Performance and Complexity Trade-offs in Archetype-Based Entity-Component-System

Laurent Voisard

A Thesis

in The Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

June 2026

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Laurent Voisard**

Entitled: **Performance and Complexity Trade-offs in Archetype-Based Entity-Component-System**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Name of the Chair

_____ Examiner
Dr. Tiberiu Popa

_____ Examiner
Dr. Charalambos Poullis

_____ Supervisor
Dr. Yann-Gaël Guéhéneuc

_____ Co-supervisors
Dr. Cristiano Politowski and Dr. Fabio Petrillo

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2026

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Performance and Complexity Trade-offs in Archetype-Based Entity-Component-System

Laurent Voisard

Entity-Component-System (ECS) is a software design pattern increasingly used in the development of real-time applications, particularly video games. Despite its growing adoption, the ECS pattern remains relatively underexplored in academic research, especially with respect to empirical evaluations of how design decisions impact runtime performance and code complexity in practice.

This thesis investigates the ECS pattern from both a systematic mapping and an empirical perspective. First, we perform a systematic mapping study to identify and analyse existing ECS research in software engineering. We identify how the ECS pattern is defined, the primary research focuses, and the commonly reported benefits and drawbacks. The results indicate that the number of academic publications has been growing since 2015 and are published in a variety of journals, conferences, and workshops, and that studies focus more on the benefits of the ECS pattern rather than its drawbacks. The mapping study also revealed that existing work emphasises high-level architectural concepts of ECS, with limited empirical analysis of implementation-level behaviour.

To address the aforementioned research gap, this thesis provides a detailed examination of the archetype-based ECS storage model, with a focus on performance and code complexity trade-offs. We conduct an empirical case study to evaluate multiple ECS-based collision-handling implementations within a real-time application context. Using a representative ECS-based game developed for this study — one of the featured projects by the Flecs ECS framework — we compare six implementations in terms of runtime performance and code complexity. The results show that implementations based on relationships exhibit performance limitations due to archetype fragmentation, while spatial partitioning techniques improve performance at the cost of increased implementation

complexity. We further apply an Analytic Hierarchy Process (AHP) to support developers' decision-making in balancing performance and complexity criteria.

This thesis bridges the literature gap between (high-level) design decisions and (low-level) implementation considerations when using the ECS pattern by providing both empirical evidence and practical guidance for developers working with the ECS pattern in performance-sensitive applications.

Acknowledgments

J'aimerais prendre le temps de remercier mes proches, qui ont rendu la rédaction de cette thèse possible. À mes parents, vous m'avez soutenu financièrement et émotionnellement, vous m'avez toujours poussé pour que je puisse atteindre un niveau d'excellence que je ne croyais pas avoir en moi. Merci, Laurence, et merci, Robert. À Élodie, tu as su m'épauler à travers les innombrables projets, devoirs, examens, etc. Pour toutes les fins de semaine où j'ai dû travailler sur des remises de travaux, tu t'es toujours occupée de tout le reste. Merci, Élodie.

Ensuite, j'aimerais remercier mon superviseur, Yann, ainsi que mes co-superviseurs, Fabio et Cristiano. Yann, je ne crois pas que j'aurais pu tomber sur un meilleur superviseur, sérieusement. Tu m'as permis de faire de la recherche dans un domaine qui me passionne vraiment. Tout au long de mon parcours de maîtrise, tu m'as conseillé, guidé, enseigné, de façon que j'en sorte beaucoup plus grand, pas seulement comme étudiant ou développeur, mais comme personne. Merci, Yann. Fabio, le gars de Chicout ! Tes idées de grandeur et ta passion pour ce que tu fais sont palpables, et si je pouvais en prendre une leçon, ça serait bien de rêver en grand comme toi. Les détails peuvent toujours être figurés plus tard. Merci, Fabio. Lastly, but not least, Cristiano, if it wasn't for you, I don't think Yann would have even considered game studies, so I guess I owe my whole master's to you in some sense? Anyways, I really appreciate all the work you did to help me find my research topic and for my publications. And also, thank you for taking all the fire from Yann. I don't think I would be able to handle all that. Thank you, Cristiano.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The significance of video games as a technological and cultural force continues to grow, with the global games market projected to generate $206.5 billion USD in 2028, and currently stands at $188.8 billion in 2025 [1]. This steady growth, driven largely by mobile gaming, emerging hardware platforms, and increasingly complex real-time experiences, underscores the need for high-performance and maintainable software architectures. As game systems scale in complexity, efficient design patterns, like the Entity-Component-System (ECS) pattern, become increasingly relevant for meeting the performance demands of contemporary titles and development teams while maintaining code complexity low and maintainability high. Game engines such as Unity and Bevy even include their own ECS implementations.

As there are no commonly accepted terms to describe ECS, we follow the example set out by [2], and will refer to ECS as a design pattern, or ECS pattern for short. When we talk about an implementation of the ECS pattern, we refer to it as an ECS framework. In addition, we will use the full name of Entity-Component-System.

The ECS pattern emphasises composition over inheritance, and is often implemented in a data-oriented (see section 2.1) manner. Rather than organising behaviour and state through class hierarchies, ECS separates data (components) from logic (systems). Entities, associated with components,

represent game objects or simulation elements. This design enables better memory locality, cache-friendly processing, and parallel execution, which are increasingly important in modern multi-core systems.

1.2 Problem

Despite ECS's growing adoption in commercial game engines and independent projects, existing academic research on the pattern remains relatively limited in scope. Current work predominantly focuses on high-level structural concepts and reported benefits, such as modularity, flexibility, and improved performance. However, comparatively few studies perform empirical evaluations of ECS as a whole, particularly with respect to how different design decisions influence runtime performance and code complexity in practice.

This lack of empirical and implementation-level analysis limits the understanding of how ECS behaves in real-world scenarios and hinders the development of informed design guidelines for developers.

1.3 Objectives

This thesis addresses this gap through the combination of a systematic literature mapping study and an empirical case study.

First, we conduct a systematic mapping study to identify trends, topics, and implementation considerations in existing ECS-related publications by asking the following research questions:

- **RQ1.1**: In software engineering, what are the **publication trends** on ECS?
- **RQ1.2**: In software engineering, what is the **research focus** on ECS?
- **RQ1.3**: In software engineering, what are the main **benefits and drawbacks** of using ECS?

From these questions, we identified the following gaps:

- **RG1**: Current research on ECS focuses on benefits above the drawbacks, suggesting a positive bias in the field.

- **RG2**: The limited discussion of ECS backend design also suggests an opportunity for further empirical research examining how implementation choices influence performance, memory behaviour, and scalability in real-time systems.

Second, to address the second gap **RG2** identified in the mapping study, we explore through a practical case study the runtime performance and code complexity of ECS collision implementations in a subset of a game developed during the span of this thesis, ECS-Survivors[1], which is featured on the Flecs ECS framework home page[2]. We ask the following research questions to guide the case study:

- **RQ2.1:** How do different ECS collision-handling methods exhibit measurable runtime performance variation?
- **RQ2.2:** How do the different integrations of these methods introduce code complexity that can be quantified using cognitive and cyclomatic metrics?

We relate the archetype concepts explained in Chapter 2 with six different collision implementations in the ECS pattern and compare runtime performance and code complexity metrics. Finally, we performed Analytic Hierarchical Process (AHP) under three scenarios: performance-first, low-complexity-first, or balanced, and recommend I5, I2 and I3, and I5 and I6, respectively.

1.4 Contributions

This case study bridges theoretical and practical perspectives, offering concrete guidance for developers working with ECS in performance-sensitive domains by making two main contributions.

First, it provides a systematic mapping study of ECS in software engineering, identifying current research trends, gaps, and limitations.

Second, it presents an empirical evaluation of multiple ECS-based collision-handling implementations, analysing both runtime performance and code complexity.

[1] https://github.com/ptidejteam/ecs-survivors
[2] https://www.flecs.dev/flecs/#ecs-survivors

As a necessary foundation for the empirical study, Chapter 2 provides a detailed technical examination of archetype-based ECS storage models — covering structural changes, archetype graphs, and fragmentation — which itself addresses part of RG2.

As an additional contribution, this thesis introduces ECS-Survivors, a publicly available ECS game. The project has been adopted as a featured example by the Flecs framework and has received engagement from the developer community, supporting its relevance as an environment for studying ECS design and behaviour. [3]

Together, these contributions provide a comprehensive multi-perspective examination of ECS. By combining an in-depth coverage of archetype-based ECSs, a systematic mapping study, and real-time experiments, this thesis hopes to advance the academic understanding of ECS and provide actionable insights for its use in software engineering and game development.

1.5 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 presents the technical background on ECS and data-oriented design, with particular emphasis on archetype-based storage models and their runtime performance implications. Chapter 3 describes the systematic mapping study and its findings. Chapter 4 introduces the empirical case study and evaluates the proposed implementations. In Chapter 5, we provide a general discussion that synthesises the findings across all chapters and reflects on their broader implications. Finally, Chapter 6 summarises the contributions and outlines directions for future work.

1.6 Related Publications

During the span of this thesis, we published one paper, "A Mapping Study of the Entity Component System Pattern", at the Games and Software Engineering (GAS) workshop at the International Conference on Software Engineering (ICSE) in Ottawa, 2025 [3]. In this thesis, we expanded the pool of primary studies identified in the original mapping study by applying the same method to any potentially relevant new academic papers.

[3] https://www.ptidej.net/news/250629-ecssurvivors

The first author performed the majority of the work. They worked on all the sections of the paper, which are mirrored and extended in this thesis, while the second author helped assess the relevance of the papers based on the inclusion and exclusion criteria. The remaining three authors helped with editing the paper and guiding the work.

We also sent a paper for review to the IEEE Transactions on Games journal (TOG) in December 2025. At the moment of writing this thesis, the paper submission has no updates. The paper presents a case study on the performance and code complexity of ECS systems in the context of collision implementations.

1.7 Adjacent Publications

We published a short paper simultaneously during my master's degree at the International Conference on Software Engineering (ICSE) workshop, Games and Software Engineering (GAS) 2026 [A1].

This short paper is a combination of two procedural techniques, Wave Function Collapse [A2] and Cyclic-Graphs [A3, A4], which enables procedurally generated levels that offer the same level of detail as handcrafted environments, while lowering the overall amount of work level designers perform.

Chapter 2

Background

This chapter establishes the foundations for the following chapters. First, we give an overview of data-oriented design and the Entity-Component-System design pattern. We then further explain ECS data storage models for sparse-set and archetype, with particular emphasis on archetypes, which is the type of ECS used in Chapter 4.

In this thesis, we focus on archetype-based ECS implementations due to their emphasis on data locality and iteration performance, which are particularly relevant for systems such as collision detection. To ground this discussion in a concrete and feature-complete framework, the concepts presented in this chapter are aligned with the design principles of Flecs, an archetype-based ECS that incorporates many features found in modern engines, such as Unity DOTS and Bevy, and used in our case study in Chapter 4.

2.1 Data-Oriented Design

Data-Oriented Design (DOD) is a software development method that prioritises data organisation and access efficiency. Originating from the game industry, DOD gained popularity among developers working on medium to large-scale games [4] where object-oriented development (OOD) could not deliver the required performance. The rise of data-intensive games, which demand complex simulations and realistic mechanics, along with the advent of multi-core gaming hardware, like the PlayStation 3 in the mid-2000s, highlighted the limitations of OOD, which often leads to

inefficient memory access patterns due to many, large, scattered objects, making poor use of CPU caches. In contrast, DOD focuses on structuring data to maximise cache efficiency and parallelism, improving performance and scalability—key advantages for modern game development [5].

2.2 Entity-Component-System

The Entity-Component-System pattern is primarily used in video games and real-time simulations, and usually implemented using DOD. ECS consists of three core elements [2]:

- **Entities**, which describe any relevant object. Each entity has a unique identifier, typically just a plain integer, that is associated with components. Entities function as handles that enable efficient data organisation, often in the form of arrays, to optimise data locality and improve cache efficiency and parallel processing [6].
- **Components**, describe only data in the form of structs or classes (*component types*), and should be devoid of any behaviour.
- **Systems**, which transform components and describe the behaviour of the entities. Each system is designed to perform an independent behaviour, enabling modular and maintainable logic.

2.3 Entity Relationships

Some modern ECS frameworks, like Bevy, Unity DOTS, and Flecs, allow users to define a relationship between entities, a feature that enables cross-entity communication.

Relationships are defined by combining a *relation* and a *target* [1]. This relationship then becomes a component, uniquely identified by the composition of the pair of elements. In this pair, the first element is described as the *relation*, e.g.: ”Owns”, and the second element corresponds to the relationship *target*, e.g.: ”Car”. So adding this relationship to an entity, e.g., ”Bob”, effectively gives the entity relationship: ”Bob Owns Car”. Of course, ”Bob” can own other things, like a

[1] https://www.flecs.dev/flecs/md_docs_2Relationships.html

bicycle, a house, a pet, etc., and each relationship is considered a unique component. Later in Chapter 4, we use relationships for some of the collision-handling implementations.

2.4 ECS Storage Models

Many different implementations of the Entity-Component-System exist, each differing in the fundamental data structures they use to store component data. Among these, two dominant approaches have emerged in publicly available ECS frameworks: sparse-set and archetype-based storage models [2, 7]. Both approaches have demonstrated their effectiveness in real-world applications and are widely adopted in modern game engines and frameworks.

Sparse-set implementations, such as EnTT [2], the most popular open-source ECS framework on GitHub, support a wide range of applications from large-scale commercial titles to independent games [3]. In contrast, several modern engines and frameworks adopt archetype-based storage. For example, the Bevy game engine is built entirely around an archetype-based ECS, enabling developers to create complete applications within a data-oriented paradigm.

These implementations illustrate that ECS is not a single unified design, but rather that it has different implementation variations. While the two storage models adhere strictly to data-oriented principles, they exhibit different runtime performance characteristics and trade-offs. Archetype implementations optimise heavily for multi-component iteration at the cost of structural mutations, whereas sparse-sets prioritise flat-rate, constant-time component modifications. Understanding these differences is essential when analysing the behaviour of ECS in performance-critical systems.

2.4.1 Sparse-Set

Sparse-set ECS implementations use both a sparse array and a dense array. The dense array contains the contiguous component data, while the sparse array is an index table that maps entity identifiers to their corresponding index within the dense array.

This design enables efficient insertion and removal of components for entities, as operations

[2] https://github.com/skypjack/entt
[3] https://github.com/skypjack/entt/wiki/EnTT-in-Action#engines-and-the-like

can be performed in constant time without requiring large-scale data movement [2, 8]. However, sparse-set ECS implementations typically exhibit lower iteration performance compared to archetype-based approaches. Systems operating on multiple components must often validate the presence of each component per entity and perform indirect lookups across separate arrays. As a result, memory access patterns may be less predictable, leading to reduced cache locality when compared to purely contiguous storage models.

Entity ID	Sparse Positions	Sparse Velocities
0	0	0
1	1	1
2	2	null

Dense Positions
(0,0)
(10, 10)
(30, 23)

Dense Velocities
(1, -1)
(0, -1)

Table 2.1: A basic sparse-set ECS representation

2.4.2 Archetypes

Archetype ECS implementations store entities with identical component compositions grouped into tables where columns represent contiguous blocks of memory for a specific component, and the rows represent the **archetype record** which holds the component data for an entity. Each entity points to its corresponding archetype record to access and write to its components. These tables are commonly referred to as archetypes.

This data layout enables faster iteration over entities in an archetype because of the predictable access patterns and cache-efficiency benefits gained from storing each component of the same type one after the other. Adding or removing components can effectively change an entity's archetype. These operations involve copying the memory of an archetype record from one archetype to another, making it significantly more expensive than the constant-time insertions or removals typically achieved in sparse-set implementations.

However, these added performance costs might not be obvious to identify for a developer interfacing with the high-level Application Programming Interface (API) of an ECS. As a result, developers may unintentionally introduce performance bottlenecks when performing frequent structural changes. Consequently, software developers must understand the internal behaviour of archetype transitions to design efficient ECS-based applications, as seemingly trivial operations at the interface

level can have significant implications on runtime performance.

A baseline approach to an ECS storage would consist of component lists, each with a length of n, where n corresponds to the number of entities in the application. Each component array is of length n even though a component might not be present for every entity.

Entity ID	Position	Velocity
0	(0,0)	(1, -1)
1	(10, 10)	(0, -1)
2	(30, 23)	null

Table 2.2: A basic ECS data storage layout

This approach leads to wasted memory by allocating additional space for components used by only a few entities. Archetypes solves this by decomposing component containers into multiple smaller tables containing only contiguous data of unique component compositions. From the layout in Table 2.2, an equivalent representation in the archetype storage model would decompose it into two archetypes, resulting in the Table 2.3.

Record ID	Position
0	(30, 23)

Entity ID
0
1
2

Record ID	Position	Velocity
0	(0,0)	(1, -1)
1	(10, 10)	(0, -1)

Table 2.3: A basic archetype ECS representation

Structural Changes

In dynamic applications, such as real-time simulations or video games, entities frequently undergo structural changes through the addition and removal of components. When such a change occurs in an archetype ECS, it moves all the components from an archetype record from one archetype to another that corresponds to the new combination of components.

Notable performance costs associated with the structural changes include the memory migration from one archetype to the other, as well as the cost of finding an appropriate archetype to hold the entity's components. The theoretical number of archetypes in an ECS application grows exponentially with the number of unique components, by a factor of 2^n, e.g., for three components `A B C`,

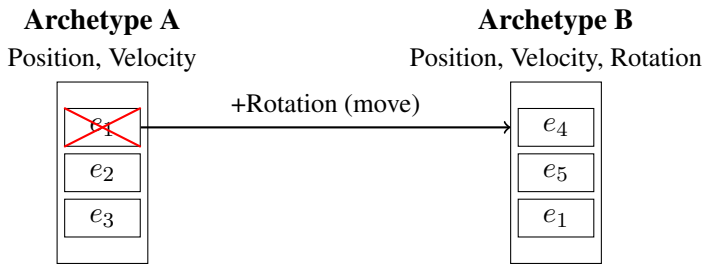

Figure 2.1: Structural change in an archetype-based ECS. Adding a component (e.g., Rotation) to an entity requires moving it from its current archetype to a new one matching the updated component composition. This operation involves copying component data into the destination archetype and removing the entity from the source archetype.

there are eight possible combinations (`__`, `A`, `B`, `C`, `AB`, `AC`, `BC`, `ABC`). With as few as ten components and 1024 archetypes, searching linearly for a correct archetype when performing an addition or a removal of a component will have severe performance costs. To optimise the new archetype search, ECS frameworks like Flecs use a data structure called an archetype-graph, an adjacency graph, which facilitates lookups when performing additions and removals. In an archetype graph, nodes represent archetypes and edges represent structural changes (add/remove). In the case where a `A` component is removed from an entity with an archetype composition of `(A, B)`, the archetype graph will typically perform an $O(1)$ direct edge lookup, indexed by a component operation, to find the new appropriate archetype.

Archetype Fragmentation

Flecs describes fragmentation[4] as a property in archetypes where entities can be spread out over multiple archetypes as the number of components (and combinations thereof) increases. This overhead is mostly present in archetype creations and queries, the latter needing to match and iterate over more tables [9]. For example, relationship `CollidedWith(A)` is different from `CollidedWith(B)`. In other words, there can be up to n different relationship components, each belonging to its own archetype, which results in many archetypes that contain very few, if any, entities, ultimately leading to a large amount of fragmentation and an increased overhead when iterating over the entities.

[4] https://www.flecs.dev/flecs/md_docs_2Relationships.html#fragmentation

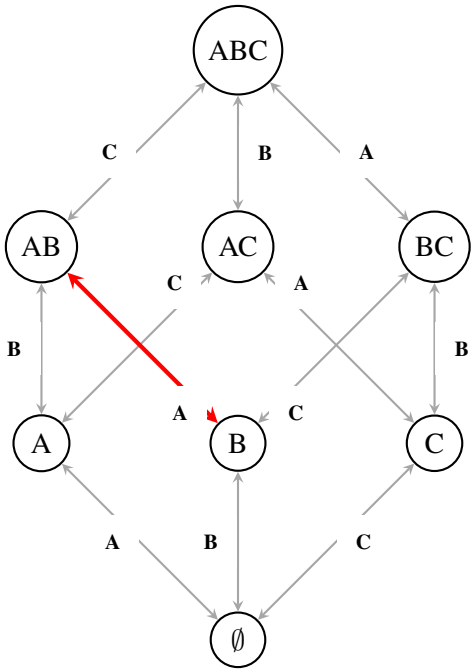

Figure 2.2: Archetype graph with bidirectional edges.
Archetype graph with bidirectional edges. Labels show the structural change for each component. Highlighted in red is the transition from archetype `AB` to archetype `B`.

Figure 2.3 shows an example of what archetypes could be generated in Flecs if an entity collided with another entity, `A`, `B` or `C`. Each combination of `CollidedWith(*)` is a new, unique component. As entities collide with others, Flecs will automatically create new archetypes using the archetype graph until all combinations have been exhausted, which could create up to 2^n new archetypes. As a result, a massive overhead is introduced when iterating each archetype with a `CollidedWith(*)` relationship component.

Flecs implemented a component trait called `DontFragment`, a flag that enables components to be stored in a separate sparse-set storage without having to create multiple archetypes that would contain just a few entities. This trait effectively eliminates the exponential growth of archetypes linked to relationships.

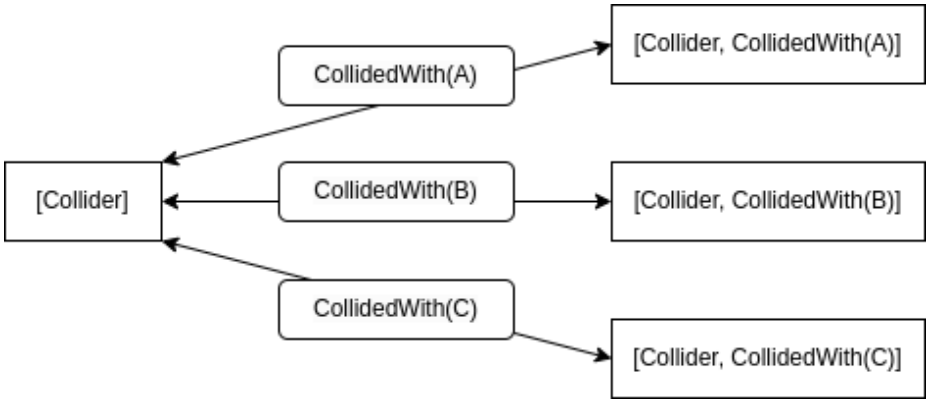

Figure 2.3: Archetype graph created from adding the `CollidedWith` relationship components. Each time a new `CollidedWith(*)` is added, it creates a new archetype, increasing the time for systems to iterate over all tables. Square boxes with arrays of components represent archetypes; Round boxes with components represent an addition or removal of said component.

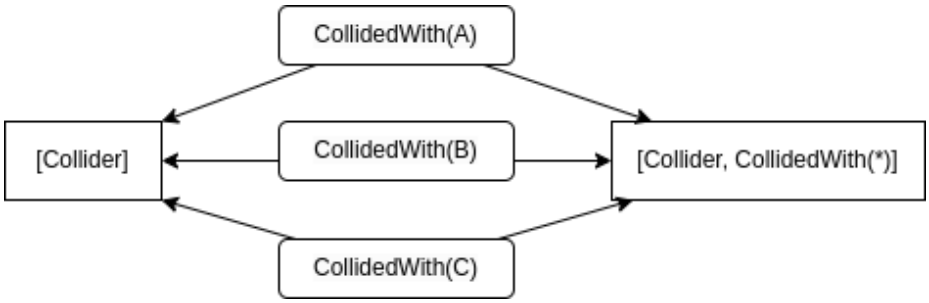

Figure 2.4: Archetype graph created from adding the `CollidedWith` relationship components with the `DontFragment` trait. Each time a new $CollidedWith(*)$ is added, no new archetype is created. Square boxes with arrays of components represent archetypes; Round boxes with components represent an addition or removal of said component.

Summary

While these structural differences suggest distinct performance trade-offs, existing academic literature provides limited empirical analysis of how these low-level design choices impact real-world ECS performance. The following chapter presents a systematic mapping study aimed at characterising current research on ECS and identifying gaps in the literature.

Chapter 3

Mapping Study

This chapter is based on a paper published at the Games and Software Engineering (GAS) workshop at the International Conference on Software Engineering (ICSE) in Ottawa, 2025 [3]. In this thesis, we expanded the pool of primary studies identified in the original mapping study by applying the same method to any potentially relevant new academic papers.

3.1 Introduction

Before diving into a specific topic to study ECS, we performed a systematic mapping study of the ECS pattern. Understanding the research field is a critical step before moving forward, and no existing mapping study existed at the moment we performed ours.

3.1.1 Problem

While ECS has seen growing adoption in the game-development industry, the same cannot be said for academia. The overall lack of documentation and research of ECS makes it difficult to assess its practical limitations with other popular game development architectures and design patterns. Hence, we perform a systematic mapping study to document existing literature in academia.

3.2 Related Work

According to Redmond et al. [M1], the existing literature on ECS is either too focused on low-level concepts like memory layouts or too dedicated to the high-level interfaces of specific ECS frameworks. They investigated several of the most prominent ECS frameworks in order to create a formal concept definition for ECS, anchored in calculus. They offer three main contributions: (i) Core ECS, which captures the essence of the ECS pattern, (ii) a characterisation of concurrency and determinism properties in ECS that can enable the deterministic-by-construction concurrent programming model for ECS, (iii) and by applying their Core ECS model to real-world ECS implementations, they show that many of the existing ECS frameworks could benefit even more from concorrent strategies possibly suggesting that alternative ECS implementation strategies could help enable greater concurrent capabilities.

3.3 Method

To structure our mapping study, we closely follow the guidelines established by Peterson [10] et al.. Our goal is to identify the publication trends of ECS, as well as the research interests/domains, and finally to extract the code quality metrics of ECS in software development.

3.3.1 Research Questions

To achieve said goal, we ask the following research questions:

- **RQ1.1:** *In software engineering, what are the **publication trends** on ECS?* By answering this question, we aim to map the academic interests in publications, popular publication venues, and the publication distribution over the years.
- **RQ1.2:** *In software engineering, what are the **research domains** of ECS?* By answering this question, we aim to classify existing research on ECS, which will consequently provide an understanding of the research gaps in the literature.
- **RQ1.3:** *In software engineering, what are the **benefits** and **drawbacks** of developing software*

Table 3.1: Number of studies per database

Database	Search results
ACM Digital Library	79
IEEE Explore	25
Scopus	50

using ECS? By addressing this question we aim to establish the design implications of the ECS design pattern.

3.3.2 Search and Selection Process

Guidelines given by [10, 11] both suggest that the search process should be performed in three of the largest databases for software engineering publications: ACM Digital Library, IEEE Explore, and Scopus; Hence we followed their direction. To have as many relevant primary studies (PS) as possible, we keep our search string simple, querying only for Entity-Component-System or terms alike. In the field of software engineering, ECS stands for many concepts, thus we did not include it in the research string.

”Entity Component System*” OR ”Entity-Component-System*”

We combine the acquired sources from the three databases: 79 from ACM, 25 from IEEE, and 50 from Scopus, totalling 91 distinct PSs.

3.3.3 Study Selection and Quality Assessment

Next, we develop a few evaluation criteria to help determine the relevance and quality of the PSs. Upon the results of the database search, we apply the following inclusion criteria:

- **I1** Papers where the Entity-Component-System is the focus or plays a supporting role.
- **I2** Papers written in English.
- **I3** Papers that are peer-reviewed.

Additionally, we also apply the following exclusion criteria:

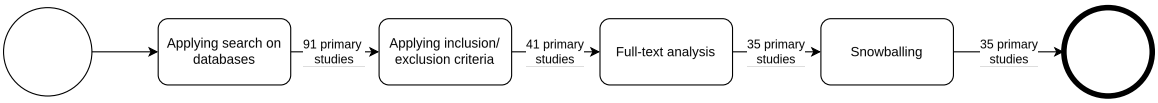

Figure 3.1: Study selection and quality assessment flow chart

- **E1** Papers that only briefly mention the Entity-Component-System and never define or explain it.
- **E2** Papers that confused the Entity-Component-System for similar patterns, such as the entity-component framework.
- **E3** Books and grey literature.
- **E4** Papers that are duplicates or shorter versions of other studies.

In the first step of the overall review, two of the authors apply the criteria previously listed(subsection 3.3.3, subsection 3.3.3) to the primary studies, based on their abstracts, titles, and keywords. If it was not immediately obvious whether the papers should be included or not, then the authors performed a partial-text reading for further investigation. Finally, if either of the two authors was not completely convinced whether a PS should be included or excluded, then both authors consulted each other to make a decision. This first step excludes 50 studies, leaving **41** PSs for the next step.

In the second step, the authors perform a full-text reading of the remaining PSs, once more applying the inclusion and exclusion criteria. This step excludes another 6 papers, resulting in **35** PSs.

Finally, in a third and last step, we perform a snowballing activity to ensure that no other possible primary study was overlooked. We use the tool Research Rabbit [1] to automatically snowball, given the 35 PSs. This step did not introduce new relevant papers for the mapping study.

In the end, after applying the three steps above, we obtain **35** PSs. See Figure 3.1 for an overview of the selection processes.

[1] https://www.researchrabbit.ai/

3.4 Results

The following section goes over the findings of each of the research questions mentioned previously.

3.4.1 Publication Trends (RQ1.1)

We extract information from the 35 PSs in order to answer our first research question. We first perform a general analysis of the research trends, notably the publication venues and the distribution of publication years of the PSs. Since ECS is said to be primarily used for games outside of academia [M2–M8], we find it notable that only three of the PSs are from game-related venues: Games and Learning Alliance Conference (GALA), ICSE Workshop on Games and Software Engineering (GAS) and the International Conference on the Foundation of Game Development (FGD). Only one venue has multiple publications: IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), and the remainder of the PSs were published in separate venues.

Figure 3.2 illustrates the yearly publication distribution since 2015, ever since the first ECS paper we included. It demonstrates that the yearly ECS publications are currently at their highest and have seen an increase since the 2020s. Table A.1 illustrates that the PSs are mostly conference papers (18), that there are 13 journal articles, as well as four workshop papers.

3.4.2 Research Focus (RQ1.2)

While ECS has gained significant traction within the game development industry, academic literature remains sparse. This study aims to systematically classify existing ECS research, establishing a comprehensive repository that highlights current trends and identifies critical gaps to provide a foundation for future academic inquiry.

Table A.1 and Table 3.3 demonstrate that 10 out of the 35 PSs are directly linked to games. However, most of the PSs acknowledge that ECS is primarily used in game development.

Table 3.3 highlights that 30 of the 35 PSs work on domains that are typically associated with ECS. Real-time applications are the most common use for ECS, covered by 18 of the PSs, six and

Table 3.2: Number of studies per venue

#	**Publication venue**
2	IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)
2	IEEE Access
2	ICSE Workshop on Games and Software Engineering (GAS)
2	International Conference on AI Revolution: Research, Ethics and Society
1	Computer Science On-line Conference (CSOC)
1	ACM Transactions on Human-Computer Interaction
1	ACM Transactions on Graphics
1	ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)
1	Proceedings of the ACM on Programming Languages (PACMPL)
1	Computer Animation and Virtual Worlds
1	Games and Learning Alliance Conference (GALA)
1	Geographical Analysis
1	SIGGRAPH Asia 2024 Educator's Forum
1	IEEE Conference on Technologies for Sustainability (SusTech)
1	IEEE Engineering International Research Conference (EIRCON)
1	IEEE International Conference on Industrial Informatics (INDIN)
1	IEEE International Conference on Software Architecture (ICSA)
1	IEEE International Symposium on Real-Time Distributed Computing (ISORC)
1	IEEE Open Journal of the Industrial Electronics Society
1	IEEE Transactions on Power Systems
1	IEEE Transactions on Visualization and Computer Graphics
1	IFIP Advances in Information and Communication Technology
1	International Conference on Computer Modeling and Simulation
1	International Conference on the Foundations of Digital Games (FDG)
1	International Conference on Power, Electronics and Computer Applications (ICPECA)
1	International Symposium on Distributed Simulation and Real Time Applications
1	International Symposium on Physical Design
1	Physics of Particles and Nuclei
1	Simulation Modelling Practice and Theory
1	Symposium on Maritime Informatics and Robotics (MARIS)

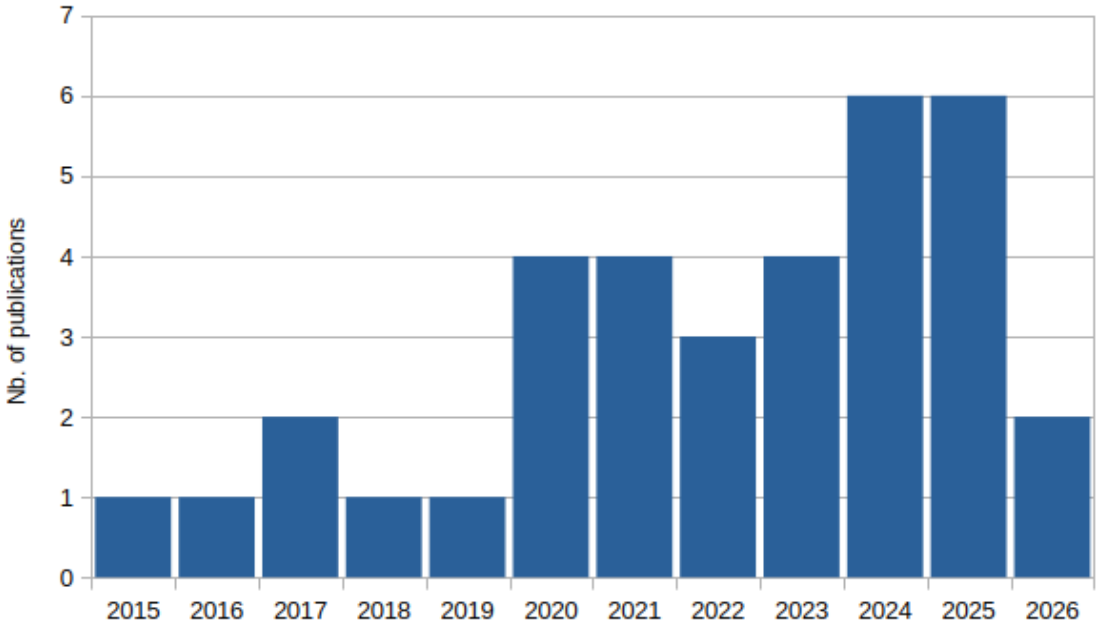

Figure 3.2: Publications per year

four PSs study video games and game engines, respectively, and two PSs are on virtual reality.

Five studies represent significant departures from traditional ECS applications, spanning the domains of machine learning, internet of things (IoT), human-computer interaction (HCI), and even compiler design. Cheng et al. [M9] leveraged the data-oriented design principles of ECS to enable GPU instancing for machine learning training; despite its simulation-based environment, the study is categorised under Machine Learning due to this unique integration. In the domain of IoT, Pouhela et al. [M2] implemented an ECS-based publish/subscribe model for an IoT broker with ECS, which outperformed a standard object-oriented reference in benchmarks. Raffaillac et al. [M6] adapted the ECS model for GUI interaction programming, demonstrating its efficacy in managing complex state changes and user interactions. Casals et al. [M10] proposed utilising the ECS pattern to simplify the engineering of Multi-Agent Systems (MAS) by aligning them with established distributed systems techniques, successfully lowering the specialised knowledge barrier for developers to implement complex agent behaviours like cooperation and task planning. Finally, Dahl et al. [M11] investigated the practicality of using ECS as the primary data structure for compilers and interpreters, demonstrating that reframing AST nodes as entities and attributes as components simplifies optimisation passes through linear iteration and facilitates efficient state serialisation.

Table 3.3: Study Focus

Domain	#	Studies
Real-time Application	18	[M1, M8, M12–M27]
Video Games	6	[M3, M7, M28–M31]
Game Engines	4	[M4, M5, M32, M33]
Virtual Reality	2	[M34, M35]
Machine-Learning	2	[M9, M10]
IoT	1	[M2]
Human-Computer Interaction	1	[M6]
Compiler-Design	1	[M11]

3.4.3 ECS Benefits (RQ1.3)

The primary studies discuss at length the benefits of developing with and using ECS in software engineering. Performance is the most mentioned (26) and most elaborated upon by the primary studies; they mostly attribute the data-oriented design and its natural affinity to parallelise tasks. The studies also bring up positive software design qualities for ECS, notably modularisation (21), maintainability (18), scalability (17), and reusability (16). Table 3.4 identifies which PS cover what benefit.

Performance (26)

As outlined in Chapter 2, the performance benefits of ECS are largely rooted in its application of data-oriented design. Unlike object-oriented design, which typically organises data using an Array-of-Structures format, data-oriented design adopts a Structure-of-Arrays layout. By grouping identical data types together rather than interleaving unrelated attributes, this method optimises CPU cache utilisation and reduces memory access overhead.

Beyond cache efficiency, the literature frequently cites ECS as being naturally conducive to multi-threading. Because data is decoupled from logic, it is easier to partition workloads across multiple CPU cores, allowing independent systems to process large entity counts and complex simulations with high levels of concurrency.

Modularity (21)

As noted in recent studies [M13], the modular nature of ECS is a direct product of its data-oriented foundation, which strictly isolates state (Components) from logic (Systems). Within this framework, Entities function merely as unique IDs that aggregate various Components, while independent Systems execute logic only on Entities that possess a matching data signature. This structure promotes a highly reusable pattern where complex behaviours are ”composed” rather than inherited [M22]. Unlike the rigid hierarchies often found in object-oriented programming, ECS minimises interdependencies, resulting in a more scalable and decoupled codebase.

A core advantage of this modularity is the ability to alter Entity behavior during runtime. By dynamically adding or removing Components, developers can transform an object's properties without side effects on the broader system. This extensibility also applies to the Systems themselves, which can be swapped or updated in isolation. For instance, games can apply a single ECS world to manage a variety of entities, from characters and vehicles to environmental objects, by leveraging reusable components and systems that adapt to different requirements.

Furthermore, this design facilitates plugin-centric development, where specific features are bundled into discrete modules that can be loaded on demand, making ECS an exceptionally modularisable choice for complex game projects [M32].

Maintainability (18)

The structural organisation of the ECS pattern is widely noted by the PSs for enhancing project maintainability, primarily due to the strict isolation of state from logic [M5]. By decoupling these elements, developers can work with simplified data structures and specialised systems, resulting in a codebase composed of granular, manageable functions that are easier to maintain [M35].

In ECS, Components act as data containers with no internal logic, which simplifies their understanding and management. Simultaneously, Systems are restricted to specific data signatures, executing clearly defined operations without the friction of cross-dependencies. This functional isolation significantly boosts code legibility, allowing for more intuitive debugging and a clearer mental model of the software's execution flow.

Scalability (17)

Research by the PSs underscores the scalability advantages of ECS, particularly in high-complexity domains such as power grid management and large-scale game engines [M13]. A primary driver of this scalability is the data-oriented design, which facilitates highly adaptable data structures and extensive component recycling. By prioritising composition over the rigid inheritance hierarchies found in OOP, ECS allows for the expansion of systems without the risk of cascading dependencies. This structural agility is also present during runtime; developers can dynamically inject or extract components and systems to suit evolving simulation needs without compromising the broader environment.

Beyond software architecture, the scalability of ECS is amplified by its alignment with modern hardware. The Structure of Arrays memory organisation inherent in many ECS implementations maximises cache throughput and computational density when processing massive datasets. This hardware-friendly layout makes ECS an ideal candidate for GPU acceleration and high-performance computing tasks [M9, M19]. Consequently, the integration of data-oriented principles and modular design establishes ECS as a versatile foundation for applications demanding both high throughput and long-term adaptability.

Reusability (16)

The primary studies identify that the ECS pattern significantly enhances code reusability through its granular, component-oriented design. By defining components as discrete data structures, such as spatial coordinates, velocity vectors, or acceleration metrics, developers can deploy identical data definitions across diverse entities and separate projects. This paradigm shifts the focus away from code duplication, as common traits no longer require unique implementations for every object type. Consequently, development cycles are shortened by integrating a library of pre-existing, verified components into new environments.

The reusability of the pattern is further amplified by Systems, which operate on data types rather than specific object classes. A standardised system, such as one handling kinematics, remains functional as long as an entity possesses the requisite components, staying agnostic to the entity's

specific purpose. Whether the project is a flight simulator or a video game platformer, the same underlying logic can be repurposed without extensive refactoring. This ability to carry over mature, tested logic between projects significantly optimises development time and reduces the likelihood of introducing new bugs.

Table 3.4: Benefits key terms

Key Term	#	Studies
Performance	26	[M1–M5, M7–M19, M24–M26, M28, M30, M32, M34, M35]
Modularity	21	[M1–M3, M5–M9, M12–M14, M16–M20, M22, M29, M30, M32, M35]
Maintainability	18	[M1–M4, M6–M9, M12, M14, M17, M18, M20, M22, M26, M29, M32, M35]
Scalability	17	[M2–M4, M6, M9, M10, M12–M15, M17, M18, M22, M29, M30, M32, M35]
Reusability	16	[M2–M4, M6, M7, M9, M12, M14, M16–M18, M22, M24, M29, M32, M35]

3.4.4 Drawbacks (RQ1.3)

The primary studies did not mention nearly as many drawbacks as benefits to the ECS pattern, with only nine papers mentioning some. Five PSs mention usability issues related to ECS, especially for new users. Four more PSs identify that ECS has limited applications, spanning mostly real-time applications. One primary study explains the complexity of implementing an ECS and the necessity of mastering lower-level data-oriented concepts. One primary study identifies memory complexity issues related to ECS implementations, and finally, another PS observes maintainability drawbacks related to ECS. Table 3.5 identifies which PS cover what drawback.

Usability (5)

Primary studies note that the adoption of ECS is often hindered by a significant learning curve, primarily because it diverges from the ubiquitous OOP model in favour of a data-driven philosophy [M2,M5,M28]. As Bayliss points out, OOP remains the pedagogical standard in most academic

institutions, whereas data-oriented design lacks similar foundational integration [M28]. Transitioning to ECS is not a simple matter of swapping class definitions; it requires developers to develop an understanding of low-level concepts such as memory spatial locality and CPU cache management to implement data-oriented design principles effectively. Sobolev et al. [M27] mention that ECS is ”not as elegant” as object-oriented approaches when it comes to handling multiple different behaviours from different entities. As the study lacks a controlled user experiment, this claim should be treated as a practitioner-based reflection on the structural overhead of ECS rather than a verified limitation of the pattern. Finally, Raffaillac *et al.* [M6] describe the design freedom of the ECS pattern as a ”double-edged sword.” While it offers immense flexibility in defining high-level logic, this lack of rigid structure can lead to fragmentation in implementation strategies. The authors suggest that further empirical research is required to establish best practices and clearer design guidelines for various use cases.

Limited Applications (4)

Integrating ECS with alternative architectural patterns can introduce substantial technical friction, as noted by Brummett et al. [M12]. A primary challenge lies in reconciling the semantic differences between the data-centric nature of ECS and other communication models, such as publish/subscribe systems. Furthermore, while highly effective in specific domains, ECS is not a universal solution; traditional object-oriented design remains more appropriate for applications where complex behavioural hierarchies and dynamic dispatching are required [M2]. Slay [M8] and Redmond et al. [M1] also point out that empirical studies for ECS remain relatively scarce. This lack of exhaustive analysis is likely due to the pattern’s specialised nature and a broader deficit of peer-reviewed research within the video game industry.

Complexity (1)

The technical overhead associated with ECS is a significant barrier to entry, as noted by [M2]. A successful implementation generally necessitates a deep knowledge of low-level hardware concepts, which can be daunting for those accustomed to higher-level abstractions. Nevertheless, the emergence of robust open-source frameworks can help to mitigate these difficulties.

Memory Overhead (1)

Dahl et al. [M11] mention that on one side, ECS tries to minimise the amount of memory usage through data structures such as Archetypes and Sparse-Sets, but on the other hand, implementing these data structures requires the addition of arbitrary information to solve specific problems. In their case, an additional eight bytes were added for each entity index; This carries over to each new entity allocated into the ECS world.

Maintainability (1)

Contrary to popular belief, Sobolev et al. [M27] highlight that ECS might not be all that great for software maintenance, especially when extending the functionality of an existing simulation. They state that extending said simulation may require creating new systems, components, instantiating new entities, etc., and this makes ECS slower for prototyping than in an object-oriented design pattern. However, it is important to note that the paper seems to base this statement on anecdotal/personal evidence, as it does not validate this claim, nor is it based on prior work that studied this.

Table 3.5: Drawbacks key terms

Key Term	#	Studies
Usability	5	[M2, M5, M6, M27, M28]
Limited Applications	4	[M1, M2, M8, M12]
Complexity	1	[M2]
Memory Overhead	1	[M11]
Maintainability	1	[M27]

3.5 Discussion

3.5.1 RQ1.2

The results confirm that ECS is mainly used in real-time applications, simulations and video game development. However, a small number of the PSs stood out from the rest, using ECS for less conventional practices.

Notably, Cheng et al. [M9] leveraged certain DOD properties of ECS to directly transfer memory to the GPU and execute it on the GPU-accelerated machine-learning model. Video games can and should also leverage these properties to instance thousands, if not millions, of objects onto the GPU. Casals et al. [M10] also used ECS within the context of machine learning. The data-oriented capabilities of ECS encapsulate complex Multi-Agent System (MAS) behaviours, such as task planning, cooperation, and coordination, as modular components within a distributed architecture.

Raffaillac et al. [M6] use ECS to implement a GUI framework. With ECS, they were able to create state-driven interfaces. This could be interesting to translate to the development of turn-based games or other game genres that are not necessarily always running in real-time. Pouhela et al. implement a publish/subscribe messaging model alongside ECS. This mix of design and architectural patterns could be replicated in game development to extend the functionality of ECS to a wide array of game genres. Finally, Dahl et al. [M11] utilised the data-oriented design properties of ECS to reframe compiler architecture by treating entities as AST nodes and components as intermediate representation (IR) attributes. This approach simplifies the implementation of optimisation and analysis passes by transforming traditional recursive tree traversals into straightforward linear iterations, while the reliance on plain old data types enables extremely efficient serialisation of the entire system state to disk.

3.5.2 RQ1.3

The results detailed in the subsection 3.4.3 and subsection 3.4.4 highlight that there exists a research gap in the current understanding ECS design implications. The primary studies predominantly emphasise the benefits of ECS over the downsides, suggesting a positive bias in the field. This opens an opportunity for the research community to discover and establish more drawbacks of the ECS pattern.

RG1

Research gap 1 (RG1): Current research on ECS focuses on benefits above the drawbacks, suggesting a positive bias in the field.

Furthermore, comparative evaluations between ECS and established design patterns, such as

Model-View-Controller (MVC), are necessary to provide a nuanced understanding of its relative strengths. Such benchmarks would enable developers to make more informed technical decisions during the architectural phase of game development.

While game development remains the primary driver of ECS research, its application in diverse domains, including IoT, wildlife modelling, and general simulation, demonstrates its broader utility. Further investigation into these non-gaming contexts may yield cross-disciplinary insights that could be retroactively applied to gaming, as evidenced by the interaction programming advancements in Polyphony [M6].

Finally, the steep learning curve associated with Data-Oriented Design (DOD) remains a barrier to adoption, as a few PSs have mentioned. Future research should prioritise accessibility by developing comprehensive tutorials, standardised best practices, and abstraction tools designed to bridge the gap for developers transitioning from traditional object-oriented programming.

3.5.3 Additional Observation

Most of the primary studies (27), when introducing and describing ECS, approach it with a high-level of abstraction [M2–M5, M9, M10, M12, M13, M15, M16, M18, M20–M35]. This happens mostly when they define common concepts, such as the separation of data and behaviour through components and systems, and how the pattern brings good software qualities such as the ones previously identified in subsection 3.4.3: performance, modularity, maintainability, scalability, and reusability.

However, only a few primary studies (8) [M1, M6–M8, M11, M14, M17, M19] examine and explain the underlying data structures that allow ECS to function properly. Although the majority of the primary studies attribute the performance qualities of ECS to its data-oriented and cache-friendly nature, these claims are often presented at a conceptual level without any detailed empirical evaluation of the backend data structures used in ECS frameworks.

This tendency may be attributed to the fact that ECS originated from the video game industry, where many implementation details are documented through blogs, engine documentation, or open-source projects rather than peer-reviewed academic publications. Consequently, the academic

literature tends to focus on explaining the core ECS pattern concepts rather than analysing its concrete implementation strategies.

RG2

Research gap 2 (RG2): The limited discussion of ECS backend design also suggests an opportunity for further empirical research examining how implementation choices influence performance, memory behaviour, and scalability in real-time systems.

3.6 Summary

The analysis of the primary studies reveals that existing research on Entity-Component-System predominantly focuses on high-level design concepts, such as the organisation of entities, components, and systems, as well as their associated design benefits, while only a few studies touch on their drawbacks. While these aspects are well documented, relatively few studies examine the underlying implementation mechanisms that enable these benefits in practice.

In particular, the results of the mapping study indicate that low-level concerns, such as memory layout, data storage strategies, and iteration efficiency, are rarely analysed in depth. Furthermore, empirical evaluations of how these design choices impact runtime performance and resource usage remain limited. As a result, the relationship between ECS implementation strategies and their practical performance characteristics is not well understood.

This gap highlights the need for empirical studies that investigate ECS at the implementation level, with a focus on how specific design decisions influence both performance and development complexity.

To address this limitation, the following chapter presents a case study examining multiple ECS-based collision-handling implementations within a real-time application context. By evaluating both performance metrics and code complexity, this study provides concrete evidence that complements and extends the findings of the mapping study.

Chapter 4

Impact of archetype-based ECS design decisions in the context of collision handling

This chapter is based on an article submitted to the IEEE Transactions on Games Journal in December 2025. At the time of writing, there have not been any updates to the processing of the submission. In this chapter, we use a simplified version of a survivors-like game made during the span of my Master's program, ECS-Survivors [1].

4.1 Introduction

The systematic mapping study presented in the previous chapter highlighted that existing research on Entity-Component-Systems design pattern primarily focuses on high-level design concepts, with limited attention given to the underlying implementation details that drive performance, identified in research gap 2 (see subsection 3.5.3). In particular, few studies provide empirical evaluations of how different ECS design choices influence real-world performance and code complexity.

To address this gap, this chapter presents an empirical case study examining the impact of

[1] https://github.com/ptidejteam/ecs-survivors

different ECS design decisions in the context of collision handling, a fundamental system in real-time game applications.

4.1.1 Problem

Given the growing adoption of ECS and its prominence in performance-critical applications, there is a need for a systematic study of the impact of different ECS design decisions on runtime performance and code complexity.

This study of the impact of ECS on runtime performance and code complexity requires choosing one or more systems of interest to assess the impact concretely, as well as choosing whether to compare against a baseline or among multiple alternative implementations. Because ECS is already prominent in industry and has been chosen over OOD, we implement and compare several ECS-based collision-handling implementations with one another, rather than comparing ECS to other OOP patterns. Collision-handling can commonly be referred to as collision systems, which can be confusing when mixed with ECS systems. Hence, **systems** will exclusively refer to ECS systems, and we will refer to collision systems as collision-handling implementations.

To reflect conditions typical of industry standards, we base our evaluation on Flecs[2], an ECS framework optimised for large-scale entity iteration and investigate how ECS collision-handling implementations, a crucial feature to video games [12], influence both runtime performance and code complexity.

4.1.2 Research Questions

Our goal is twofold: first, to compare the computational runtime performance of different ECS collision-handling implementations; second, to analyse and document the complexity and challenges of integrating these collision techniques within an ECS. We ask the following research questions:

- **RQ2.1:** How do different ECS collision-handling methods exhibit measurable runtime performance variation?

[2] https://www.flecs.dev/flecs/

- **RQ2.2:** How do the different integrations of these methods introduce code complexity that can be quantified using cognitive and cyclomatic metrics?

Together, these research questions directly address RG2 (see subsection 3.5.3).

4.2 Background

This section introduces the concepts relevant to our study. We first explain ECS and its variants, which underpin modern game development frameworks, like Flecs. We then describe data-oriented design (DOD), the principle guiding ECS implementations for cache-efficient and scalable runtime performance. Finally, we motivate the choice of a survivors-like game as a case study, highlighting its suitability for evaluating ECS due to high entity counts and runtime performance-sensitive gameplay.

4.2.1 Survivors-like Games

Survivors-like games, e.g., Vampire Survivors[3], offer an ideal environment for evaluating the ECS design pattern due to their intrinsic gameplay, which involves a large number of simultaneous on-screen entities (players, enemies, etc.). Vampire Survivors commonly displays hundreds of entities at once, making efficient and scalable entity management essential. This high-density, performance-sensitive behaviour makes the genre particularly well-suited for studying ECS implementations.

The remarkable popularity and critical acclaim of Vampire Survivors further justify selecting this genre for our case study. Despite its humble origins, the game achieved significant recognition, notably winning the Best Game award at the 2023 BAFTA Games Awards. By focusing on this genre, our study targets a scenario that is both performance-sensitive and representative of games commonly developed by small or indie teams, reinforcing the practical relevance of evaluating ECS collision-handling in this context.

[3] Vampire Survivors

4.3 Related Work

This section situates our study within the existing research on the ECS pattern and collision-handling systems in games. We first summarise broader investigations into the ECS pattern and its associated runtime performance characteristics, then highlight studies on collision detection and resolution, and finally discuss benchmarks comparing ECS frameworks. This section shows the current gap in empirical analyses of ECS-based collision-handling implementations, motivating our case study using Flecs.

In situating the exploration of the Entity-Component-System for physics collisions in games, the comprehensive mapping study by Voisard et al. [3] provides context by highlighting existing research trends and identifying areas requiring further investigation. The study reveals that while ECS has gained popularity in the video game industry, academic research on its practical benefits and limitations is nascent. Notably, the study highlights a need for more in-depth analysis of specific game mechanics within the ECS design pattern, including their runtime performance implications. Our investigation into the integration and runtime performance comparison of various collision-handling implementations within the Flecs ECS framework for a performance-intensive survivors-like game contributes to filling this gap. By evaluating the runtime performance of different collision-handling implementations, our study contributes to a more nuanced understanding of the practical runtime performance characteristics of a fundamental game system within an ECS design pattern, an area that needs further empirical studies.

In a recent survey on collision detection implementations for video games, Lazaridis et al. [12] emphasised the central role of collision-handling in gameplay mechanics, where interactions between objects, such as characters, weapons, and terrain, depend on accurately detecting contact events. The study discusses how collision detection affects realism and contributes to game responsiveness and fairness, particularly in fast-paced or physics-intensive scenarios. Building on these observations, our work focuses on how different ECS-based collision implementations behave in practice, providing empirical runtime performance and complexity measurements that complement the conceptual overview provided by Lazaridis et al.

Several benchmarks evaluate the runtime performance of different ECS frameworks, with a notable benchmark being the repository by Beimler [13]. This repository systematically compares various ECS frameworks, highlighting their efficiency in handling entity updates and queries. Among the evaluated frameworks, Flecs consistently demonstrated superior runtime performance, particularly in systems requiring frequent entity queries and dynamic component management. We use this study to base our choice of using Flecs in this work, as its efficiency in handling large-scale simulations can play a crucial role in optimising collision-handling.

4.4 Method

We now describe the method that we used to evaluate different collision-handling approaches within an ECS game. We first introduce the custom test-bed developed for this study, ECS-Survivors, a lightweight game built using the Flecs framework. We then outline the experimental design, detailing the setup, data collection process, and evaluation metrics. Finally, we present and explain the different collision-handling implementations compared in our experiments.

We want to emphasise here that we are not looking to implement the most efficient collision-handling implementation possible, as that topic has already been thoroughly explored. Instead, we focus on how different collision-handling implementations specific to ECS, using the Flecs framework, impact runtime performance and code complexity.

4.4.1 ECS-Survivors

When we started our study, there were no open-source projects that used Flecs and would have allowed us to study collision-handling approaches. Many projects used Flecs; however, they did not make their source code available and, consequently, could not be used for such empirical studies. Therefore, as a first contribution, we develop our own lightweight game using Flecs, which we choose to implement as a survivors-like game, called ECS-Survivors. ECS-Survivors is one of the featured projects on the Flecs homepage and an example of open source projects[4].

[4] https://www.ptidej.net/news/250629%20-%20ecssurvivors

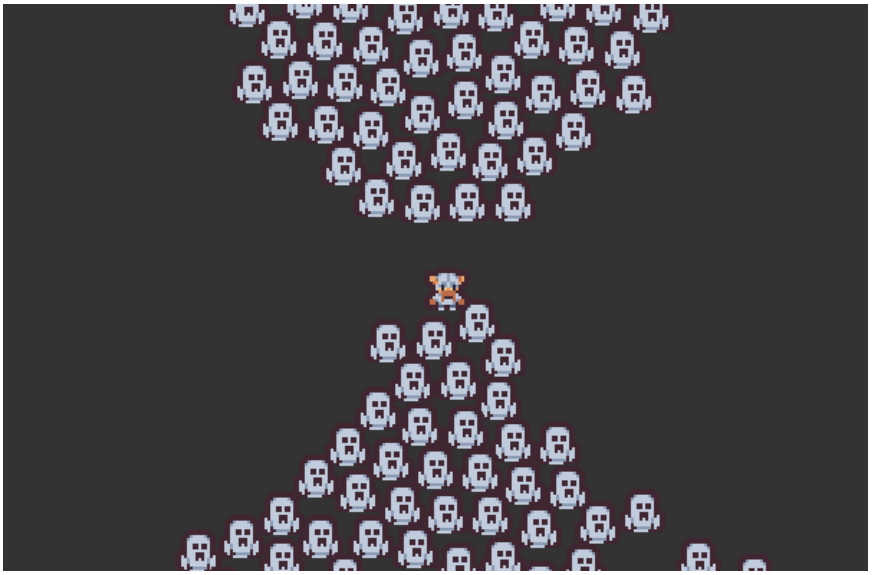

Figure 4.1: Study scenario
Study scenario. The player is chased by ghost enemies.

The source code is available in GitHub[5] under the GPLv3 license. ECS-Survivors is implemented in C++ with the Flecs ECS framework and Raylib[6], a lightweight, code-only, and beginner-friendly game library designed for simplicity and ease of use, which we use for rendering and vector math. We chose Flecs and Raylib because they are not tied to any specific game engine, which increases the generalisability of our study and keeps the focus on evaluating ECS rather than engine-specific features.

For the experiment, which features an altered version of ECS-Survivors, we create an open-source fork[7] to facilitate data collection and remove unnecessary features.

4.4.2 Study Design

Scenario

A player is situated in the middle of the screen as enemies spawn from around the screen, moving towards the player. Enemies are added at a rate of one per frame. Neither the enemies nor the player can be damaged, so each will stay until the target frame rate is reached.

[5] https://github.com/ptidejteam/ecs-survivors
[6] https://www.raylib.com/
[7] https://github.com/LVoisard/ecs-survivors-collision-test

Hardware

We carried out our experiment on a Ryzen 7 5700X3D CPU and an RX 6800 graphics card on Ubuntu 2024.04 at a 1920x1080 resolution. This represents a mid-to-high-end gaming computer.

Collision Implementations

For the experiment, we compare six different ECS collision implementations: *Collision relationships* I1, *Collision non-fragmenting relationships* I2, *collision record entities* I3, *collision record list* I4, *spatial-hash grid (per-cell)* I5, *spatial-hash grid (per-entity)* I6 (see subsection 4.4.4).

Spatial-Hashing Parameters

When choosing a size for the grid cells, it is generally recommended to pick a size to fit the largest entity [14]. In our case, all of the entities are the same size (32x32 pixels), which simplified the decision. The spatial hashing grid also spans the entire window at a resolution of 1920x1080, with an additional two rows and columns of cells around the outside of the screen border, resulting in $(60 + 2) * (34 + 2)$ cells, or 2,232 total cells.

Data Collection

For each ECS collision implementation, we collect and average data over 30 trials. Once the number of frames per second (FPS) drops to below 60, the experiment stops. To avoid stopping the experiment on a frame-time spike, we compute the frame rate by averaging the length of the last 60 frames.

Runtime Measures

At every frame, we record the number of entities, FPS, frame time, and physics frame time. Every trial is saved to a comma-separated file, and later merged with every other trial through a Python script. The total collected data amounts to around 40 MB and takes around 1h30 to collect.

Frame Budget Allocation.

Games require specific allocations of time for tasks such as game logic, rendering, physics, and more [14, 15]. A game that aims to be aesthetically pleasing might allocate more budget to rendering, while a game such as Vampire Survivors will allocate more budget to physics, hence we present the collected results at specific time budgets: 16ms for 100% budget, 8ms for 50%, 4ms for 25%, 2ms for 12.5% and 1ms for 6.25% at 60 frames per second.

Code Measures

In addition to runtime measures, we also use a well-known, industry-used software quality analysis tool, SonarQube [8], to measure the complexity of the source code of the different implementations. We measured both *cyclomatic* [16] and *cognitive* complexities. The former is a measure of the minimum number of test cases required for full test coverage, and the latter is a measure of how difficult the code segment is to understand[9]. We supplement these measures with the number of systems used in an implementation, as well as the number of lines of code.

Compounded Measures

To integrate runtime performance and complexity into a single measure, we computed an *efficiency index* that balances entity throughput with code complexity. The efficiency index is defined as

$$\text{Eff} = \frac{E}{\sqrt{C_c \cdot C_g} \cdot S}$$

where E is the number of entities handled, C_c the cyclomatic complexity, C_g the cognitive complexity, and S the number of ecs systems. This index is useful as a compound measure to compare trade-offs across implementations.

[8] https://www.sonarsource.com/

[9] https://docs.sonarsource.com/sonarqube-server/10.8/user-guide/code-metrics/metrics-definition

4.4.3 Collision Detection and Resolution

The bulk of the collision-handling logic is separated into two phases, implemented as two ECS systems: (*i*) *collision detection* and (*ii*) *collision resolution*, followed by the additional specification of each of the collision implementations.

Collision Detection

The detection process is greatly simplified since the only shapes we used are circles of the same size. When detecting collisions, it is possible to raise the same collision pair (a, b) or (b, a). To avoid this situation, we can leverage one of the core principles of ECS and use the entity IDs. Ensuring that the ID of entity a is strictly less than entity b's ID will ensure we never compute duplicate checks ($a.id < b.id$). We use basic circle collision checks from Ericson [14].

Collision Resolution

In the resolution phase, we iterate over the identified collision pairs and the related information, such as the overlap identified in the previous step. For an identified pair, we move them apart and adjust their velocities.

4.4.4 Collision Implementations

Relationships (I1)

I1 uses the relationship component provided by the Flecs framework. A relationship is assigned to an entity as any other component, but has a second value associated as a pair, the relationship and the target [9]. In this specific implementation, the relationship (in bold) **`CollidedWith`**`(entity)`, and the target `CollidedWith`**`(entity)`**. During the detection phase, when we detect a collision between entities A and B, we add this relationship to both entities (`CollidedWith(A)`, `CollidedWith(B)`). When in the resolution phase, we iterate over all relationship pairs and perform the resolution logic. After the resolution is complete, we remove all of the `CollidedWith` relationships from the entities.

Relationships Non-fragmenting (I2)

I2 works exactly like the above implementation; however, it uses the Flecs `DontFragment` component flag. It treats any relationship (`CollidedWith`, entity) as a unique component, reducing the number of archetypes from n to only one. This reduction lowers the overall fragmentation and reduces the archetype iteration overheads.

Collision entities (I3)

In the collision detection phase of I3, when a collision is detected, a brand new entity is created with a `CollisionRecord` component, which stores the identifiers of the collided entities. In the collision resolution phase, we iterate over all entities with the `CollisionRecord` component and perform the resolution logic. After the resolution is complete, we remove all of the entities with a `CollisionRecord` component from the world.

Record List (I4)

I4 takes advantage of the `Singleton` component flag provided by Flecs[10]. In Flecs, a singleton component is available in the global context, thus making it accessible from any system. In the collision detection phase, we append detected collisions to the `RecordList` singleton component. In the resolution phase, we access the list of records and iterate over them, applying the required resolution logic. Finally, once the resolution is complete, we then clear the list of records.

Spatial Hashing Grid (I5-I6)

Also known as a uniform grid, the spatial-hashing grid is an acceleration technique used to speed up collision detection [14]. Using their spatial coordinates as an identifier, entities can be associated with a cell in the grid. Then, by looking in neighbouring cells, we can access nearby entities and perform collision detections. To detect collisions, we proceed in two ways. The first is by iterating over every grid cell first, which we call per-cell, and the second iterates over all entities, and finds in which cell the current entity is in, which we call per-entity. We explain each variant in more detail

[10] https://www.flecs.dev/flecs/md_docs_2ComponentTraits.html#singleton-trait

below.

Spatial Hashing Grid (Per Cell) (I5)

I5 iterates over each cell and compares all entities from the current cell with the entities in the neighbouring cells. See Algorithm 1.

Input: Grid cells $cells$; each $cell$ has $cell.entities$ and $cell.neighbours$
foreach $cell \in cells$ **do**
  foreach $neighbour \in cell.neighbours$ **do**
    foreach $a \in cell.entities$ **do**
      foreach $b \in neighbour.entities$ **do**
        `// detect collision`
      end
    end
  end
end

Algorithm 1: Spatial hashing grid (per-cell)

Spatial Hashing Grid (Per Entity) (I6)

I6 is similar to a per-cell spatial hashing grid; however, instead of iterating each cell first, it iterates over each entity and checks for collisions with entities in its own cell and neighbouring cells. See Algoritm 2.

Input: Entities with grid coords (x, y); grid cells with neighbour lists
for $a \in entities$ **do**
  $cell \leftarrow getCell(a.x, a.y)$;
  foreach $neighbour \in cell.neighbours$ **do**
    foreach $b \in neighbour.entities$ **do**
      `// detect collision`
    end
  end
end

Algorithm 2: Spatial hashing grid (per-entity)

The ECS collision implementations *collision relationships*, *collision non-fragmenting relationships collision record entities*, and *collision record list* suffer from a worst-case time complexity of $O(n^2)$ because they compare every entity with every other entity in the world. *spatial-hash grid*

(per-cell) and *spatial-hash grid (per-entity)* have a time complexity when detecting collisions of $O(m*(n/m))$ and $O(n*(n/m))$ respectively, with m being the number of cells, and n the number of entities.

4.5 Results

We now apply the evaluation method described in the previous section to the six ECS collision-handling implementations, we perform a static code analysis to assess code complexity and we combine these dimensions using the Analytic Hierarchy Process (AHP) 4.5.3 to derive a comparative ranking and practical recommendations for ECS-based game development.

4.5.1 Experimental Results

Table 4.1 and Table 4.5.1 summarise the performance of the six collision-handling implementations in terms of the maximum number of entities handled under different physics frame-time budgets (16 ms down to 1 ms). Implementations I1-I4 exhibit quadratic complexity ($O(n^2)$) and scale poorly as entity counts increase. Among these, I2-I4 perform nearly identically across all time budgets, significantly outperforming I1, which consistently handles the fewest entities.

Table 4.1: Entities handled at specific physics time allocation
Entities handled at specific physics time allocation. I5 at 2 and 1ms did not yield significant entity counts.

Impl.	16 ms	8 ms	4ms	2ms	1ms
I1	593.8 ± 10.3	418.7 ± 9.3	296.1 ± 9.1	205.6 ± 11.7	152.7 ± 8.3
I2	755.2 ± 10.5	540.5 ± 8.8	389.4 ± 5.9	279.9 ± 6.9	196.0 ± 7.1
I3	758.9 ± 5.2	544.3 ± 3.4	391.8 ± 4.6	280.8 ± 7.5	192.6 ± 8.4
I4	775.9 ± 7.2	556.4 ± 5.6	400.9 ± 4.1	285.4 ± 8.4	198.8 ± 8.9
I5	6713.2 ± 209.0	3141.8 ± 82.3	984.3 ± 21.8	-	-
I6	5491.2 ± 131.9	3115.3 ± 45.9	1849.3 ± 21.9	1122.0 ± 26.9	622.9 ± 32.1

In contrast, the spatial hashing approaches demonstrate a substantial improvement. The per-cell spatial hash handles almost an order of magnitude more entities at larger time budgets (6,713.2 entities at 16 ms). However, its runtime performance rapidly declines under tighter constraints; at 2 and 1ms, it did not yield significant entity counts. Conversely, the per-entity spatial hash achieves better runtime performance at intermediate time budgets (e.g., 1849.3 entities at 4 ms vs. 984.3

entities). Still, it is surpassed by the per-cell implementation when entity counts become very large.

RQ2.1

Simpler implementations I1-4 scale quadratically and perform poorly as entity counts grow. The introduction of the `DontFragment` trait allows I2 to perform significantly better than I1, its fragmenting counterpart. Spatial hashing greatly improves scalability: the per-entity variant is best at low to medium entity counts, while the per-cell variant is best at high entity counts but fails under tight frame budgets.

4.5.2 Static Code Analysis Results

Table 4.2: Implementation complexity measured by cyclomatic and cognitive complexity and system count.

Impl.	Cyclo. Comp.	Cogn. Comp.	System Count	LOC
I1	9	8	3	63
I2	9	8	3	63
I3	9	8	3	71
I4	13	14	4	66
I5	23	38	8	135
I6	23	32	8	139

Table 4.2 shows the code complexity of each implementation, as well as the number of systems and lines of code. The simpler implementations (I1, I2, I3, I4) have relatively low complexity, with cyclomatic complexity ranging from 9 to 13 and cognitive complexity from 8 to 14. The I1, I2, and I3 implementations are the simplest (9 cyclo., 8 cogn., 3 systems), while I4 adds modest complexity due to the use of a global singleton component.

The spatial hashing implementations (I5-I6) are significantly more complex. Both I5 and I6 variants require 8 systems and more than 130 LOC, with cyclomatic complexity reaching 23 for both and cognitive complexity at 38 and 32, respectively. This higher complexity reflects the need to manage grid structures, neighbour lookups, and iteration across multiple entities and cells.

RQ2.2

Implementations I1 through I4 are simple, requiring fewer systems and lines of code, while spatial hashing I5 and I6 are more complex due to grid management and neighbour lookups. Cyclomatic and cognitive complexity metrics reflect this difference. Overall, simpler methods offer easier implementation, whereas spatial hashing trades complexity for scalability.

Table 4.3: Efficiency index at various physics time budgets. Higher is better.

Impl.	**1 ms**	**2 ms**	**4ms**	**8ms**	**16ms**
I1	6.00	8.08	11.63	16.45	23.33
I2	7.70	11.00	15.30	21.23	29.68
I3	7.57	11.03	15.39	21.38	29.81
I4	3.68	5.29	7.43	10.31	14.38
I5	-	-	4.26	13.58	29.02
I6	2.93	5.29	8.71	14.68	25.87

Table 4.3 shows the efficiency of each implementation at different time budgets, using the results from Table 4.1 and Table 4.2. I2 and I3 implementations achieve the highest efficiency index (29.68 and 29.81, respectively), closely followed by spatial-hashing-per-cell (29.02). The per-entity variant has an index value of 25.87, relationships implementation at 23.33, while the record list scores lowest (14.38).

Hence, the experimental results show a clear runtime performance hierarchy: I5 and I6 outperform all other approaches, followed by I2, I3, and I4, which perform similarly, while I1 consistently yields the least efficient results. A closer break-even analysis reveals that per-entity hashing is more efficient at small to medium scales, whereas per-cell hashing becomes dominant when handling very large numbers of entities. In terms of implementation effort, I1, I2, I3, and I4 are easier to develop and maintain, requiring fewer systems and less code, while (I5-I6) introduces substantially more complexity due to grid management and neighbour lookups. Taken together, the best trade-offs suggest that the record-entity method is a practical choice for developers seeking simplicity, while per-cell spatial hashing is the most suitable solution when scalability is the primary concern, despite its higher implementation cost.

4.5.3 AHP Recommendation

To support developers in selecting the most appropriate collision-handling strategy, we apply the Analytic Hierarchy Process (AHP) [17, 18], a structured decision-making technique that evaluates alternatives based on multiple criteria. In our case, the criteria are **performance** and **complexity**, reflecting common trade-offs in ECS collision-handling design.

We simulate three developer scenarios:

(1) **Scenario 1 (S1)**: Performance is three times more important than complexity ($\lambda = 3$).

(2) **Scenario 2 (S2)**: Complexity is three times more important than performance ($\lambda = 1/3$).

(3) **Scenario 3 (S3)**: Performance and complexity are equally important ($\lambda = 1$).

The corresponding criteria pairwise table (item 4.5.3) encodes these preferences.

Table 4.4: Pairwise comparison of criteria (Si)

Criteria	**Performance**	**Complexity**
Performance	1	$1/\lambda$
Complexity	λ	1

We then compare the six collision-handling implementations, I1 through I6, pairwise under each criterion, as shown in Table 4.5.3 and Table 4.5.3, with 2.6% and 0.6%, respectively, for consistency ratios (CR). In these tables, values greater than 1 indicate a preference of the row implementation over the column implementation for the given criterion.

Table 4.5: Pairwise comparison of methods under the *performance* criteria

Implementation	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**
I1	1	1/2	1/2	1/2	1/8	1/7
I2	2	1	1	1/2	1/7	1/6
I3	2	1	1	1	1/7	1/6
I4	2	1	1	1	1/7	1/6
I5	8	7	7	7	1	2
I6	7	6	6	6	1/2	1

Finally, we calculate weighted scores for each implementation according to the criteria preferences (Table 4.5.3) and highlight the highest scoring alternatives per scenario in green, the second

Table 4.6: Pairwise comparison of methods under the *complexity* criteria

Implementation	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**
I1	1	1	1	2	5	5
I2	1	1	1	2	5	5
I3	1	1	1	2	5	5
I4	1/2	1/2	1/2	1	4	4
I5	1/5	1/5	1/5	1/4	1	1
I6	1/5	1/5	1/5	1/4	1	1

highest scoring alternative in yellow, and the worst scoring alternative in red.

Table 4.7: AHP scored alternatives based on the criteria comparisons

Implementation	**S1**	**S2**	**S3**
I1	0.093	0.199	0.146
I2	0.112	0.205	0.158
I3	0.112	0.205	0.158
I4	0.086	0.139	0.107
I5	0.346	0.147	0.246
I6	0.253	0.116	0.184

AHP Recommendation

Based on the AHP-derived rankings, we summarise the most suitable collision-handling implementations for each developer scenario:

- When **performance** is prioritised (Scenario 1), **Spatial-Hashing per-cell (I5)** achieves the highest score (0.346), followed by **Spatial-Hashing per-entity (I6)** (0.253). This confirms our earlier observations that per-cell spatial hashing scales best with high entity counts.
- When **low complexity** is prioritised (Scenario 2), **Collision-Entity (I3)** and **Relationships Non-Fragmenting (I2)** score highest (0.205), closely followed by **Relationships (I1)** (0.199). These implementations are simpler to understand, modify, and maintain, making them suitable for small projects or rapid prototyping.
- When **performance and complexity are equally important** (Scenario 3), the highest performing method **Spatial-Hashing per-cell (I5)** remains the top choice (0.246), followed by **Spatial-Hashing per-entity (I6)** (0.184). This observation suggests that even when balanced criteria are applied, spatial hashing offers the best overall trade-off between speed and maintainability.

4.6 Threats to Validity

This section outlines the primary threats to the validity of our findings and discusses the limitations that may affect their generalisability and usability by practitioners. We identify potential methodological simplifications and framework dependence, and suggest directions for future work to mitigate these limitations.

This study focuses exclusively on ECS-based collision-handling and compares several with one another. Another study could compare these implementations against traditional component-based and inheritance-based implementations. Such a comparison would provide a broader understanding of how data-oriented designs differ from object-oriented ones in real-world performance. However, implementing and benchmarking an OOP component-based counterpart to the ECS one was beyond

the scope of this work. Our findings should be interpreted as comparative only within the ECS pattern, rather than across different common OOP models. Future work will perform a similar study with the Unity Dots ECS and compare it to a traditional Unity component-based program.

This research only involved simple circle shape collisions of the same size. While this simplified the implementation of our collision detection and resolution methods and helped choose the parameters for the spatial-hashing grid, it does not fully represent the diversity of shapes, sizes, and interaction complexities found in most video games or real-time simulations. Consequently, the runtime performance characteristics observed here could differ when applied to more complex geometries, irregular shapes, or varying entity sizes. Future work will extend this study to include diverse collision shapes and sizes, enabling a more comprehensive assessment of ECS-based collision-handling implementations under realistic gameplay scenarios.

Most scalable physics simulations separate collision detection into broad and narrow phases, making our first three implementations, suffering from quadratic complexity, undesirable for very high entity count situations. However, since our goal is to address the lack of learning resources for ECS beginners, it is useful to start with simpler implementations before moving on to more complex and efficient ones, as we did with the spatial-hashing implementations. However, survivors-like games might not even reach as many entities as some of the methods reached. Vampire Survivors even caps the number of enemies that can spawn[11] at 300, only allowing bosses and map events after that number. Thus, even I1-4 could be sufficient depending on the physics allocated time budget. Using the principles learned in this case study, future research should extend these lessons to other acceleration structures and optimisations.

This study does not consider multithreading. While parallelisation could significantly improve the runtime performance of the collision implementations, our focus was on understanding implementation trade-offs in archetype-based ECS frameworks. We acknowledge that ECS is well-suited to parallel computing, but exploring this aspect was beyond the scope of the present study. Nevertheless, the insights gained here could inform future work on designing multithreaded and accelerated collision detection and resolution systems.

We performed our experiment strictly on the Flecs ECS framework. As we identified in the

[11] https://vampire-survivors.fandom.com/wiki/Enemies

section 4.3, a GitHub repository empirically [13] conducts runtime performance tests on multiple ECS frameworks and identifies that Flecs consistently outperformed the other frameworks at large entity counts. While this justifies our use of Flecs in this context, our work mostly applies to archetype-based ECS frameworks; thus, some of the insights shared might not properly apply to each case. Future work will extend this study to different ECS frameworks to fill this gap.

4.7 Discussion

We discuss now key factors influencing runtime performance and scalability, including archetype fragmentation and spatial-hashing strategies, and analyse how these relate to development effort and architectural complexity in the context of our study.

4.7.1 Archetype Fragmentation

The relationship implementation performed much worse than the other three naive implementations, which can be mostly attributed to table fragmentation. Flecs describes fragmentation[12] as a property in archetypes where entities can be spread out over multiple tables as the number of components (and combinations thereof) increases. This overhead is mostly present in table creations and queries, the latter needing to match and iterate over more tables [9]. For example, relationship `CollidedWith(A)` is different from `CollidedWith(B)`. In other words, there can be up to n different relationship components, each belonging to its own archetype, which results in many archetypes that contain very few, if any, entities, ultimately leading to a large amount of fragmentation and an increased overhead when iterating over the entities.

Flecs recently introduced a component trait called `DontFragment`, which stores the components in a sparse set without having to create multiple archetypes that would contain just a few entities. This is perfect for this specific scenario, where relationships would have created many archetypes containing just a few entities. The `DontFragment` relationships lower the overhead of iterating over all of the archetypes, resulting in better runtime performance. This is confirmed from

[12] https://www.flecs.dev/flecs/md_docs_2Relationships.html#fragmentation

our results, at all frame budgets, non-fragmenting relationships handle around 25% increased entities. Figure 2.4 shows an example of what the non-fragmenting relationships look like in storage.

4.7.2 Spatial-hash per-cell vs. Per-entity

The two spatial-hashing variants exhibited very different runtime performance profiles across the simulation. The per-cell variant failed to meet the 1 ms and 2 ms frame-time budgets, while the per-entity variant consistently outperformed all other implementations. This result is somewhat surprising, as most physics simulations that employ spatial hashing grids typically rely on a per-cell design. Two factors help explain this discrepancy. First, per-cell approaches are generally more amenable to multithreading, which is how they achieve peak efficiency in large-scale simulations. Second, these simulations often assume a maximum number of entities that are relatively uniformly distributed across the grid, conditions under which per-cell hashing outperforms per-entity methods.

In our case, the grid contained 2,232 cells. For the per-cell variant, this guarantees a minimum of 2,232 × 9 cell and neighbour accesses, even if only a few entities are present. By contrast, the per-entity variant only accesses the 9 neighbouring cells of each occupied cell, reducing the minimum operations to N × 9, where N is the number of entities. This explains its superior runtime performance at lower entity counts. However, at higher counts, the per-cell method begins to close the gap. Its data-friendly memory layout results in a lower L1 cache miss rate (Figure 4.3). The per-cell variant had a 2% miss rate over the entire experiment, while the per-entity version had a 3.2 % miss rate. Because entities in each cell are stored contiguously, iteration becomes more efficient once occupancy increases. Theoretically, the crossover point where per-cell becomes faster should occur when the number of entities roughly equals the number of cells, assuming uniform distribution. Due to uneven distributions of entities in our simulation, the threshold was closer to 3,100 entities, observed at 8 ms frame time (per-entity: 3,112; per-cell: 3,135) rather than 2,232.

These findings suggest practical design trade-offs. For physics simulations with consistently high and uniformly distributed entity counts, the per-cell variant is preferable. In contrast, for gameplay scenarios like Vampire Survivors, where entity counts are large but not maximised, the per-entity approach is likely better suited. An adaptive strategy that dynamically switches between per-entity and per-cell variants when most appropriate could combine the strengths of both methods

without incurring excessive complexity.

4.7.3 Performance vs. Development effort/Complexity

The experiment we conducted revealed that each implementation varied in runtime performance and development complexity. This echoes concerns raised by Ericson [14], who emphasises key aspects when designing collision-handling implementations, among others: runtime performance and ease of use. On the runtime performance side, games targeting smooth visuals at 60 FPS have a strict frame budget of 16.7 ms per update, of which collision-handling may consume only 2–5 ms. This time allocation implies that each collision query has to execute within microseconds, and must also avoid worst-case slowdowns that can produce noticeable frame drops. Our experiments reflect this concern: while naive ECS implementations scale poorly with $O(n^2)$ complexity, the spatial hashing variants maintain higher throughput and more predictable timings, especially at larger entity counts.

Equally relevant is the ease-of-use dimension. Ericson notes that implementation time, maintainability, and the number of special cases or tweaking variables are critical in game development [14], which often proceeds under strict deadlines. This aligns with our static code analysis: the naive ECS methods required fewer systems and less code, making them easier to prototype and maintain, even if they scale less effectively. In contrast, spatial hashing achieves much better runtime performance but at the cost of higher system count, code complexity, and additional data-structure management. This trade-off highlights that the choice of collision implementation is not purely about runtime efficiency, but must also consider development productivity, debugging effort, and the adaptability of the system throughout the game's lifecycle.

4.8 Summary

We now conclude this chapter by summarising our key findings and their implications for ECS-based game development. We reflect on how the results contribute to understanding the trade-offs between runtime performance and complexity in implementing collision-handling approaches within the ECS design pattern for game development.

In the previous chapter, we identified research on the Entity-Component-Systems design pattern primarily focuses on high-level design concepts, with limited attention given to the underlying implementation details that drive runtime performance (see subsection 3.5.3). This gap underscores the need for practical evaluations of different ECS design variants along with a better understanding of their respective runtime performance and code complexity implications.

Thus, we presented a study on the implementations of various ECS collision-handling methods, with an emphasis on runtime performance and code complexity in a performance-intensive survivors-like game using the Flecs ECS framework.

We implemented and compared six ECS collision-handling implementations: relationship, relationship non-fragmenting, collision entity, collision record list, and spatial hashing with per-cell and per-entity variants. We conducted our experiment in a basic survivors-like game scenario, measuring the number of entities and the physics time for each implementation across 30 trials.

Our study yields several contributions toward understanding how the ECS design pattern can be effectively applied to performance-intensive applications, highlighting the practical trade-offs in terms of runtime performance and code complexity between different collision-handling implementations:

- We provide a public game repository as well as a public fork containing all ECS collision-handling implementations, performance benchmarks, and analysis scripts, offering a reproducible foundation for future research and practical experimentation in ECS-based game development.

- We observed that spatial partitioning implementations achieved significantly better performance than the other implementations, followed by the non-fragmenting relationships, record-list and collision-entity methods, and finally the relationship method. In addition, the spatial-hashing per-cell variation performed worse at lower entity counts than the per-entity variation, but it performed better in larger entity counts than its per-entity counterpart.

- We discussed potential reasons for the observed runtime performance differences through fragmentation in ECS relationship-based approaches, differences in spatial hashing variants, and runtime performance vs. complexity metrics.

- We performed a static code analysis on each of the collision implementations using SonarQube. The analysis revealed that the naive implementations (relationships, collision-entity, record-list) were much less complex than the spatial-hashing methods. Our efficiency index revealed that relationships, non-fragmenting, collision entities, and spatial-hashing (per-cell) all shared a similar runtime performance per complexity output at high entity counts.
- We performed an AHP to determine the best recommendations for developers and researchers for three different scenarios: (S1) performance preferred over lower complexity, S2 lower complexity preferred over performance, and (S3) performance and lower complexity equally preferred. The AHP reveals that spatial-hashing per-cell (I5) is best for S1, collision relationships non-fragmenting (I2), and collision-entity (I3) is best for S2, and spatial-hashing per-cell (I5) is best for S3.

Our findings offer practical considerations for developers choosing between various ECS implementations in ECS-based games, particularly for performance-sensitive genres, like survivors-like games. We highlighted the runtime performance implications of different integration strategies and provided a relative measure of code complexity.

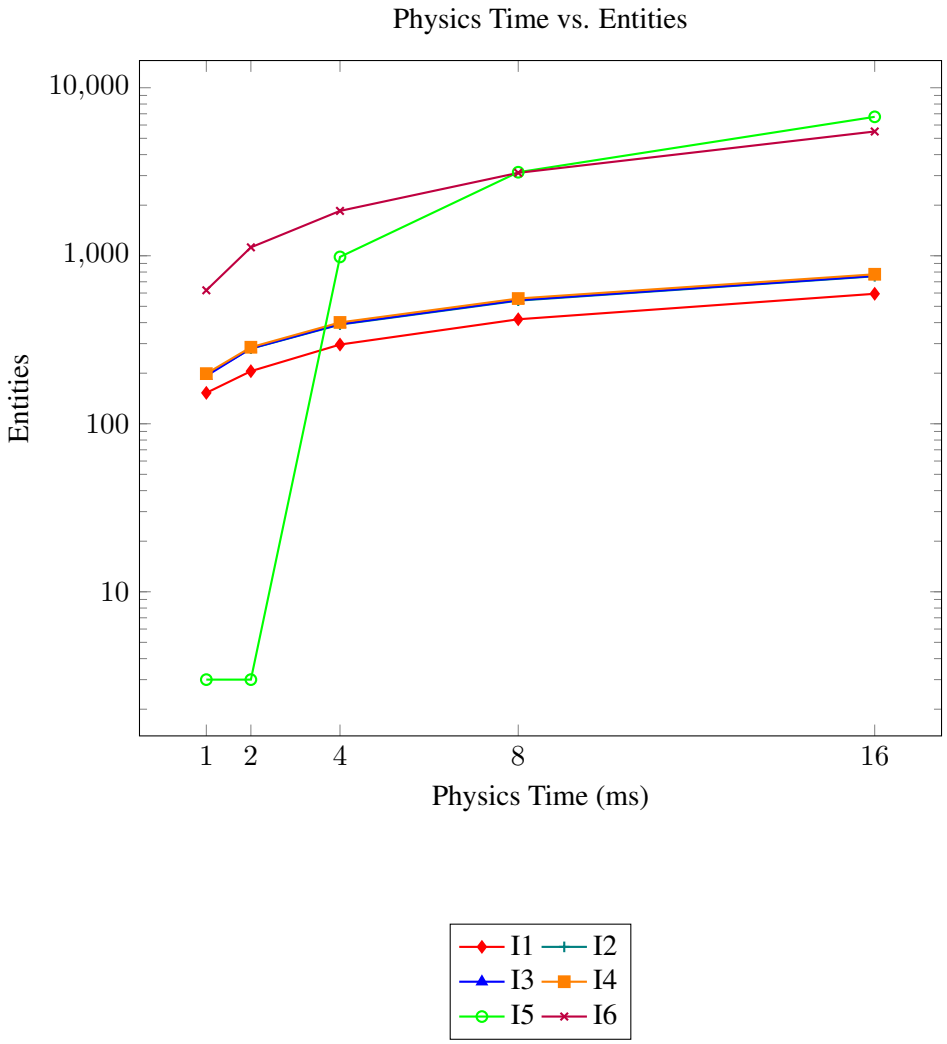

Figure 4.2: Physics time in ms against the number of entities in the world. I2, I3, and I4 perform almost the same; relationship performs the poorest, spatial-hashing-per-entity is fastest to start, but falls short of spatial-hashing-per-cell, which is the fastest when there are the most entities, but has an inherent runtime performance cost.

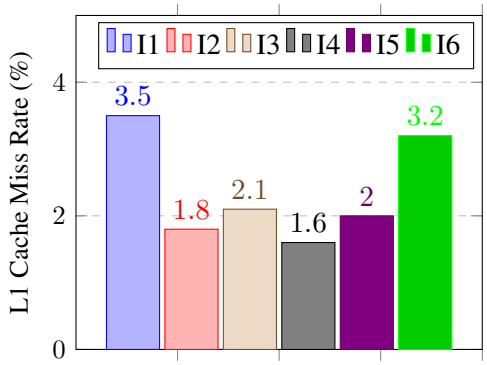

Figure 4.3: L1 cache miss rates for per-cell and per-entity spatial hashing variants.

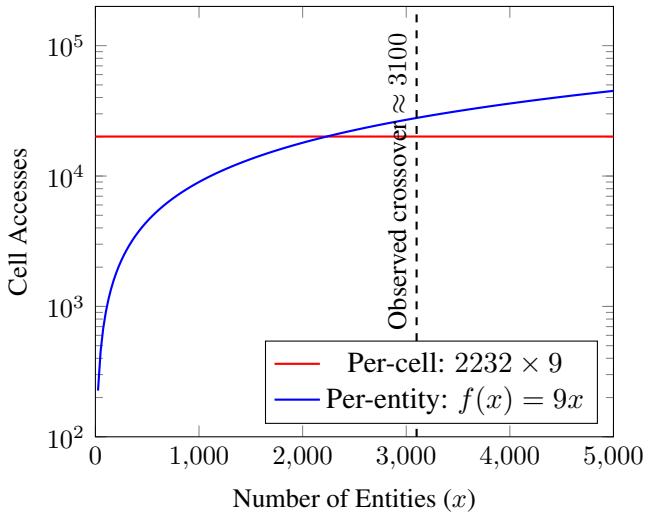

Figure 4.4: Comparison of theoretical cell accesses for per-cell and per-entity spatial hashing variants with observed crossover point marked.

Chapter 5

General Discussion

This chapter synthesises the technical background on archetype-based ECS (Chapter 2), the findings from the systematic mapping study (Chapter 3), and the empirical case study on collision-handling (Chapter 4). It reflects on the broader implications of this work and its contributions to both research and practice.

5.1 Bridging the Theory and the Practice

The Entity-Component-System pattern has gained widespread adoption across the game development industry, yet academic research has largely remained at a conceptual level and lacks quantitative, empirical evaluations. The mapping study in Chapter 3 revealed that most existing work focuses on conceptual descriptions, reported benefits, and framework-specific implementations, with limited empirical investigation of implementation-level trade-offs. Chapter 2 addressed this by providing a detailed examination of archetype-based storage mechanics, particularly structural changes, archetype graphs, and archetype fragmentation, which are often opaque to developers using high-level APIs.

Building on Chapter 2, the empirical case study in Chapter 4 examined how these theoretical concepts manifest in practice. By implementing six different collision-handling strategies within the same non-trivial application (ECS-Survivors), the study demonstrated that design decisions which

appear minor at the architectural level—such as the use of relationships versus spatial partitioning—can produce substantial differences in runtime performance and code complexity.

This thesis thus bridges a critical gap (RG2): it connects the high-level enthusiasm documented in the literature with concrete, measurable consequences of archetype-based design choices. The results confirm that while archetype-based ECS excels at cache-friendly iteration, it introduces non-trivial challenges related to structural changes and fragmentation that developers must actively manage.

5.2 Samples, Best Practices, and Guidelines

As highlighted in the mapping study [4,19–21], the ECS pattern can be challenging to learn, particularly for developers coming from object-oriented backgrounds. The absence of rigid standards and the variety of implementation strategies further complicate adoption. Our six collision implementations in Chapter 4 illustrate this "double-edged sword" of flexibility: while it enables tailored solutions, it also makes identifying optimal approaches difficult without empirical guidance.

This work contributes toward filling RG2 by offering concrete examples and analysis rather than abstract recommendations. The case study shows that there is no universal best implementation. Instead, choices depend on priorities such as performance constraints and code complexity. The Analytic Hierarchy Process (AHP) analysis provides a practical decision-making tool for navigating these trade-offs. While the findings are grounded in collision detection for survivors-like games, the underlying principles of managing archetype fragmentation, balancing structural changes against iteration performance, and quantifying complexity apply more broadly to performance-sensitive ECS applications.

5.3 ECS Survivors

To further support the community and address the documented learning curve, we developed and openly documented ECS-Survivors, an ECS-based game built with Flecs. This project was intentionally designed to push the boundaries of "pure" ECS usage, deliberately favouring ECS-driven solutions over more convenient approaches in other programming patterns and paradigms.

The development process was shared through a series of blog posts (Table 5.1) that emphasise not only the final implementation but the reasoning and challenges encountered along the way.

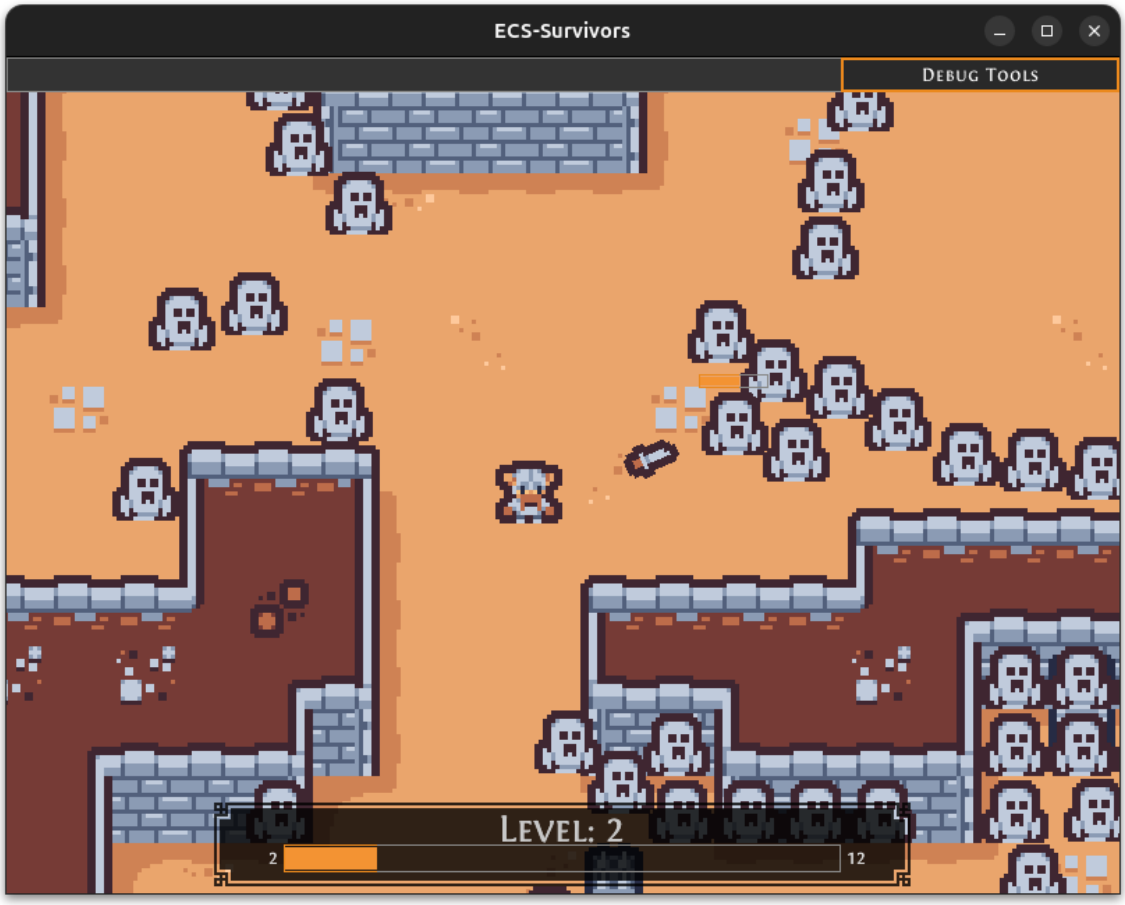

Figure 5.1: ECS-Survivors

The project has since been featured on the official Flecs website [1], indicating its value as a realistic reference implementation. It serves both as a research artefact for studying ECS behaviour and as an educational resource for developers seeking deeper insight into the ECS pattern.

5.4 Future of the Entity-Component-System Pattern

During the development of *ECS-Survivors* and our experimentation with the Entity-Component-System pattern, we encountered several practical challenges when designing software features.

[1] https://www.flecs.dev/flecs/#ecs-survivors

Part #	Link to the post
Part i	https://blog.ptidej.net/ecs-survivors-part-i/
Part ii	https://blog.ptidej.net/ecs/
Part iii	https://blog.ptidej.net/ecs-game/
Part iv	https://blog.ptidej.net/ecs-survivors-part-iv/
Part v	https://blog.ptidej.net/ecs-survivors-part-v-gameplay/
Part vi	https://blog.ptidej.net/ecs-survivors-part-vi-code-refactor/
Part vii-x	https://blog.ptidej.net/ecs-survivors-parts-vii-x/

Table 5.1: Development blog posts for ECS-Survivors.

Many of these challenges align with existing academic concerns identified in our systematic mapping study. In this section, we document some unresolved problems faced during our development lifecycle and propose concrete avenues of research for future researchers and practitioners to explore.

5.4.1 Static Code Analysis

The steep learning curve associated with the ECS pattern remains its most widely recognised and critical barrier to adoption. Current literature suggests that the development of architectural best practices and pedagogical guides will help mitigate this hurdle.

However, during the development of *ECS-Survivors*, we observed that contemporary static analysis tools are largely inadequate for evaluating ECS-based software. Because the ECS pattern relies fundamentally on composition rather than object-oriented class hierarchies, traditional structural analysis reveals little more than a flat topology of independent data structures.

To address this tooling gap, we envision two primary avenues for the development of ECS-specific static analysis tools.

OOP-to-ECS Migration. The first avenue concerns automated migration from object-oriented architectures to ECS-based architectures. Such a tool could analyse an existing class hierarchy, identify shared data and behaviour, and automatically extract them into components and systems.

Although this idea remains conceptual, prior work demonstrates promising foundations. Using a technique known as Formal Concept Analysis (FCA) [22], Llansó et al. [23] proposed a method for converting object-oriented class hierarchies into a component-based architecture. Their approach

extracts common features across classes and groups them into reusable components. Extending this methodology to ECS would require an additional transformation stage capable of extracting behavioural logic from class methods and translating it into ECS systems.

Such a tool could provide several benefits. First, it could assist developers transitioning from object-oriented paradigms by explicitly illustrating how traditional class hierarchies map onto ECS concepts. This may help reduce the learning curve frequently associated with ECS adoption. Second, practitioners maintaining large legacy codebases could use such tooling to partially automate architectural migration toward ECS, reducing the cost and risk associated with manual refactoring.

Accidental System Matching. The second avenue concerns improving the observability and understanding of behaviour propagation in ECS applications. One commonly reported strength of ECS is its promotion of low coupling through the decomposition of behaviour into small, highly specialised systems. During our own development process, we observed that once a system was implemented, its internal logic rarely required modification. While this modularity is generally beneficial, it can also introduce new challenges related to behavioural predictability.

In particular, we observed a recurring issue where entities unintentionally became eligible for processing by unrelated systems, which we call the "accidental system matching problem". This problem occurs when a developer adds a component to an entity with the intention of enabling behaviour associated with a specific system, but the addition also causes the entity to satisfy the requirements of other unrelated systems.

For example, during the development of the modular upgrade system in ECS-Survivors, projectiles initially used a `NoEffectProjectileCollided` system that handled basic collision logic and deleted the projectile after impact. When we introduced new effects through components, such as `Chain`, `Pierce`, and `Split`, a new `ChainProjectileCollided` system was created.

Because the two systems required the same subset of the components present on these projectiles, both the `ChainProjectileCollided` and the original system began matching the same entities. As a result, projectiles with chaining effects were processed by multiple collision systems simultaneously, causing the execution of the `NoEffectProjectileCollided`, unexpectedly deleting the projectile. Because ECS systems are often developed independently and are loosely

coupled, these emergent interactions can be difficult to predict and debug.

This issue is further amplified by the archetype graph structure inherent to archetype-based ECS frameworks. Since each component combination corresponds to a unique archetype, adding or removing a single component may move an entity into a different region of the graph, thereby changing the set of systems that process it. While this dynamic behaviour contributes to ECS flexibility, it can also obscure the relationship between structural modifications and runtime behaviour.

Future tooling could help address this issue by statically visualising system membership across the archetype graph. For example, a developer could inspect which systems would process an entity before and after adding or removing a component. Such tooling could provide warnings about unintended behavioural changes and significantly improve debugging, maintainability, and architectural comprehension in large ECS applications.

5.4.2 CPU–GPU Unified ECS

Modern software increasingly relies on Graphics Processing Units (GPUs) to execute highly parallel workloads. This trend is particularly relevant for the ECS pattern due to its data-oriented structure and emphasis on contiguous memory layouts.

Shacklett et al. [24] proposed an ECS framework executing entirely on the GPU and capable of running a large number of simulations concurrently. Their framework demonstrates the strong compatibility between the ECS pattern and massively parallel hardware. However, its primary objective is large-scale AI training and simulation rather than real-time game development.

One limitation of their approach is the lack of runtime dynamism. The framework computes execution pipelines and allocates component buffers during initialisation, after which structural changes are no longer permitted. While this design maximises performance, it sacrifices one of the defining characteristics of the ECS pattern in games: the ability to dynamically create, destroy, and modify entities and components during execution.

For this reason, we envision future ECS frameworks adopting a unified CPU-GPU execution model capable of supporting both high performance and runtime flexibility. In such a system, developers could write systems using a high-level programming language, after which the framework

would automatically generate equivalent GPU-compatible code through a code-generation or compilation stage. Developers could additionally specify whether a system should execute on the CPU or GPU, depending on workload characteristics and synchronisation requirements.

The most significant challenge in such a framework would likely be the scheduler responsible for coordinating execution across heterogeneous hardware. Systems executing on the CPU and GPU would require efficient synchronisation mechanisms, dependency tracking, and dynamic workload distribution. Fortunately, much of the theoretical groundwork regarding ECS concurrency and scheduling has already been explored by Redmond et al. [2].

A successful unified CPU–GPU ECS framework could significantly expand the applicability of the ECS pattern, enabling highly dynamic simulations while preserving the scalability advantages of GPU computation.

Chapter 6

Conclusion

6.1 Summary

Despite the growing adoption of ECS in performance-sensitive applications, existing academic literature primarily focuses on high-level architectural concepts and reported benefits. Comparatively few studies empirically evaluate how ECS design decisions influence runtime performance and development complexity in practice.

Hence, this thesis investigated the Entity-Component-System design pattern from the combination of a systematic mapping study and an empirical case study, to improve the current understanding of how the ECS pattern design decisions impact runtime performance and code complexity in practice.

To provide the necessary technical foundation for this analysis, in Chapter 2, this thesis examined ECS storage models in detail, going over sparse-sets and focusing deeper on archetype-based implementations. This analysis highlighted key trade-offs between these approaches, particularly the balance between efficient iteration through contiguous memory layouts and the cost of structural changes due to data migration between archetypes. It also identified archetype fragmentation as a critical factor that can negatively impact performance in dynamic scenarios.

First, we conducted a systematic mapping study to analyse existing research on ECS within the software engineering literature. The study asked three research questions:

- **RQ1.1**: In software engineering, what are the **publication trends** on ECS?

- **RQ1.2**: In software engineering, what is the **research focus** on ECS?
- **RQ1.3**: In software engineering, what are the main **benefits and drawbacks** of using ECS?

To answer them, we performed a search in the most prevalent software engineering databases (ACM Digital Library, IEEE Explore, and Scopus). Our initial search yielded 91 results, which we assessed for quality and revised, giving a total of 35 relevant studies. We then performed a snowballing activity, but did not find any additional relevant studies. Finally, we extracted relevant data from the obtained articles and catalogued our observations.

The systematic mapping study revealed two clear gaps in existing ECS research:

- **RG1.**: The primary studies predominantly emphasised the benefits of ECS over the drawbacks, suggesting a positive bias in the field. This opens an opportunity for the research community to discover and establish more drawbacks of the ECS pattern.
- **RG2.**: The limited discussion of ECS backend design also suggests an opportunity for further empirical research examining how implementation choices influence performance, memory behaviour, and scalability in real-time systems.

Next, we conducted an empirical case study evaluating six ECS-based collision-handling implementations using the Flecs framework. These implementations included relationship-based approaches, non-fragmenting relationships, collision entities, record-list structures, and spatial hashing variants. To structure our empirical case study and answer the second gap **RG2** revealed in the mapping study, we asked the following research questions:

- **RQ2.1:** How do different ECS collision-handling methods exhibit measurable performance variation?
- **RQ2.2:** How do the different integrations of these methods introduce code complexity that can be quantified using cognitive and cyclomatic metrics?

The results showed that ECS design choices have a significant impact on both performance and code complexity. In particular, relationship-based approaches exhibited lower performance due to

archetype fragmentation, while spatial hashing approaches significantly improved performance at the cost of increased implementation complexity. Intermediate approaches, such as collision entities and non-fragmenting relationships, offered a balance between these trade-offs.

In addition to performance evaluation, this thesis analysed code complexity using cognitive and cyclomatic metrics, and applied the Analytic Hierarchy Process (AHP) to support decision-making based on performance and complexity criteria. This provides practical guidance for developers when selecting an appropriate ECS design strategy depending on their priorities.

Overall, this thesis contributes to the understanding of ECS by bridging the gap between high-level architectural discussions and low-level implementation behaviour, specifically for archetype-based ECS. By combining a systematic mapping study, a detailed technical analysis, and an empirical evaluation, it provides a more comprehensive view of how ECS design decisions influence both runtime performance and development complexity.

6.2 Future Work

6.2.1 Short Term

We performed the systematic mapping study strictly on academic papers, intentionally avoiding grey literature since it can be opinionated and biased. However, given that the ECS pattern originated from the game industry, as future work, the mapping study could be extended to include grey literature and provide a greater overview of the ECS pattern picture.

We limited our empirical case study evaluation to same-size circle collisions, which simplified the implementation and parameter selection for the spatial hashing grid. As future work, the study could be extended to include diverse collision shapes and varying entity sizes to better reflect realistic gameplay scenarios.

We also performed our experiment strictly on the Flecs ECS framework, which we selected based on its consistently superior performance in existing benchmarks. As future work, the study could be replicated across other archetype-based and sparse-set-based ECS frameworks to determine how much of the observed behaviour is framework-specific.

We did not consider multithreading in this study, as our focus was on understanding single-threaded implementation trade-offs in archetype-based ECS. As future work, the impact of parallelisation on archetype fragmentation and spatial partitioning strategies could be investigated.

We also compared only ECS-based collision implementations against one another, rather than against traditional object-oriented or component-based approaches. As future work, a similar study could compare these implementations against a standard Unity component-based counterpart to provide a broader understanding of how data-oriented designs differ from object-oriented ones in practice.

Finally, the crossover behaviour observed between the per-cell and per-entity spatial hashing variants suggests that an adaptive strategy that dynamically switches between the two based on entity count could combine the strengths of both approaches, and warrants further empirical investigation.

6.2.2 Long Term

As discussed in Chapter 5, we identified possible research directions for the Entity-Component-System pattern based on the thesis results and our research observations during the thesis process.

One important direction concerns the development of ECS-specific tooling. Current static analysis tools are poorly adapted to composition-based architectures. Future research could explore (1) automated migration tools that transform existing object-oriented codebases into ECS designs, building on techniques such as Formal Concept Analysis [23], and (2) tools to address the *accidental system matching problem*. Such tools could statically analyse and visualise how component additions or removals affect entity system matching across the archetype lattice, thereby improving predictability, debuggability, and overall architectural understanding in complex ECS applications.

Another significant avenue lies in the exploration of unified CPU-GPU execution models for archetype-based ECS. While existing GPU-centric frameworks [24] demonstrate strong potential for massively parallel simulations, they often sacrifice the runtime dynamism essential for real-time applications. Future work should investigate hybrid frameworks that maintain both high-performance

GPU execution and flexible structural changes during runtime. This will require advances in heterogeneous scheduling, automatic code generation for GPU-compatible systems, and efficient synchronisation mechanisms. The theoretical foundations of ECS concurrency proposed by Redmond et al. [2] offer a strong basis for such developments.

Appendix A

Mapping Study Appendix

Table A.1: Primary studies and ECS-related contributions.

Field	Description
Title	A cloud based simulation service for 3D crowd simulations
Reference	[M18]
Venue Type	Conference
On Games	No
ECS Contribution	Introduces a cloud-based simulation framework based on the Entity Component System (ECS) pattern.
Title	A Concurrent Entity Component System for Geographical Wildlife Epidemiological Modeling
Reference	[M15]
Venue Type	Journal
On Games?	No
ECS Contribution	Employs a parallel ECS for epidemiological modelling, highlighting Rust's mutability/immutability features and the use of SPECS library for parallelism and performance.

Field	Description
Title	A Generalizable Entity-Component-System Architecture for Underwater ROV Control
Reference	[M23]
Venue Type	Conference
On Games?	No
ECS Contribution	Presents a novel software architecture for underwater ROVs based on ECS, enabling modular cross-platform state synchronisation and adaptable thruster control.
Title	A Mapping Study of the Entity Component System Pattern
Reference	[M31]
Venue Type	Workshop
On Games?	Yes
ECS Contribution	Conducts a systematic mapping study to identify ECS publication trends, research focus, and the implications (benefits/drawbacks) of developing with ECS.
Title	A Model-driven Middleware Integration Approach for Performance-Sensitive Distributed Simulations
Reference	[M12]
Venue Type	Conference
On Games?	No
ECS Contribution	Discusses ECS for code organisation, modularity, and composability in military simulators.
Title	Accessibility and Serious Games: What About Entity Component System Software Architecture?
Reference	[M30]
Venue Type	Journal
On Games?	Yes

Field	Description
ECS Contribution	Examines using ECS for accessibility in the serious game E-LearningScape, highlighting its modularity and scalability for adding unanticipated functionalities.
Title	An Argument for the Practicality of Entity Component Systems as the Primary Data Structure for an Interpreter or Compiler
Reference	[M11]
Venue Type	Conference
On Games?	No
ECS Contribution	Examines how ECS can benefit compiler and interpreter design, specifically simplifying optimisation passes and improving serialisability.
Title	An auxiliary development framework for lightweight RPG games based on Unity3D
Reference	[M3]
Venue Type	Journal
On Games?	Yes
ECS Contribution	Develops a framework for RPG games using ECS, highlighting its modularity and flexibility for managing game entities and their behaviours.
Title	An Educational Experience to Raise Awareness About Space Debris
Reference	[M34]
Venue Type	Journal
On Games?	No
ECS Contribution	Focuses on VR experience development using Unity ECS to run a simulation with thousands of objects.
Title	An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation
Reference	[M19]
Venue Type	Conference

Field	Description
On Games?	No
ECS Contribution	Presents Madrona, an architecture for high-performance simulation using ECS, showcasing its use in a hide-and-seek environment.
Title	Benchmarking Performance of Unity's Data Oriented Technology Stack
Reference	[M25]
Venue Type	Conference
On Games?	No
ECS Contribution	Compares Unity's DOTS (ECS) to traditional object-oriented paradigms in flight simulations, finding ECS superior in CPU performance, FPS, and memory usage.
Title	Decoupling the Entity Component System pattern using semantic traits for reusable realtime interactive systems
Reference	[M22]
Venue Type	Workshop
On Games?	No
ECS Contribution	Proposes decoupling ECS using semantic traits to enhance reusability in interactive systems. Mentions semantic grounding and queries for inter-system communication.
Title	Developing games with data-oriented design
Reference	[M28]
Venue Type	Workshop
On Games?	Yes
ECS Contribution	Uses ECS as a practical design pattern to leverage data-oriented design principles for efficient game development.
Title	ECS-Grid: Data-Oriented Real-Time Simulation Platform for Cyber-Physical Power Systems
Reference	[M13]

Field	Description
Venue Type	Conference
On Games?	No
ECS Contribution	Presents ECS-Grid for power systems simulation, explains core ECS concepts and benefits, and cites game industry examples (Minecraft, Unity's DOTS, Call of Duty: Vanguard)
Title	Entity Component System Architecture for Scalable, Modular, and Power-Efficient IoT-Brokers
Reference	[M2]
Venue Type	Conference
On Games?	No
ECS Contribution	Proposes using ECS architecture for IoT brokers to enhance scalability, modularity, and power efficiency. Explains core ECS benefits compared to OOP.
Title	Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern
Reference	[M1]
Venue Type	Journal
On Games?	No
ECS Contribution	Designs a formal model ("Core ECS") to reveal the essence of the pattern and identifies a class of ECS programs that enable deterministic concurrency.
Title	How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library
Reference	[M5]
Venue Type	Conference
On Games?	Yes

Field	Description
ECS Contribution	Presents ECS as an alternative to traditional object-oriented design for electronic design automation tools, specifically for its performance and software engineering benefits.
Title	Leveraging the Learning Curve: Reusing Existing Architectural Patterns to Design and Implement MAS
Reference	[M10]
Venue Type	Journal
On Games?	No
ECS Contribution	Proposes introducing a minimal set of agent concepts into the Distributed Systems domain to improve Multi-Agent System (MAS) engineering using the ECS pattern.
Title	Machine-Learning-Reinforced Massively Parallel Transient Simulation for Large-Scale Renewable-Energy-Integrated Power Systems
Reference	[M9]
Venue Type	Journal
On Games?	No
ECS Contribution	Demonstrates the applicability of ECS beyond game development, showing its effectiveness in integrating machine learning models and facilitating parallel computation for complex simulations.
Title	Modulith: A Game Engine Made for Modding
Reference	[M32]
Venue Type	Conference
On Games?	Yes
ECS Contribution	Presents Modulith, a game engine designed for modding using ECS for data organisation and communication.
Title	Open-Source Game Engine & Framework for 2D Game Development
Reference	[M4]

Field	Description
Venue Type	Conference
On Games?	Yes
ECS Contribution	Focuses on game engine development using ECS and performs a user study on ECS development.
Title	Optimizing Bevy-ECS Using Predictive JSSP Approach
Reference	[M33]
Venue Type	Conference
On Games?	Yes
ECS Contribution	Models the Bevy ECS integrated scheduler as a Job Shop Scheduling Problem (JSSP) and proposes a predictive multi-threaded scheduler to improve CPU utilisation.
Title	OSS Scripting System for Game Development in Rust
Reference	[M7]
Venue Type	Journal
On Games?	Yes
ECS Contribution	Implements a scripting system for Amethyst, a Rust game engine, using the Legion ECS for managing and organising in-game objects
Title	Parallel Computing Technologies and Rendering Optimization in the Problem of Fluid Simulation by the Example of the Incompressible Schrödinger Flow Method
Reference	[M21]
Venue Type	Journal
On Games?	No
ECS Contribution	Utilises ECS, C# job systems, and Burst compilation in Unity for fluid simulation rendering optimisation, leveraging CPU capabilities
Title	Polyphony: Programming Interfaces and Interactions with the Entity Component System Model

Field	Description
Reference	[M6]
Venue Type	Journal
On Games?	No
ECS Contribution	Introduces Polyphony, a GUI toolkit built on ECS, analysing ECS implementations across different frameworks. Discusses adapting ECS for HCI, focusing on event handling and interactions.
Title	Proposed Application for an Entity Component System in an Energy Services Interface
Reference	[M8]
Venue Type	Conference
On Games?	No
ECS Contribution	Proposes applying ECS in Energy Service Interfaces for managing dynamic entity characteristics. Compares ECS libraries for performance and suggests advantages over traditional databases.
Title	pyGANDALF - An open-source Geometric, Animation, Directed, Algorithmic, Learning Framework for Computer Graphics
Reference	[M26]
Venue Type	Conference
On Games?	No
ECS Contribution	Introduces a Python-based computer graphics framework built on ECS and WebGPU, evaluating its effectiveness in educational settings.
Title	PySeidon - A Data-Driven Maritime Port Simulation Framework
Reference	[M20]
Venue Type	Conference
On Games?	No
ECS Contribution	Introduces PySeidon, a port simulation framework, using ECS as the foundation for modelling entities like ships and port infrastructure.

Field	Description
Title	Real-Time Cyber-Physical Digital Twin for Low Earth Orbit Satellite Constellation Network Enhanced Wide-Area Power Grid
Reference	[M14]
Venue Type	Journal
On Games?	No
ECS Contribution	Introduces RustSat, a satellite digital twin, built using the Bevy ECS game engine. Explains the basic elements of ECS and its data-driven nature.
Title	Semantic Entity-Component State Management Techniques to Enhance Software Quality for Multimodal VR-Systems
Reference	[M35]
Venue Type	Journal
On Games?	No
ECS Contribution	Proposes extending ECS with semantic techniques for multimodal VR systems to enhance software quality attributes, providing implementation insights.
Title	Software Patterns for Creating Computer Simulations
Reference	[M27]
Venue Type	Conference
On Games?	No
ECS Contribution	Compares OOP, ECS, and Functional Reactive Programming patterns for implementing real-time simulations, noting ECS's efficiency in parallelisability.
Title	Understanding Unity's ECS Architecture
Reference	[M24]
Venue Type	Conference
On Games?	No

Field	Description
ECS Contribution	Focuses on implementing a flight simulator using Unity's DOTS package to demonstrate multi-threading and the separation of data from logic.
Title	Unlimited Rulebook: a Reference Architecture for Economy Mechanics in Digital Games
Reference	[M29]
Venue Type	Conference
On Games?	Yes
ECS Contribution	Proposes a reference architecture for game economies, referencing game engine architecture and data-driven design related to ECS.
Title	Vico: An Entity Component System based co-simulation framework
Reference	[M16]
Venue Type	Journal
On Games?	No
ECS Contribution	Introduces Vico, an ECS-based co-simulation framework. Explains core ECS concepts, including families for efficient entity selection.
Title	Wait-free hash maps in the Entity Component System pattern for realtime interactive systems
Reference	[M17]
Venue Type	Workshop
On Games?	No
ECS Contribution	Applies wait-free hash maps in ECS for real-time systems, showcasing its use in spacecraft simulation with entities representing components and the environment.

Bibliography

[1] Newzoo, "Free report: Newzoo's global games market report 2025 — free version," Sep 2025, [Online]. Accessed: Jun. 3, 2026. [Online]. Available: https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2025

[2] P. Redmond, J. Castello, J. M. Calderón Trilla, and L. Kuper, "Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025, [Online]. Accessed: Jun. 3, 2026. [Online]. Available: https://doi.org/10.1145/3763050

[3] L. Voisard, H. De Freitas Serra, C. Politowski, F. Petrillo, and Y.-G. Géhéneuc, "A mapping study of the entity component system pattern," in *2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS)*, 2025, pp. 33–40.

[4] J. D. Bayliss, "Developing games with data-oriented design," in *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, ser. GAS '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 30–36.

[5] R. Fabian, "Data-oriented design," *framework*, vol. 21, pp. 88–96, 2018.

[6] T. Cheng, T. Duan, and V. Dinavahi, "Real-Time Cyber-Physical Digital Twin for Low Earth Orbit Satellite Constellation Network Enhanced Wide-Area Power Grid," *IEEE Open Journal of the Industrial Electronics Society*, pp. 1–14, 2024.

[7] J. Dahl and F. C. Harris, "An Argument for the Practicality of Entity Component Systems as the Primary Data Structure for an Interpreter or Compiler," in *Proceedings of the 2025*

ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ser. Onward! '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 85–98, [Online]. Accessed: Jun. 3, 2026. [Online]. Available: https://doi.org/10.1145/3759429.3762622

[8] Skypjack, "Entity component system," [Online]. Accessed Jun. 3, 2026. [Online]. Available: https://github.com/skypjack/entt/wiki/Entity-Component-System

[9] S. Mertens, "Flecs documentation," 2025, [Online]. Accessed: Jun. 3, 2026. [Online]. Available: https://www.flecs.dev/flecs/md_docs_2Docs.html

[10] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[11] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, ver. 2.3 ebse technical report. ebse, Tech. Rep., 2007.

[12] L. Lazaridis, M. Papatsimouli, K.-F. Kollias, P. Sarigiannidis, and G. F. Fragulis, "Hitboxes: A survey about collision detection in video games," in *HCI in Games: Experience Design and Game Mechanics*, X. Fang, Ed. Cham: Springer International Publishing, 2021, pp. 314–326.

[13] A. Beimler, "Benchmarks of common ecs (entity-component-system)-frameworks in c++ (or c)," 2023, [Online]. Accessed: Jun. 3, 2026. [Online]. Available: https://github.com/abeimler/ecs_benchmark

[14] C. Ericson, *Real-Time Collision Detection*. USA: CRC Press, Inc., 2004.

[15] J. Gregory, *Game Engine Architecture, Second Edition*, 2nd ed. USA: A. K. Peters, Ltd., 2014.

[16] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[17] R. W. Saaty, "The analytic hierarchy process—what it is and how it is used," *Mathematical modelling*, vol. 9, no. 3-5, pp. 161–176, 1987.

[18] T. L. Saaty, “Analytic hierarchy process,” in *Encyclopedia of operations research and management science*. Springer, 2013, pp. 52–64.

[19] T. Raffaillac and S. Huot, “Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model,” *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. EICS, pp. 8:1–8:22, Jun. 2019.

[20] T. Fontana, R. Netto, V. Livramento, C. Guth, S. Almeida, L. Pilla, and J. L. Güntzel, “How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library,” in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ser. ISPD ’17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 25–31.

[21] F. Pouhela, D. Krummacker, and H. D. Schotten, “Entity Component System Architecture for Scalable, Modular, and Power-Efficient IoT-Brokers,” in *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, Jul. 2023, pp. 1–6, iSSN: 2378-363X.

[22] B. Ganter, R. Wille, and R. Wille, *Formal concept analysis*. Springer, 1999, vol. 150.

[23] D. Llansó, M. Gómez-Martín, P. Gómez-Martín, and P. González-Calero, “Knowledge guided development of videogames,” in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 7, no. 3, 2011, pp. 8–13.

[24] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian, “An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation,” *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–13, Aug. 2023.

Primary Studies

[M1] P. Redmond, J. Castello, J. M. Calderón Trilla, and L. Kuper, "Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3763050

[M2] F. Pouhela, D. Krummacker, and H. D. Schotten, "Entity Component System Architecture for Scalable, Modular, and Power-Efficient IoT-Brokers," in *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, Jul. 2023, pp. 1–6, iSSN: 2378-363X.

[M3] B. Zhang, H. Shi, and X. Wang, "An auxiliary development framework for lightweight RPG games based on Unity3D," *Computer Animation and Virtual Worlds*, vol. 35, no. 1, 2024.

[M4] S. A. E. Campos, B. A. M. Morales, and A. A. V. Núñez, "Open-Source Game Engine & Framework for 2D Game Development," in *2022 IEEE Engineering International Research Conference (EIRCON)*, Oct. 2022, pp. 1–4.

[M5] T. Fontana, R. Netto, V. Livramento, C. Guth, S. Almeida, L. Pilla, and J. L. Güntzel, "How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library," in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ser. ISPD '17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 25–31.

[M6] T. Raffaillac and S. Huot, "Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. EICS, pp. 8:1–8:22, Jun. 2019.

[M7] P. da Silva, R. Campos, and C. Rocha, “OSS Scripting System for Game Development in Rust,” *IFIP Advances in Information and Communication Technology*, vol. 624, pp. 51–58, 2021.

[M8] T. Slay, G. B. Spitzer, and R. B. Bass, “Proposed Application for an Entity Component System in an Energy Services Interface,” in *2022 IEEE Conference on Technologies for Sustainability (SusTech)*, Apr. 2022, pp. 177–180.

[M9] T. Cheng, R. Chen, N. Lin, T. Liang, and V. Dinavahi, “Machine-Learning-Reinforced Massively Parallel Transient Simulation for Large-Scale Renewable-Energy-Integrated Power Systems,” *IEEE Transactions on Power Systems*, pp. 1–12, 2024, conference Name: IEEE Transactions on Power Systems.

[M10] A. Casals and A. A. F. Brandão, “Leveraging the Learning Curve: Reusing Existing Architectural Patterns to Design and Implement MAS,” *IEEE Access*, vol. 13, pp. 45 809–45 825, 2025.

[M11] J. Dahl and F. C. Harris, “An Argument for the Practicality of Entity Component Systems as the Primary Data Structure for an Interpreter or Compiler,” in *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! ’25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 85–98. [Online]. Available: https://doi.org/10.1145/3759429.3762622

[M12] T. Brummett, K. An, A. Gokhale, and S. Mertens, “A Model-driven Middleware Integration Approach for Performance-Sensitive Distributed Simulations,” in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, May 2020, pp. 65–73, iSSN: 2375-5261.

[M13] T. Cheng, T. Duan, and V. Dinavahi, “ECS-Grid: Data-Oriented Real-Time Simulation Platform for Cyber-Physical Power Systems,” *IEEE Transactions on Industrial Informatics*, vol. 19, no. 11, pp. 11 128–11 138, Nov. 2023, conference Name: IEEE Transactions on Industrial Informatics.

[M14] ——, "Real-Time Cyber-Physical Digital Twin for Low Earth Orbit Satellite Constellation Network Enhanced Wide-Area Power Grid," *IEEE Open Journal of the Industrial Electronics Society*, pp. 1–14, 2024, conference Name: IEEE Open Journal of the Industrial Electronics Society.

[M15] A. V. Davis and S. Wang, "A Concurrent Entity Component System for Geographical Wildlife Epidemiological Modeling," *Geographical Analysis*, vol. 53, no. 4, pp. 836–868, 2021, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/gean.12258.

[M16] L. Hatledal, Y. Chu, A. Styve, and H. Zhang, "Vico: An entity-component-system based co-simulation framework," *Simulation Modelling Practice and Theory*, vol. 108, 2021.

[M17] P. Lange, R. Weller, and G. Zachmann, "Wait-free hash maps in the entity-component-system pattern for realtime interactive systems," in *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, Mar. 2016, pp. 1–8, iSSN: 2328-7829.

[M18] R. Pax, J. J. Gomez-Sanz, I. S. Olivenza, and M. C. Bonett, "A cloud based simulation service for 3D crowd simulations," in *Proceedings of the 22nd International Symposium on Distributed Simulation and Real Time Applications*, ser. DS-RT '18. Madrid, Spain: IEEE Press, Oct. 2018, pp. 112–119.

[M19] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian, "An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation," *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–13, Aug. 2023.

[M20] P. Skaisgiris, W. Simoncini, F. Barbero, A. Ahangi, and R. Mockel, "PySeidon - A Data-Driven Maritime Port Simulation Framework," in *Proceedings of the 13th International Conference on Computer Modeling and Simulation*, ser. ICCMS '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 164–171.

[M21] B. Tiulkin, "Parallel Computing Technologies and Rendering Optimization in the Problem of Fluid Simulation by the Example of the Incompressible Schrödinger Flow Method," *Physics of Particles and Nuclei*, vol. 55, no. 3, pp. 519–521, 2024.

[M22] D. Wiebusch and M. E. Latoschik, "Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems," in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, Mar. 2015, pp. 25–32, iSSN: 2328-7829.

[M23] E. McIvor, G. Sklivanitis, and D. A. Pados, "A Generalizable Entity-Component-System Architecture for Underwater ROV Control," in *2025 Symposium on Maritime Informatics and Robotics (MARIS)*, Jun. 2025, pp. 1–4.

[M24] B. Martin, D. Hodson, and L. Merkle, "Understanding Unity's ECS Architecture," in *AI Revolution: Research, Ethics and Society*, H. R. Arabnia, L. Deligiannidis, S. Amirian, F. Ghareh Mohammadi, and F. Shenavarmasouleh, Eds. Cham: Springer Nature Switzerland, 2026, pp. 504–515.

[M25] B. Martin, J. Visci, M. Visci, J. Thompson, and D. Hodson, "Benchmarking Performance of Unity's Data Oriented Technology Stack," in *AI Revolution: Research, Ethics and Society*, H. R. Arabnia, L. Deligiannidis, S. Amirian, F. Ghareh Mohammadi, and F. Shenavarmasouleh, Eds. Cham: Springer Nature Switzerland, 2026, pp. 516–526.

[M26] J. Petropoulos, M. Kamarianakis, A. Protopsaltis, and G. Papagiannakis, "pyGANDALF - An open-source Geometric, ANimation, Directed, Algorithmic, Learning Framework for Computer Graphics," in *SIGGRAPH Asia 2024 Educator's Forum*, ser. SA '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3680533.3697057

[M27] Y. Sobolev, R. Kreinin, T. Kudinova, and F. Nanay, "Software Patterns for Creating Computer Simulations," in *Artificial Intelligence Algorithm Design for Systems*, R. Silhavy and P. Silhavy, Eds. Cham: Springer Nature Switzerland, 2024, pp. 563–573.

[M28] J. D. Bayliss, "Developing games with data-oriented design," in *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, ser. GAS '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 30–36.

[M29] W. K. Mizutani and F. Kon, "Unlimited Rulebook: a Reference Architecture for Economy Mechanics in Digital Games," in *2020 IEEE International Conference on Software Architecture (ICSA)*, Mar. 2020, pp. 58–68.

[M30] M. Muratet and D. Garbarini, "Accessibility and Serious Games: What About Entity-Component-System Software Architecture?" *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12517 LNCS, pp. 3–12, 2020.

[M31] L. Voisard, H. De Freitas Serra, C. Politowski, F. Petrillo, and Y.-G. Géhéneuc, "A mapping study of the entity component system pattern," in *2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS)*, 2025, pp. 33–40.

[M32] D. Götz and S. von Mammen, "Modulith: A Game Engine Made for Modding," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, ser. FDG '23. New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 1–8.

[M33] B. Ma and Y. Du, "Optimizing Bevy-ECS Using Predictive JSSP Approach," in *2025 IEEE 5th International Conference on Power, Electronics and Computer Applications (ICPECA)*, Jan. 2025, pp. 754–759.

[M34] C. Colombo, N. D. Blas, I. Gkolias, P. L. Lanzi, D. Loiacono, and E. Stella, "An Educational Experience to Raise Awareness About Space Debris," *IEEE Access*, vol. 8, pp. 85 162–85 178, 2020, conference Name: IEEE Access.

[M35] M. Fischbach, D. Wiebusch, and M. E. Latoschik, "Semantic Entity-Component State Management Techniques to Enhance Software Quality for Multimodal VR-Systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 4, pp. 1342–1351, Apr. 2017, conference Name: IEEE Transactions on Visualization and Computer Graphics.

Adjacent Bibliography

[A1] L. Voisard, C. Politowski, F. Petrillo, and Y.-G. Géhéneuc, "Cg-wfc: A hybrid cyclic-graph & wfc method for designer-guided and replayable procedural content generation," in *Proceedings of the 10th International Workshop on Games and Software Engineering*, ser. GAS '26. New York, NY, USA: ACM, April 2026, p. 5. [Online]. Available: https://doi.org/10.1145/3786171.3788381

[A2] M. Gumin, "Wave function collapse algorithm," 2016, software version 1.0. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse

[A3] J. Dormans and S. Bakkes, "Generating missions and spaces for adaptable play experiences," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 216–228, 2011.

[A4] J. Dormans, "Cyclic generation," in *Procedural Generation in Game Design*. AK Peters/CRC Press, 2017, pp. 83–96.