SYNCHROINFO JOURNAL

www.media-publisher.eu/synchroinfo-journal

ECS COMPREHENSIVE STUDY IN HIGH-PERFORMANCE AND SECURE GAME DEVELOPMENT

I.D. Udalov [1], **V.V. Maklachkova** [1], **V.A. Dokuchaev** [1]

[1] Network Information Technologies and Services, MTUCI, Moscow, Russia;
igor.udalov.95@mail.ru, v.v.maklachkova@mtuci.ru, v.a.dokuchaev@mtuci.ru

ABSTRACT

This article explores the Entity Component System (ECS) paradigm as a modern architectural approach for constructing scalable, secure, and high-performance game systems. ECS is examined in contrast with traditional object-oriented programming (OOP), emphasising significant improvements in performance, memory efficiency, modularity, and behavioural flexibility. The study extends beyond classical comparisons and discusses the broader implications of adopting data-oriented architectures for secure game development, including the protection of personal data within large-scale interactive platforms. Special attention is devoted to Unity's Data-Oriented Technology Stack (DOTS), which serves as an example of an industrial ECS implementation. The article expands the analysis of security mechanisms inherent in ECS workflows, memory management, and job systems, offering a deeper understanding of how the architectural pattern influences safe data handling in modern games.

KEYWORDS: *Entity Component System; ECS, data-oriented architecture; Unity DOTS; game development; personal data protection, memory safety; software security.*

DOI: 10.36724/2664-066X-2025-11-5-10-17

Received: 17.08.2025
Accepted: 18.10.2025

Citation: I.D. Udalov, V.V. Maklachkova, V.A. Dokuchaev, "ECS comprehensive study in high-performance and secure game development", *Synchroinfo Journal* **2025**, vol. 11, no. 5, pp. 10-17.

Licensee IRIS, Vienna, Austria.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

Copyright: © 2025 by the authors.

1 Introduction

The evolution of digital entertainment has significantly changed the expectations imposed on game engines and interactive systems. Contemporary projects no longer consist of a few dozen objects acting in isolation; instead, they frequently contain thousands or even millions of entities that must be processed in real time. These include agents in crowd simulations, dynamic environmental elements, physics-driven particles, AI actors, destructible geometry, persistent world objects, and a constantly expanding assortment of interactive game elements. Each of these components contributes additional load on computational resources, making efficient processing an indispensable requirement of modern game engines.

The pursuit of realism, immersion, and system-level complexity has exposed the limitations of traditional OOP-based engines, which often struggle to maintain high frame rates when dealing with such volumes of data. OOP's reliance on object encapsulation and polymorphism, while conceptually elegant, introduces a substantial amount of overhead at the hardware level. Objects tend to be scattered across memory, resulting in unpredictable memory access patterns and frequent cache misses. Additionally, deep inheritance chains and virtual function calls introduce branching costs that degrade performance, especially in large-scale simulations [1-3].

The historical prevalence of object-oriented programming (OOP) in game development is understandable: encapsulation, inheritance, and polymorphism provide intuitive abstractions for modelling game logic. However, as real-time requirements tighten and hardware evolves toward multi-core, multi-threaded designs, the gap between intuitive modelling and efficient computation widens. OOP representations incur significant object overhead, mutable shared state, and difficulties in thread-safe parallelisation, making high-performance simulations increasingly difficult to maintain, especially in complex, large-scale worlds where thousands of objects need to be updated simultaneously.

In response to these limitations, the game industry has gradually moved towards architectures based on data-centric reasoning. Among these, the Entity Component System has become one of the most influential paradigms. While ECS has existed conceptually for decades, only recent hardware trends – multi-core CPUs, wide SIMD instructions, heterogeneous computing environments, and deep cache hierarchies – have made the approach not only relevant but often necessary. By organising data in contiguous memory arrays and decoupling logic from state, ECS enables processors to execute operations with maximal efficiency [4].

This article offers a substantially expanded exploration of ECS, covering its theoretical underpinnings, practical benefits, and implications for data security. It also discusses how ECS contributes to safer processing of personal information by enforcing deterministic data access patterns, explicit memory management, and structural guarantees that reduce risks such as data races, privilege leaks, and uncontrolled state mutation. The following sections analyse the differences between traditional OOP and ECS, allowing for a more detailed understanding of why the latter is rapidly gaining traction in the development of modern games.

2 Comparison Between ECS and Object-Oriented Programming

ECS fundamentally changes the way game logic is represented. Figure 1 illustrates the architectural differences between OOP and ECS.

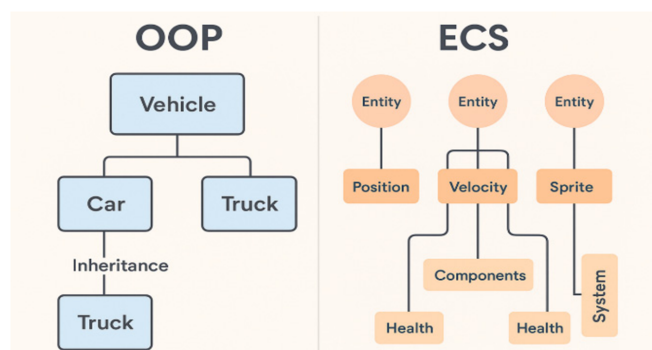

Figure 1. Conceptual comparison of OOP and ECS architecture

While OOP binds data and behaviour inside an object and allows objects to interact via method calls, ECS decomposes this structure into three separate concepts. Entities act as unique IDs, components contain only data, and systems implement behaviour in pure processing routines. This separation removes typical OOP constraints, allowing games to scale more effectively and enabling parallelism that would be exceedingly difficult to achieve with traditional object hierarchies [5].

In an OOP model, objects encapsulate both behaviour and state, forming deep inheritance structures to represent shared features. This often leads to rigid hierarchies that become increasingly fragile as projects grow. Even minor modifications in base classes may propagate unexpected changes within derived classes, resulting in bugs that are difficult to diagnose [6]. Moreover, due to the scattered nature of object memory layouts, processing large numbers of objects becomes inefficient, as frequent jumps across memory slow down CPU pipelines and lead to unpredictable performance.

In contrast, ECS emphasises composition rather than inheritance. Entities can be composed dynamically by attaching different components, each representing a specific data attribute. Systems then process only the relevant components, operating on large contiguous data sets. This data-oriented design results in highly predictable, linear memory access patterns, enabling processors to prefetch information efficiently and execute instructions with fewer pipeline stalls. Core differences are listed in Table 1.

Table 1

Comparative Characteristics of OOP and ECS

Characteristic	Object-Oriented Programming (OOP)	Entity Component System (ECS)
Organization	Objects encapsulate data and methods	Entities (ID), Components (data), Systems (logic)
Behaviour	Internal methods, inheritance	Behaviour defined in external systems
Reusability	Inheritance is the main mechanism	Composition via adding components
Memory Layout	Scattered, unpredictable object placement	Contiguous arrays of homogeneous components
Main Focus	Object behaviour and modelling	Efficient data processing
Optimization Difficulty	Harder to optimize at scale	Designed for cache efficiency and parallelism
Parallel Processing	Limited and complex	Natural parallelization via systems

While the table highlights structural distinctions, the larger contrast emerges when these models are applied to real workloads. OOP code tends to degrade in performance as systems grow in complexity, because even simple operations – like iterating through hundreds of enemies – can involve jumping through layers of indirection. Virtual function calls and pointer dereferencing introduce branching and unpredictable memory access patterns, making it difficult for CPU caches to operate efficiently.

ECS eliminates most of this overhead by reinterpreting game logic as sequential data transformations. Instead of each object individually updating itself, systems operate on large sets of homogeneous components, allowing for optimised loops that leverage vectorisation, parallelism, and cache locality. This approach transforms the performance characteristics of game logic, enabling large-scale simulations that would be impractical under traditional OOP.

3 Advantages and Limitations of the Entity Component System

The primary benefit of ECS is its capacity to organize data in a way that aligns with modern CPU architecture. By placing identical components in contiguous memory, ECS allows processors to prefetch data efficiently, enabling vectorized operations over arrays.

This drastically reduces cache misses and enables systems to process thousands of entities in tight, predictable loops [7]. The overall effect is not simply incremental performance improvement but a structural rethinking of game execution, where computational flow becomes far more aligned with hardware behaviour.

In an OOP-based engine, each individual object typically contains multiple pieces of data grouped together – some of which may be needed only occasionally. When a system attempts to iterate over all objects to update, for instance, position or velocity, it must jump from one object's memory location to another, repeatedly pulling entire memory blocks that contain unused fields. This leads to wasted bandwidth and decreased cache utilisation. ECS eliminates these inefficiencies by ensuring that only relevant component data is loaded into the cache, significantly reducing memory traffic.

The absence of inheritance has profound implications. Developers can construct entities dynamically using combinations of components instead of rigid hierarchies. This approach eliminates deep class trees, reduces the fragility of shared base classes, and simplifies the introduction of new gameplay mechanics. In many large OOP projects, developers encounter the "diamond problem," unintended overriding behaviours, or the need to refactor long inheritance chains when introducing a new feature. ECS avoids all these issues entirely by promoting modularity and flexibility through composition. Instead of creating ten subclasses for different types of enemies, developers simply attach or remove components to form new behaviours.

Testing becomes easier because components are pure data, and systems are pure functions operating on them. Such deterministic logic is easier to validate, optimize, isolate, and secure. A system, being stateless and reactive, can be tested using simple input-output patterns [8]. Developers no longer need to instantiate complex object graphs to replicate game behaviour. Systems can be fed synthetic data in controlled environments, leading to more reliable and robust tests.

However, some challenges accompany these benefits. Developers accustomed to OOP must adjust their way of thinking. ECS requires explicit component definitions, explicit system registrations, and explicit memory management when working with native containers. Debugging can be initially difficult because logic is distributed across multiple systems rather than contained within self-describing objects, which means that tracing behaviour requires understanding execution order, job dependencies, and the specific data a system processes.

Integration with legacy code or third-party libraries can also be challenging, as many external tools and frameworks are designed with an OOP mindset. Nonetheless, once teams adapt to ECS principles, the advantages typically outweigh the early friction. The structural clarity of ECS often leads to better long-term maintainability, more consistent performance, and a more scalable architecture suitable for future expansion.

4 Combining OOP and ECS in a Game Project

Although ECS provides strong performance benefits, game development rarely adopts it exclusively. Many systems – such as UI logic, menu navigation, high-level state machines, cutscene tools, or editor utilities – are often more naturally implemented in OOP due to their complexity or uniqueness. These systems usually involve highly specific behaviour that does not require the mass processing of similar data sets, which makes the overhead of ECS unjustified.

Meanwhile, ECS is ideal for large, repetitive, data-driven processes such as movement, AI behaviour, transformation updates, or physics integration. Systems that must handle thousands of similar objects each frame – like projectiles, enemies, particles, or environmental elements – benefit immensely from ECS's memory locality and parallel execution capabilities. This hybrid approach allows both paradigms to coexist, leveraging the strengths of each while compensating for their respective weaknesses.

Many modern engines, including Unity, are specifically designed to support this blended architecture. Unity allows developers to write gameplay using MonoBehaviours (OOP) while simultaneously using DOTS for performance-critical subsystems. As a result, designers and gameplay programmers can continue using familiar tools, while engine developers and technical teams enhance performance-sensitive areas with ECS.

Large game studios often adopt an incremental approach: existing OOP systems remain untouched, while computationally expensive parts of the game are gradually migrated to ECS. For example, AI perception, target selection, animation culling, and physics broad-phase detection can be moved to ECS first. Over time, the engine evolves

into a hybrid system in which performance-sensitive loops are fully data-oriented, while configuration-heavy or narrative-driven systems remain in OOP [9].

This hybrid model also improves maintainability, since the boundaries between ECS-driven systems and OOP subsystems create natural modularity. Each subsystem can be developed, tested, and optimised independently. As the game evolves, developers may migrate additional systems to ECS when needed. This flexible integration approach is one of the reasons ECS adoption is steadily growing in the gaming industry.

5 Unity DOTS as an Applied Example of ECS

Unity's Data-Oriented Technology Stack serves as one of the most mature mainstream implementations of ECS principles. The DOTS ecosystem was designed specifically to handle large-scale simulations, extremely high entity counts, and multithreaded execution. It includes several tightly integrated components, each crafted to address specific bottlenecks in traditional Unity workflows.

Unity Entities provides the foundation: a robust ECS framework that manages entities, archetypes, and chunks. The architecture is built around a highly structured memory layout that allows components of the same type to reside within contiguous blocks. This chunk-based storage ensures excellent memory locality and facilitates optimised iteration over entities.

The C# Job System enables safe multithreading by offering a high-level interface for scheduling parallel tasks. Instead of manually creating threads, locking shared resources, and managing concurrency, developers define lightweight jobs that express what data they read or write. Unity's scheduler then automatically handles job dependencies, ensuring that no two jobs write to the same component simultaneously [10].

The Burst Compiler plays a critical role by converting high-level C# code into highly optimized native machine code. Burst harnesses vector instructions, register-level optimisations, branch elimination, and loop unrolling to produce exceptionally fast executables. Systems compiled with Burst often rival or surpass hand-written C++ in performance.

Native Collections integrate tightly with Burst and the Job System, enabling predictable memory usage and ensuring deterministic access patterns. These collections – including NativeArray, NativeList, and NativeHashMap – are designed to avoid the overhead of garbage collection and provide explicit control over memory lifetime.

The combined effect of these technologies is transformative. By restructuring the engine around data flow rather than object hierarchies, Unity enables extremely large worlds with tens or hundreds of thousands of active entities. Simulations that were once impossible to maintain within acceptable frame-rate targets become feasible, allowing developers to experiment with richer gameplay mechanics, larger worlds, and more dynamic interactions.

6 Security of Personal Data and ECS-Oriented Architecture

One of the most significant additions in this expanded article is the detailed examination of how ECS influences the safety, confidentiality, and integrity of personal data within modern gaming platforms. As game environments increasingly integrate monetization systems, biometric tracking, behavioural analysis, and real-time telemetry, the amount of sensitive information collected from players grows dramatically. This data may include gameplay preferences, social interactions, location information, device fingerprints, session logs, and even biometric inferences derived from player behaviour.

Ensuring the safety of this data is not merely a technical requirement but a legal obligation, especially given the regulations defined in legislation such as Federal Law "On Personal Data" (152-FZ), GDPR, CCPA, and other international standards. ECS, due to its structural separation of data and logic, naturally facilitates compliance with such regulations by ensuring that sensitive data is stored and accessed in predictable and controlled ways.

ECS contributes to safer data handling in several ways. The separation of data and logic inherently reduces the likelihood of accidental or unauthorized access to sensitive fields [11]. Components containing personal information or identifiers can be isolated in

dedicated memory segments processed by specialized systems with restricted access patterns. Developers may tag such components as write-protected or store them in archetypes accessible only to trusted systems.

Since systems operate only on explicitly declared component types, the architecture inherently promotes minimization: systems cannot accidentally read or write personal data unless explicitly allowed to do so. This sharply contrasts with traditional OOP systems, where an object might contain multiple unrelated fields and methods, any of which could unintentionally expose data to different parts of the codebase.

Safety is further enhanced by Unity's Job System, which prohibits unsafe race conditions by analyzing job dependencies and enforcing read/write constraints. Attempting to write to protected data from multiple threads immediately triggers diagnostics, preventing subtle bugs that could otherwise cause data leaks or corruption. The Job System enforces strict memory access discipline, ensuring that sensitive information is manipulated only by authorized threads at predictable points in the execution cycle.

Native containers in DOTS introduce explicit memory ownership, forcing developers to handle lifecycle events deliberately. This explicit handling reduces the risk of unauthorized access to freed or reallocated memory, a common cause of vulnerabilities in unmanaged systems. Because ECS encourages deterministic update loops, audits of data flow become simpler. Security teams can track which systems access specific data types, making compliance easier and risk analysis more transparent [12].

Another security-related benefit is the natural alignment of ECS with sandboxed simulation logic. When personal data is stored in separate components, potentially sensitive information can be segmented away from gameplay systems, allowing internal firewalls or logic gates to restrict access. Such structural compartmentalization is far more difficult in traditional OOP models, where data and behaviour reside tightly coupled inside monolithic, mutable hierarchies.

As game companies increasingly rely on telemetry, behavioural prediction, machine learning analytics, and cross-platform user identification, the architectural discipline imposed by ECS becomes a valuable safeguard. Large batches of analytics data can be processed in dedicated systems with strong read-only guarantees, preventing unintentional mutation [13]. Furthermore, because ECS encourages immutable or semi-immutable data flows, it enhances auditability and decreases the attack surface for malicious actors attempting to manipulate internal game state.

Thus, ECS plays a meaningful role not only in performance and architectural clarity but also in strengthening personal-data security through deterministic, explicit, and compartmentalized data-handling patterns. In an industry where data breaches can lead to serious legal, financial, and reputational harm, the advantages of this architecture extend far beyond computational efficiency [14].

7 Practical ECS Example: Movement Processing in DOTS

A practical illustration is the implementation of a movement system. Instead of each object carrying its own script with logic, DOTS decomposes movement into data components – such as Position and Velocity – and a system that processes all entities containing both. This structure makes it possible to run highly optimized movement calculations across thousands of entities at once.

Because components are stored contiguously in memory, the system can load large batches of positions and velocities into the cache with minimal overhead. The Burst compiler can further optimize tight loops by converting them into vectorized instructions, enabling the CPU to update multiple entities simultaneously. Large simulations, which would require nested loops or expensive function calls in OOP, can be reduced to simple, predictable sequential operations.

This model dramatically outperforms traditional OOP GameObject-based movement, where each object executes its own Update() method, generates overhead, and performs unpredictable memory references. In Unity's classic OOP workflow, virtual function calls prevent effective inlining and block many compiler optimisations. Each object's state may reside in widely separated memory locations, causing cache thrashing during iteration.

DOTS consolidates the entire operation into a single optimised pipeline. Moreover, by using the Job System, movement updates can be split across multiple CPU cores, allowing near-linear scaling with available hardware. Whether processing fifty entities or fifty thousand, DOTS systems maintain stable performance characteristics thanks to their cache-friendly layout and predictable execution flow [15, 16].

In addition to basic movement, more complex behaviours – such as flocking algorithms, physics-based motion, or real-time steering – also greatly benefit from ECS. Because systems operate on large homogeneous datasets, even sophisticated operations such as Boids simulations or collision sweeps can be performed efficiently at scale.

8 Conclusion

The Entity Component System represents a foundational shift in game architecture, replacing decades of object-centric thinking with a model optimized for modern hardware, massive parallelism, and predictable data flow. In a landscape where game worlds are becoming increasingly complex and data-rich, ECS offers developers the ability to handle large-scale simulations with unprecedented performance and clarity. The architectural separation of identity, data, and behaviour provides a transparent and extensible structure that scales naturally with project size and complexity.

Unity's DOTS implementation demonstrates how ECS principles transform real projects, enabling massive simulation throughput while improving data safety, memory determinism, and the clarity of computational workflows. By aligning with the underlying physical characteristics of modern processors, DOTS unlocks optimisations that are difficult or impossible to achieve in traditional OOP systems.

Although the learning curve is significant, especially for developers deeply accustomed to OOP, the benefits in terms of performance, modularity, and security make ECS an increasingly compelling paradigm for the next generation of game engines. The shift towards data-oriented design reflects broader trends in software engineering, where performance, safety, and scalability require rethinking long-established paradigms.

Given the accelerating growth of real-time interactive worlds and the rising importance of secure personal-data processing, ECS is likely to remain a cornerstone of future game development methodologies. As game engines continue evolving and as hardware becomes increasingly parallel, ECS offers a robust and future-proof foundation capable of supporting the next generation of interactive experiences.

REFERENCES

[1] Ernest Adams, J. Dormans. Game Mechanics: Advanced Game Design. 2012. [Online]. URL: https://typeset.io/papers/game-mechanics-advanced-game-design-23pl62mlvp [Accessed: Nov., 2025].

[2] D.H. Eberly, "3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics," Morgan Kaufmann, 2006. [Online]. URL: https://www.academia.edu/26649713/3D_game_engine_design_a_practical_approach_to_real_time_computer_graphics_second_edition [Accessed: Nov., 2025).

[3] R. Zubek, "Game Physics Engine Development," CRC Press, 2018.

[4] A. Brown, "Entity-Component Systems and C#," Apress, 2019.

[5] M.V. Askerli, "A Key to Efficient Development: ECS Architecture in Comparison," *Bulletin of Science*, Vol. 1, no. 5 (74), pp. 490-495, 2024. URL: estnik-nauki.com/article/14272 [Accessed: Oct., 2025].

[6] V.A. Dokuchaev, "Influence of New Information and Communication Technologies on the Confidentiality of Personal Data," *In Proceedings of the XXIII International Scientific and Practical Conference "Current Problems and Prospects of Economic Development",* Simferopol – Gurzuf, October 17-19, 2024. Simferopol: Zueva T.V. Publishing, 2024, pp. 12-15.

[7] V.Y. Statev, V.A. Dokuchaev, and V.V. Maklachkova, "Information security in the big data space," *T-Comm*, vol. 16, no. 4, pp. 21-28, 2022, doi: 10.36724/2072-8735-2022-16-4-21-28.

[8] V.A. Dokuchaev, V.V. Maklachkova, V.Yu. Statev, "Data subject as augmented reality," Synchroinfo Journal, vol.6, no.1, pp.11-15, 2020.

[9] V.A. Dokuchaev, K.S. Vladimirova, V.V. Maklachkova, V.Yu. Statyev, "Audit of Information Risks in Personal Data Processing," *In: Information Society Technologies: Proceedings of the XIII International Industry Scientific and Technical Conference*, Moscow, March 20-21, 2019. Vol. 2. Moscow: Media Publisher, 2019, pp. 34-36.

[10] V.A. Dokuchaev, V.V. Maklachkova, V.O. Sundatov, “Approaches to Protecting Personal Data in the Big Data Space,” *In: Theory and Practice of Economics and Entrepreneurship: Proceedings of the XX International Scientific and Practical Conference*, Simferopol – Gurzuf, April 20–22, 2023. E . N.V. Apatova. Simferopol: V.I. Vernadsky Crimean Federal University, 2023, pp. 34-36.
[11] V.A. Dokuchaev, “Classification of Personal Data Security Threats in Information Systems,” *T-Comm*, 2020, Vol. 14, No. 1, pp. 56-60. DOI: 10.36724/2072-8735-2020-14-1-56-60.
[12] V.A. Dokuchaev, “Digitalization of the Personal Data Subject,” *T-Comm*, 2020, Vol. 14, No. 6, pp. 27-32. DOI: 10.36724/2072-8735-2020-14-6-27-32.
[13] V.I. Nemanova, I.S. Vakurin, V.V. Maklachkova, D.V. Gadasin, “Determining the Economic Efficiency of Enterprise Expenditures on Personal Data Protection,” *DSPA: Issues of Digital Signal Processing Application*, 2024, Vol. 14, No. 3, pp. 37-45.
[14] V.A. Dokuchaev, “Formulation of the Problem of Assessing Information Quality in Personal Data Processing within Multi-Cloud Information Systems,” *In: Theory and Practice of Economics and Entrepreneurship: Proceedings of the XX International Scientific and Practical Conference*, Simferopol – Gurzuf, April 20-22, 2023. Ed. N.V. Apatova. Simferopol: V.I. Vernadsky Crimean Federal University, 2023, pp. 37-39.
[15] V.A. Dokuchaev, “Formulation of the Problem of Assessing Information Quality in Personal Data Processing within Multi-Cloud Information Systems,” *In: Theory and Practice of Economics and Entrepreneurship*, 2023, pp. 37-39.
[16] Entity Systems Are the Future of MMOG Development. [Online]. URL: https://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ [Accessed: Oct., 2025].