingly multidisciplinary. Development spans multiple platforms, continuous integration pipelines, live service deployments and distributed tooling. Features are expected to evolve independently, while content creation, gameplay programming and engine development increasingly overlap. As engines continue to grow, complexity has shifted from implementing functionality to managing it.

This paper illustrates the next evolutionary step in game engine architecture is not the addition of new engine features, but a reconsideration of the engine's role itself. Rather than viewing the engine as a collection of rendering, physics, audio and gameplay systems, it is proposed to view it as the runtime environment in which those systems execute. In this model, the engine assumes responsibilities traditionally associated with an operating system: resource management, scheduling, capability management, communication, package loading and execution orchestration, while functionality itself is provided through independent runtime modules.

This complements the previously presented Capability-Oriented Software Architecture and Data-Driven Development Environment by focusing on the runtime that connects both worlds. Rather than redefining those concepts, this model illustrates why such an architectural separation has become increasingly relevant for modern game development and how it addresses challenges that traditional engine architectures were never designed to solve

2. Historical Retrospective

Game engines did not emerge from a predefined architectural vision. They evolved as a practical solution to software reuse. Early computer games were almost entirely monolithic applications. Rendering, input, physics, game logic and platform-specific code were developed together as a single executable because every title was effectively an isolated product. As development costs increased and sequels became common, developers naturally began extracting reusable functionality into shared code bases. The first game engines were therefore not

designed as products of their own; they were simply the intersection of multiple games.

As technology advanced, this reusable foundation expanded continuously. Rendering engines gained animation systems, physics engines, audio systems, networking, scripting environments and eventually complete editor frameworks. The engine gradually became a comprehensive software platform intended to solve as many common development problems as possible.

This evolution was entirely logical. Hardware limitations demanded efficient native implementations, small development teams benefited from centralized code ownership, and shipping a complete engine together with the game simplified deployment and maintenance. The architecture successfully optimized for the challenges of its time.

However, every evolutionary step also introduced additional coupling. Systems that originally existed as independent concerns became increasingly interconnected. New functionality was commonly implemented by extending the engine itself, making the runtime progressively larger and more difficult to evolve. Plugin systems improved extensibility, while data-oriented approaches such as Entity Component Systems addressed many performance limitations, yet the engine itself largely remained a monolithic collection of subsystems.

The fundamental question therefore changes. Modern game development no longer asks how additional functionality can be integrated into an engine. Instead, it asks whether the engine should remain the primary location where functionality resides at all. This observation forms the foundation for the architecture illustrated through this paper

3. The Next Evolutionary Step in Architecture

The architectural model illustrated does not seek to replace established concepts such as data-oriented design, entity component systems or modular software development. Instead, it builds upon these principles by reconsidering the role of the game engine itself. Traditional engines are typically characterized by the functionality they provide. Rendering, physics, animation, audio, scripting and user interface systems are regarded as integral parts of the engine, while individual games extend this foundation with project-specific code. Consequently, the engine becomes the central location where reusable infrastructure and application-specific functionality gradually accumulate.

This architectural model has proven highly successful for several decades. However, as projects continue to increase in scale and complexity, the responsibilities assigned to the engine have expanded accordingly. Modern engines are expected to provide not only rendering or physics, but also asset processing, package management, editor integration, networking, build automation, live services and increasingly sophisticated development workflows. As a result, the primary challenge is no longer the implementation of individual systems, but the management of the growing complexity created by their interaction.

An alternative architectural perspective is to separate runtime responsibilities from functional implementation. Rather than treating the engine as the implementation of

game functionality, the engine is instead defined as the runtime environment responsible for executing that functionality. Similar to an operating system, the runtime provides services such as resource management, scheduling, communication, package loading and capability management, while the implementation of gameplay and engine features resides in independently deployable runtime modules. This distinction fundamentally changes the responsibility of the engine.

Instead of deciding which systems belong inside the engine, the more fundamental question becomes which services must the runtime provide to allow systems to execute efficiently, safely and independently. Although this distinction appears subtle, its architectural consequences are significant. Rendering, physics, artificial intelligence, gameplay systems and editor extensions no longer represent special cases within the engine itself. Instead, they become independent software components operating under a common execution model and interacting exclusively through explicitly defined capabilities.

This separation also changes the relationship between software and content. The runtime itself contains no application-specific behavior. Instead, behavior emerges from the composition of runtime modules, while content determines how these modules are configured, connected and executed. Applications are therefore no longer created by extending a monolithic engine, but by composing independent software components within a shared runtime environment.

Such an architecture naturally aligns with modern software engineering practices. Independent versioning, package-based distribution, continuous integration, platform abstraction and feature isolation become architectural properties rather than development conventions. Equally important, the runtime remains free to optimize execution strategies, scheduling policies and resource management independently of the implementation of individual modules, allowing the execution model to evolve without requiring modifications to application code.

The architecture should therefore not be understood as an alternative to existing engine technologies, but as a reconsideration of their responsibilities. Data-oriented execution, modular software and content-driven development remain fundamental concepts; the difference lies in the role of the engine itself. Rather than acting as the primary implementation of functionality, the engine becomes the runtime responsible for coordinating, executing and managing independently developed software components

4. The Runtime Architecture

The architectural model is founded on a simple observation: a game engine should no longer be viewed as the implementation of a game, but as the runtime in which the game executes. This distinction fundamentally changes the responsibilities assigned to the engine. Rather than implementing application-specific functionality, the runtime becomes responsible for managing execution, communication and resources, while functional behavior is provided by independently developed software modules.

From this perspective, the relationship between the runtime and its modules closely resembles the relationship between an operating system and user applications. The operating system does not implement the functionality of an application; it

provides the services required for applications to execute safely and efficiently. Likewise, the runtime provides the infrastructure required to execute game-specific functionality without embedding that functionality directly into the engine itself.

Consequently, the runtime consists of a small set of responsibilities:

- Package discovery and loading
- Capability registration and dependency resolution
- Resource and memory management
- System scheduling and execution
- Communication between independent modules
- Validation of compatibility, integrity and runtime constraints

Every other aspect of the application, including gameplay systems, rendering pipelines, physics simulations, user interfaces or editor extensions, is treated as software executing within the runtime rather than software defining the runtime. This separation significantly reduces the conceptual complexity of the engine. The runtime remains responsible for how software executes, while modules define what is executed

Host Runtime

The host represents the central execution environment of the application. It owns the lifetime of all runtime services and provides the execution context in which modules operate. Unlike traditional engines, the host contains only functionality that is fundamentally required by every application. Services such as scheduling, memory allocation, asset management, package loading, capability discovery and communication exist because they are necessary to coordinate execution, not because they implement gameplay.

This distinction allows application behavior to evolve independently from the runtime itself. New functionality is introduced by composing additional modules rather than extending the host

Runtime Modules

Modules encapsulate behavior. Each module represents an independently deployable software component exposing functionality through explicitly declared capabilities. The runtime neither distinguishes between gameplay systems and engine systems nor assigns architectural significance to either. Both participate in the same execution model and communicate through identical mechanisms.

As a result, rendering, physics, artificial intelligence or gameplay systems become architectural peers rather than privileged engine subsystems. This uniformity simplifies both composition and maintenance while enabling functionality to evolve independently throughout the lifetime of the application

Communication

Communication between modules follows the principles introduced by the capability-oriented software architecture. Rather than exposing internal implementations, modules communicate exclusively through explicitly defined capabilities projected by the runtime. This approach eliminates compile-time coupling

between independently developed components while allowing direct native invocation through statically resolved function pointers.

Unlike reflection-based or interface-driven systems, no runtime discovery or dynamic dispatch is required once a capability has been projected. Module boundaries therefore become largely transparent during execution, allowing communication to remain predictable while preserving implementation independence. The runtime consequently assumes responsibility for establishing communication, whereas participating modules remain entirely unaware of each other's internal implementation

Package Loading

Application functionality is introduced through packages discovered and loaded by the runtime. Each package contains one or more runtime modules together with the metadata required for dependency resolution, capability registration and compatibility validation. Rather than linking applications during compilation, composition occurs dynamically as packages become available to the runtime.

This process enables applications to evolve through composition rather than modification while preserving a deterministic execution model

Trust and Validation

The introduction of independently deployable modules naturally raises questions regarding integrity and trust. Within the presented architecture, package loading and trust validation are treated as separate concerns.

The runtime is responsible for discovering and loading packages, while validation policies determine whether a package should be accepted for execution. Such policies may include compatibility verification, digital signatures, package manifests, cryptographic integrity checks, capability validation or optional package encryption. This separation closely mirrors the relationship between executable loaders and code-signing mechanisms employed by modern operating systems. The loader establishes execution, whereas trust is established through independent verification mechanisms.

Consequently, modularity does not inherently reduce application security. Instead, the architecture provides explicit integration points through which trust policies can be implemented without affecting the execution model itself

5. Rethinking Design – Challenges and Solutions

The transition from monolithic engines towards independently deployable runtime modules inevitably raises several architectural questions. While modularity improves maintainability, scalability and separation of concerns, it also introduces concerns regarding performance, communication and execution efficiency. This chapter discusses these considerations and the design decisions adopted to address them

5.1 Module Boundaries and Performance

Perhaps the most common criticism of modular software architectures concerns execution performance. Separating functionality into independently compiled

modules naturally prevents aggressive compiler optimizations such as cross-module inlining. Furthermore, communication between modules is performed through indirect function calls rather than direct compile-time references. From a purely theoretical perspective, these observations are correct. A module boundary inevitably introduces an additional level of indirection compared to a completely monolithic executable.

The practical impact, however, depends far less on the existence of module boundaries than on where those boundaries occur during execution. Traditional object-oriented software frequently performs virtual dispatch as part of the innermost execution loop, causing indirect calls to occur repeatedly during entity processing. The architecture presented in this paper intentionally avoids this execution model.

Instead, module boundaries exist primarily at system granularity. Runtime services invoke systems through projected capabilities, after which systems execute directly on contiguous component data without further interaction across module boundaries. Consequently, the cost of crossing a module boundary is amortized over the complete execution of a system rather than every processed entity. Performance therefore becomes primarily a function of data locality and execution strategy rather than module composition itself

5.2 Sparse-Based Entity Component Systems

The execution model described above relies on the sparse-based Entity Component System introduced in the accompanying ECS paper. Unlike archetype-oriented approaches, the sparse-based model stores every component type in its own dedicated storage. Components remain completely independent from one another, avoiding artificial memory growth caused by infrequently used components while preserving maximum flexibility during composition. Execution is organized around systems operating on owned groups.

During execution, the runtime resolves the required component storages before entering the processing loop. Systems subsequently iterate linearly over contiguous component arrays. Conceptually, execution follows a pure linear access model rather than repeatedly crossing runtime boundaries for every processed entity.

As a consequence, the module boundary exists outside the performance-critical portion of the execution pipeline. Once a system begins iterating over component arrays, execution proceeds entirely on contiguous memory without additional capability resolution or runtime dispatch. This execution model allows the runtime architecture to preserve modularity while maintaining the cache-efficient processing characteristics expected from modern data-oriented engines

5.3 Runtime Component Aggregation

Although systems execute directly on contiguous component arrays, the runtime must first establish a common representation of component types across independently compiled modules. Unlike managed environments capable of sharing runtime type information, native C++ modules cannot directly reference template types originating from unrelated compilation units. Consequently, component access

cannot rely on template parameters alone. To address this limitation, every module registers its component definitions during package loading. The runtime aggregates these definitions into a global component registry that associates a deterministic component identifier with its corresponding storage.

Rather than relying on compiler-generated type information, component identity is derived from the hashed component name. This produces a deterministic identifier that remains stable across independently compiled modules while avoiding compile-time coupling. Each registry entry references the underlying component storage together with the metadata required to construct runtime views over the corresponding memory.

As a result, independently developed modules may reference identical component definitions without requiring shared template instantiations or common compilation units. Once the appropriate storage has been resolved, systems operate directly on native memory without further registry interaction.

The same architectural principle already governs capability discovery within the SDK architecture. Capabilities map deterministic identifiers to executable function pointers, while the runtime component registry maps deterministic identifiers to component storage. Software and data therefore become integrated through identical runtime mechanisms despite representing fundamentally different concepts

5.4 Optimizing Partial Groups

Owned groups represent the optimal execution path within the sparse-based ECS because all participating component arrays share identical ordering. Systems therefore iterate over contiguous memory without requiring additional lookups. Partial groups introduce a different challenge. They combine owned component arrays with component types that reside outside the ownership boundary. While this preserves the flexibility of the sparse storage model, direct iteration would require an additional lookup for every processed entity.

Rather than performing these lookups inside the execution loop, the runtime constructs a temporary index cache before processing begins. This cache stores the indices of externally owned components corresponding to the entities contained within the owned group. Execution therefore follows two separate stages.

- The preparation stage performs the required sparse lookups once while constructing the cache

- The execution stage subsequently iterates over contiguous component arrays together with the cached indices, allowing external component access to proceed linearly without repeating sparse lookups during every iteration

This approach intentionally moves indirection outside the performance-critical loop, preserving the linear execution characteristics of owned groups while supporting arbitrary component composition

5.5 Runtime Communication

Communication between runtime modules follows the capability-oriented projection model described in the SDK architecture. Rather than exposing implementation-specific interfaces or relying on reflection, runtime type information or dynamic object hierarchies, modules communicate

exclusively through projected capabilities resolved during package initialization.
For native C++ applications, projection establishes direct bindings between the runtime and individual modules through statically defined function pointers. Once projection has completed, no additional discovery, interface resolution or dynamic dispatch is required during execution.

Consequently, communication across module boundaries consists only of direct native function calls while preserving complete implementation independence between participating modules. Projection therefore separates dependency resolution from execution. The runtime performs discovery exactly once during initialization, allowing execution itself to proceed without additional architectural overhead

5.6 Why Separate the Runtime?

The primary motivation for separating runtime responsibilities from functional implementation is not performance but architectural scalability. As software systems continue to grow, complexity increasingly originates from interactions between independently evolving features rather than the implementation of individual systems. Treating the engine as a runtime allows these responsibilities to be separated explicitly:

- The runtime becomes responsible for execution, scheduling, resource management, communication, package loading and capability orchestration
- Modules become responsible for implementing functionality
- Content becomes responsible for configuring behavior.

Each layer evolves independently while remaining connected through well-defined runtime abstractions. This separation provides several architectural advantages:

- Independent modules may be developed, tested, versioned and deployed without requiring modifications to the runtime itself
- Execution strategies may evolve independently of gameplay implementation
- Platform-specific services remain isolated within the runtime while application logic remains platform agnostic
- Architectural complexity shifts from compile-time integration towards explicit runtime composition, reducing coupling while preserving deterministic execution

The resulting architecture treats functionality as software executing inside the runtime rather than functionality defining the runtime itself

5.7 Trust and Reverse Engineering

A common concern regarding modular software concerns application security. The presence of independently deployable modules is often assumed to simplify reverse engineering or unauthorized modification compared to monolithic executables. In practice, both architectural models expose fundamentally identical attack surfaces. Native executables, shared libraries and runtime modules ultimately execute as machine code within the same process and remain equally accessible to

debugging, memory inspection and binary analysis.
The modular architecture therefore does not introduce a fundamentally new class of attacks. Instead, it enables trust to become an explicit responsibility of the runtime.

Package loading may be combined with manifest validation, digital signatures, integrity verification, capability validation and optional cryptographic protection without affecting the execution model itself. Because package discovery and trust validation remain independent concerns, applications may adopt security policies appropriate for development, testing or production environments without modifying runtime behavior. Additional mechanisms, including encrypted package distribution, server-provided session keys or chained trust relationships between packages, may further strengthen deployment scenarios while remaining orthogonal to the runtime architecture.

Consequently, modularity should not be interpreted as a reduction in application security. Rather, it provides clearly defined integration points through which authentication, integrity verification and deployment policies can be implemented while preserving the flexibility of independently deployable runtime modules

6. Conclusion

Game engine architecture has continuously evolved in response to the challenges of its time. Early engines emerged from the need for software reuse, later evolving into increasingly comprehensive platforms that combined rendering, physics, animation, audio, scripting and tooling within a single software foundation. This progression proved remarkably successful and continues to influence the design of many modern engines.
However, the challenges facing contemporary game development differ fundamentally from those that shaped these architectures. Modern projects emphasize long-term maintainability, distributed development, independent feature evolution, continuous deployment and data-driven workflows in often large distributed teams. As a consequence, architectural complexity has shifted from implementing functionality to managing the relationships between independently evolving software components.

This architectural model proposes a corresponding shift in perspective. Rather than viewing the engine as the implementation of application functionality, the engine is instead regarded as the runtime responsible for executing that functionality. Resource management, scheduling, package loading, communication and capability orchestration become responsibilities of the runtime, while gameplay systems, rendering pipelines, physics simulations and other application features are implemented as independent runtime modules.

This separation does not eliminate existing engine technologies, nor does it replace established concepts such as data-oriented design or entity component systems. Instead, it provides a common execution environment in which these concepts may coexist while remaining architecturally independent. Performance continues to be determined primarily by data layout and execution strategy, whereas modularity becomes a property of software organization rather than a limitation of runtime efficiency.

Equally important, the separation between runtime, software and content establishes a

clear architectural hierarchy. The runtime defines how software executes, software defines available behavior through capabilities, and content determines how this behavior is composed into a complete application. Each layer addresses a distinct set of responsibilities while remaining connected through explicit runtime abstractions.

The resulting architecture represents an evolutionary step rather than a revolutionary departure from existing practice. It preserves the performance characteristics of modern data-oriented systems while introducing a runtime model designed to address the organizational and architectural challenges of contemporary game development.

As software systems continue to increase in scale and complexity, the role of the engine is likely to evolve accordingly. Future engine architectures may therefore be characterized less by the functionality they implement and more by the execution environments they provide. From this perspective, the engine ceases to be viewed as a collection of subsystems and instead becomes the operating system on which the game executes