Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Game Design and Development

Capstone Final Design & Development Approval Form

May 07, 2025

Student Name: **Jared Goronkin**

Research Title: **The Utility of a Modular Framework: Evaluating Mosaic for Game Development**

Keywords: **Mosaic, Modular, Game Object Model, Ability System, Character Controller**

Presentation: **https://youtu.be/ciGfz5IPof0**

Sten McKinzie
Lead Capstone Advisor

Christopher Egert
Research Advisor

David Schwartz, Ph.D.
Director, School of Interactive Games and Media

The Utility of a Modular Framework: Evaluating Mosaic for Game Development

Jared Goronkin
Rochester Institute of Technology
School of Interactive Games & Media
B. Thomas Golisano College of Computing and Information Sciences
jgoronkin@gmail.com

Paper submitted in partial fulfillment of the requirements for the degree of Master of Science in Game Design and Development
May 07, 2025

Abstract: This paper presents a taxonomy for *Mosaic*, along with select Character Controllers and Ability Systems. *Mosaic* is a lightweight package for Unity that enables developers to create gameplay features that are modular, allowing for scalable development and the preservation of features across projects. For the taxonomy ten architectural and feature categories were chosen through an analysis of the systems. This study evaluates *Mosaic's* position within the landscape of these various products, finding that *Mosaic* prioritizes applicability, reusability, and architectural clarity while forgoing out of the box features. These tradeoffs make *Mosaic* particularly suited to teams prioritizing custom behaviors, long term maintainability, and the ability to create an ever expanding cross compatible library of gameplay features. This analysis contributes to an understanding of how prioritizing modular design can impact production, and outlines opportunities to extend *Mosaics* capabilities.

1. Problem Statement

Mosaic is a modular game development framework which I've developed to address inefficiencies in creating, integrating, and managing gameplay features. While the system has provided flexibility, productivity gains, and collaborative benefits, to smaller projects, testing is required to see if such results scale to larger productions.

This paper evaluates *Mosaic's* real-world utility within the context of the development of Echoes In The Mists by Petrichor Studios. The goal is to determine whether *Mosaic* effectively reduces development complexity, enhances productivity, and supports collaboration as claimed, and to identify any limitations or challenges in its application.

2. Significance

The flexibility of a chosen game object model is highly valued by developers. These early structural decisions come with tradeoffs and can have far reaching consequences throughout development impacting everyone on the development team. Utilizing an object oriented approach may make things easier to get started, but can scale poorly when scoping up or pivoting the project, sometimes requiring major redesigns. Utilizing an Entity Component System (ECS) improves modularity, but can increase the challenge of scripting minor features for less experienced programmers and designers. (Skypjack, n.d.)

These models are fundamental, rather than introducing a new one, *Mosaic* offers a framework that can work alongside these models that is targeted at the game objects that are core to most games, that of the actor. *Mosaic* breaks down the actor into truly modular components both across actors, and across any games that utilize *mosaic*, without requiring extra code to wire the various components together.

By exploring *mosaics* strengths, weaknesses, and solutions to challenging problems we can help further guide its development, determine its real world value to those in the industry, as well as help developers streamline their development processes.

3. Background

Mosaic is a framework that sits just above the game object model, and just below the character controller. By extending *Mosaic*, developers can create fully modular runtime features that are cross compatible across completely different actors and even projects. These features can include anything from movement, to attacks, to skill trees, and scripted sequences.

When the development of *Mosaic* began, the validity of the underlying concept would often be called into question by those in the industry that I shared it with. The sentiment has generally been something along the lines of the sentiment expressed in a Stack Exchange thread from 2023 (Kevin). Often the responses were something along the lines of "Well that would be amazing if it was possible, but there must be a reason no one has done this before." *Mosaic* isn't the first project to try to tackle concepts like this, but it does approach it from a unique angle, which I attribute to it finding success where others haven't as of yet. *Mosaic* wasn't designed to provide solutions for gameplay features, so there isn't the sort of fluff that similar systems employ. It was designed to tackle the abstract problem of truly modular actors with full cross compatibility while imposing no limitations on their capabilities.

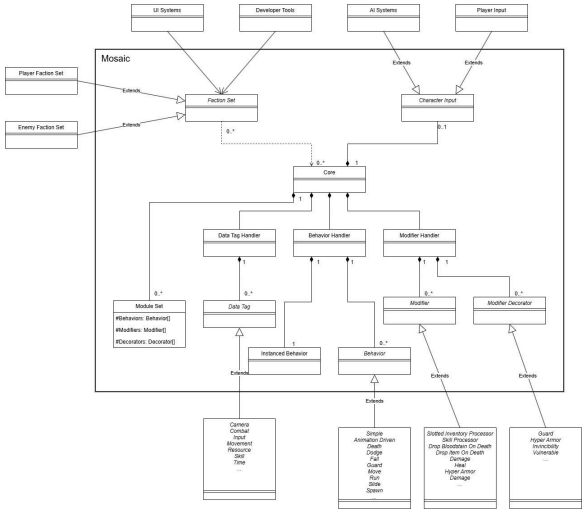

Figure 1: *Mosaic* Architectural Diagram

Systems that work with *Mosaic* can be broken down into two categories. External systems reference and interact with *Mosaic* through a single unified interface, allowing them to interact with actors built with *Mosaic* in an abstracted manner. Internal systems extend *Mosaics* Modifiers, Modifier Decorators, and DataTags, affording them all of the benefits of *Mosaic*.

Mosaic shares many similarities with the component architecture and builds on the concepts to guarantee interoperability across actors (Gregory, 2019).The Core can be thought of as the container object, DataTags can be thought of the container objects state, and Behaviors, Modifiers, and Modifier Decorators, can all be thought of as the components. In Game Programming Patterns, Nystrom (n.d.) outlines some of the major challenges with the component architecture, such as an inherent lack of encapsulation of data which can cause issues with code clarity and unnecessary memory usage. The benefit of *Mosaic* over the standard component architecture is that *Mosaic* solves all of these issues.

DataTags are essentially a dynamic type safe blackboard. This allows components to share data, without the risk of introducing bugs due to spelling errors, while still being able to add new data types as needed.

One of the fundamental components of *Mosaic* are the behaviors. Each behavior must be fully modular, which requires a modular behavior selection algorithm. A

utility system was chosen due to both its simplicity, as well as its ability to simulate any other behavior selection algorithm with relative ease. (Daw et al., 2019)

Not all actor behaviors are stateful, some are instantaneous, and some can persist for undetermined amounts of time while overlapping other stateful and non-stateful behaviors. *Mosaic's* solution to supporting this type of behavior was heavily inspired by For Honors modifiers. In For Honor Modifiers are used for everything from adding visual flair to a character, to applying status effects over time. (GDC 2025, 2019) *Mosaic* simplifies this solution down to its fundamentals, improving flexibility. *Mosaic* also includes a structure that allows for the dynamic decoration of any modifier. This enables a modular approach to reacting to and extending the modifiers functionality, and is an essential part of achieving full modularity and cross-compatibility. (Gamma et al., 2016)

There are two main types of systems that could be categorized as similar to *Mosaic*. That of ability systems and character controllers. We will be taking a look at three, the *Gameplay Ability System* (*GAS*) for Unreal, as well as the *Opsive* and *Invector* character controllers. *GAS* falls squarely into the category of a standalone ability system, designed to be used alongside a character controller or other system. *Opsive* is a character controller with a built in ability system. *Invector* is a stand alone character controller.

The *Gameplay Ability System* (*GAS*) was originally developed for Unreal. It's used in many games including Fortnight, and is a free package. The main purpose of *GAS* is to create abilities for games in a unified manner with built in networking capabilities. This system does a good job at modularizing features, although it does not achieve full modularity. There is also a lot of prebuilt functionality targeting conventional character driven games. (Gameplay Ability System for Unreal Engine, n.d.)The gameplay ability system also has a steep learning curve. *Mosaic* is a much more lightweight framework, allowing it to fit into more contexts, and also achieves full modularity and cross compatibility.

There are various character controllers available as packages on the Unity asset store such as *Invector* (Invector, n.d.) and *Opsive*(Character Solution, n.d.). These are hard coded solutions and broken up into pieces to be sold. Unlike *GAS*, these solutions offer prebuilt functionality and are very quick to set up. They are however difficult to adjust and extend, relying on utilizing and remixing existing features to achieve new functionality. These add ons are not modular and often require unique setup. *Mosaic* is much faster in terms of new feature creation, with its modular data driven design.

4. Methodology

There isn't anything out there that does exactly what *Mosaic* does, but there are many products out there that can help achieve similar things from the perspective of various audiences. Identifying where this overlap occurs will allow us to analyze the relevant features and identify the strengths and weaknesses in regards to the various use cases.

Adding anything to a project comes with a cost, so it is essential that the benefits are tangible. Frameworks like *Mosaic* are intended to improve development efficiency. This research aims to outline the framework's strengths, weaknesses, and potential applications through its categorization.

This research focuses on the comparison of various systems that fall into categories similar to gameplay ability systems and character controllers (Ultimate Character Controller n.d., Gameplay Ability System for Unreal Engine, n.d.) The goal is to evaluate each system against a set of traits relating to their usability and architecture to better understand their utility in real world production environments.

As of now, what *Mosaic* is, is not well defined. Three systems were selected to be compared alongside *Mosaic* based on their relevance, popularity, and architectural variance.

- *Mosaic*
- *Gameplay Ability System* (*GAS*)
- *Opsive* Character Controller

- *Invector* Character Controller

A taxonomy of ten categories was developed to assess the architectural features and utility of each system.

4.1 Architectural Criteria

Reusability

- High: Features require no engineering to be reusable
- Medium: Features require significant modification & engineering to be reusable
- Low: Not supported

Modularity

How granularly the system is broken down into parts, and how effectively those parts have been decoupled from each other.

- Full: components are fully decoupled from each other and external systems.
- Partial: the architecture encourages modular coding practices, but relies on direct connections to external systems.
- None: Systems are tightly coupled.

Prebuilt Feature Integration

- Seamless: External feature integration requires no customization of assets
- Asset Coupled: External feature integration requires customization of assets
- Code Coupled: External feature integration requires modification of code

Custom Feature Development

- Easy: Custom features require minimal integration;
- Challenging: Custom features require creative problem solving
- Unsupported: Custom features are not supported

Codebase Scalability

- High: Encourages parallel development, minimizes merge conflicts, separates responsibilities.
- Medium: Works for small teams, shared resources require careful planning to avoid conflicts.
- Low: High coupling and monolithic structures make feature development and collaboration difficult

Feature Criteria

Target User:

- Developers: Engineers directly implementing gameplay subsystems.
- Designers: Team members who utilize the developed systems to create experiences for the player.

Networking Support

- Yes
- No

Character Controller

- Yes
- No

Ability System

- Yes
- No

Model Swapping

- Yes
- No

Each category's ratings were based on traits observable through documentation and user experience.

The systems were analyzed through various means. A thorough analysis of all systems documentation was done in order to develop an understanding of their architecture, use cases, and developer intent. Hands-on testing with the software was done for *Mosaic* and *GAS*. The same was not done for *Opsive* and *Invector* as they are both expensive systems upfront, with many features locked behind further paywalls.

Through the classification we can make informed judgments to determine the practical uses of the various systems.

5. Results

System	Reusability	Modularity	Prebuilt Feature Integration	Custom Feature Development	Codebase Scalability
Mosaic	High	High	Effortless	Easy	High
GAS	Low	High	None	Easy	High
Opsive	Low	Low	Low	Limited	Low
Invector	Low	Low	Low	Limited	Low

Figure 2: Architecture Characteristics

System	Target User	Networking Support	Character Controller	Ability System	Model Swapping
Mosaic	Developer	No	No	No	No
GAS	Developer	Yes	No	Yes	No
Opsive	Designer	Partial (Add-on)	Yes	Yes	Yes (Humanoid)
Invector	Designer	Partial (Add-on)	Yes	No	Yes (Humanoid)

Figure 3: Feature Coverage

5.1 Reusability

Mosaic was the best fit in regards to reusability due to its ability to encapsulate gameplay logic, along with its various features that allow systems extended from *Mosaic* to be compatible across projects. Other systems scored low, as this functionality would need to be custom built by the developer when using these systems.

5.2 Modularity

Mosaic exhibited high modularity due its low coupling between components thanks to both DataTags and its modularized behavior selection algorithm. *GAS* utilizes a similar layout, although is slightly less modular in terms of the selection of its Gameplay Abilities. This can lead to situations where the user compromises the modularity of the system. *GAS* receives a medium score for this category as this is a minor detail that could be overcome by a more experienced developer. *Opsive* and *Invector* both score low in terms of modularity, as the only modular aspect of the design is their ability and skills systems respectively.

5.3 Prebuilt Feature Integration

Mosaic was the only system to support effortless prebuilt feature integration. Prebuilt packages containing features for *Mosaic* can easily be imported into the project like any other package. That is all it takes to fully incorporate it into your project. From there the features can be dropped onto any core. *GAS* received a score of none as this is not a supported feature. *Opsive* and *Invector* both receive a score of low, as the feature does exist, but it requires a fair bit of setup. This setup often includes automatic code generation and user modification of the animator components.

5.4 Custom Feature Development

Mosaic and *GAS* both received a rating of easy for this category. Both of these systems have been designed from the ground up to support custom feature development and have been proven to do so through various projects. *Opsive* and *Invector* both receive a score of limited as only a single aspect of these systems can be extended to create custom features, and these are heavily constrained to the capabilities of the systems.

5.5 Codebase Scalability

In terms of codebase scalability *Mosaic* receives a score of High. Throughout the development of Echoes In The Mists, at no point did the project ever suffer from complexity creep due to the weight of its multitude of systems. *Mosaic* allowed us to rush bad code out the door, without any worry about far reaching consequences. All of the rushed code was then able to be replaced with engineered solutions without any hassle. *GAS* also received a score of high. Its high scalability is evidenced by its increasing popularity amongst large scale studios. *Opsive* and *Invector* receive a score of low as their high coupling and reliance on animators makes concurrent development amongst programers very difficult.

5.6 Target Users

Mosaic's target users are developers. While designers are likely to become familiar with the utility *Mosaic* affords them, it requires a developer to create the features they will be interfacing with. The same goes for *GAS*. *Opsive* and *Invector* are both targeted to designers, as they are predominantly sets of completed gameplay that can help designers get a kickstart on game development.

5.7 Networking Support

Out of the box, *Mosaic* does not provide networking support. *GAS* provides full networking support for systems designed with it. *Opsive* and *Invector* both provide networking support, but as paid add ons. These add ons are not guaranteed to be compatible with all of their prebuilt features.

5.8 Character Controller

Mosaic and *GAS* do not provide character controllers, rather leaving it up to the developer to add their own. *Mosaic* could in fact encapsulate *Opsive* or *Invector*, and use them as a specific behavior. *Opsive* and *Invector* are both built to be character controllers.

5.9 Ability System

Mosaic does not provide the standard set of tools you would expect an ability system to provide. Both *GAS* and *Opsive* provide ability systems, although *GAS* is much more robust. *Invector* provides what it calls a skill system to add some extensibility, but this is not comparable to the other offerings.

5.10 Model Swapping

Mosaic and *GAS* do not support model swapping, as the model itself is not explicitly part of the system. *GAS* provides gameplay cues which can be used for visuals, but does not handle models. *Opsive* and *Invector* both support model swapping with any humanoid rig.

Mosaic being the lightest-weight out of all offerings in terms of code and features, is set apart by its high reusability, and its effortless pre-built feature integration. *Mosaic* scored high across the board in terms of architecture characteristics, while at the same time lacking all of the additional features similar systems boasted. *Mosaic* was built from the ground up to allow for cross compatibility between developer created features and projects, whereas the other offerings were designed to serve more specific game-development oriented purposes. This makes *Mosaic* great for rapidly iterating through features.

GAS does not support any pre-built feature integration, so it does not support the preservation and sharing of features between projects and developers. On the other hand it does support both networking and a robust ability system out of the box, giving it a head start on that requires its base features.

Opsive and *Invector* share nearly identical feature sets, and both target very similar use cases. The only difference is the lack of an ability system for *Invector*. Instead it has what it calls a skill system, which is the only method for extending *Invector* and is less functional than *Opsives* abilities. Both of these scored by far the lowest in regards to architectural features. *Opsive* and *Invector* are paid assets, with charges for additional features. These both target designers rather than programmers and are particularly useful as a starting point for devs making games that match their supported feature set.

6. Discussion

The taxonomy indicates that *Mosaic* occupies a unique space in regards to gameplay systems. While tools like *Opsive* and *Invector* offer an out of the box solution for designers, *Mosaic* targets developers seeking full control and modularity of their systems. Unlike *GAS* which was designed to be as robust as possible, providing solutions for specific genres, *Mosaic* embraces its simplicity and modularity.

This makes *Mosaic* particularly useful for:

- Preservation and re-use of features
- Projects with non-standard requirements
- Developers making projects that need to scale over time
- Communities and developers who would like to share or monetize gameplay features.

Mosaics current usability is limited for:

- Designers who need pre-built features out of the box
- Teams looking for a solution with built in networking support

Mosaic's high scores in architectural categories and low scores in feature categories are a reflection of its intended use case. *Mosaic* doesn't provide features, it provides the tools developers need to develop features that are not only scalable, but fully cross compatible across projects and teams that use the systems. Its lack of

networking, character controllers, and ability systems isn't necessarily a weakness, it reflects a philosophy of not imposing structure unless absolutely necessary. This narrower scope of concern even allows it to be more easily integrated into an existing toolset, avoiding duplication.

This analysis has highlighted a variety of areas of growth for *Mosaic*.

- **Visual Feedback Integration:** A system akin to *GAS* Gameplay Cues would help integrate the visual style of assets into a given project.
- **Model Swapping:** Prebuilt systems for humanoid model swapping, as seen in *Opsive* and *Invector*, would make *Mosaic* features gameplay ready with just drag and drop functionality.
- **Networking Support:** Enforced network support for gameplay features would help immensely with creating multiplayer experiences.
- **Custom Editor Windows:** Making *Mosaic* feel as integrated as possible into the engine would help drive developer confidence in the systems.

These areas highlight opportunities for *Mosaic's* evolution, without compromising on its simplicity or modularity.

For developers, *Mosaic* offers a clean and powerful foundation for building out their software architecture. Its focused design allows for faster iteration, easy onboarding, and a reduced technical debt as the projects progress. It allows for the reuse of gameplay features, turning the cost of developing a feature into an investment in the studios feature library. This library can either be used by the team, or sold to other developers and hobbyists as a separate revenue stream.

Mosaic occupies a unique place in the market, and opens new and unique opportunities that have yet to be explored.

7. Conclusions

This research categorizes *Mosaic* alongside its closest piers, that of ability systems and character controllers. Using a Taxonomy of 10 architecture characteristics and feature coverage, the findings highlight that *Mosaic* targets a need not yet addressed by other products on the market.

Compared to systems like *Opsive*, *Invector*, and *GAS*, *Mosaic* takes a minimalistic approach, positioning it as a framework that unifies the gameplay foundation layer, allowing for both scalable development, and the preservation and reuse of developer made assets.

The analysis suggests that *Mosaic* is best utilized by teams who have technically skilled gameplay programmers on their teams.

Future development of *Mosaic* may involve extending *mosaics* built in capabilities for visual feedback, networking support, and other features that could broaden its utility without compromising the unique position it currently holds in the market.

8. Future Work

This paper aims to quantify the utility of *Mosaic* by creating a taxonomy around similar products such as character controllers and ability systems. There is much more to explore in terms of how systems like *Mosaic* can impact the game development process, what we can learn from them, as well as how these systems can be improved.

Developer Feedback

These systems were analyzed through documentation guided by some hands-on experience. Interview developers and gathering feedback would add help introduce a more human-centered perspective to the research.

System Impact

Future work could analyze how these systems impact development timelines, designs, and the technical complexity across a variety of real world cases.

Potential Growth

This research brought to light potential areas of improvement within *Mosaic*, such as visual effect integration, such as Unreals Gameplay Cues.

References

Dawe, M., Gargolinski, S., Dicken, L., Humphreys, T., & Mark, D. (2019). Behavior Selection Algorithms. In S. Rabin (Ed.), *Game AI Pro 360* (1st ed., pp. 1–14). CRC Press. https://doi.org/10.1201/9780429055058-1

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (2016). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gameplay Ability System for Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community. (n.d.). Retrieved January 28, 2025, from https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-ability-system-for-unreal-engine

GDC 2025. (2019, August 2). *Data-Driven Dynamic Gameplay Effects on For Honor* [Video recording]. https://www.youtube.com/watch?v=JgSvuSaXs3E

Gregory, J. (2019). Game Engine Architecture. CRC Press, Taylor & Francis Group.

Invector. (n.d.). *Third Person Controller—Basic Locomotion*. Retrieved May 7, 2025, from https://www.invector.xyz/copy-of-third-person-documentation

Kevin. (2023, September 7). *Answer to "How are character controllers built upon complex gameplay systems?"* [Online post]. Game Development Stack Exchange. https://gamedev.stackexchange.com/a/207110

Nystrom, R. (n.d.). *Component · Decoupling Patterns · Game Programming Patterns*. (n.d.). Retrieved January 28, 2025, from https://gameprogrammingpatterns.com/component.html

Skypjack (n.d.). *entt: Gaming meets modern C++—A fast and reliable entity component system (ECS) and much more*. (n.d.). Retrieved May 8, 2025, from https://github.com/skypjack/entt

Ultimate Character Controller. (n.d.). Opsive. Retrieved May 5, 2025, from https://opsive.com/support/documentation/ultimate-character-controller/