From Monoliths to Modularity: Reimagining Game Development Through ECS and Data Streams

1. Introduction

Game development has undergone a rapid evolution in the past two decades. What once required intimate knowledge of graphics APIs and hardcoded game logic is now accessible through powerful, commercial-grade engines like Unreal Engine and Unity 3D. These platforms have enabled the rise of indie studios and massive open worlds alike. But with their growth came a new kind of complexity—one deeply tied to rigid architectures, bloated tooling, and a growing disconnection between creativity and execution.

This research presents a different path forward: a fully data-driven development environment where everything from logic to rendering, interaction, and behavior is expressed through configurable data, not embedded in fixed code. It's a system built on modular principles, visual editing, and dynamic composition—a framework that rethinks how we build games from the ground up.

Rather than being trapped in an editor-centric, monolithic workflow, developers using this framework are empowered to construct, adapt, and evolve game systems organically, using flexible pipelines, reactive systems, and rich metadata. The goal is not only to make game creation faster and more efficient—but to make it smarter, more dynamic, and ultimately more fun.

The system described introduces a data-centric, reactive game development framework. Unlike traditional engines, where core functionalities like the rendering pipeline, UI behavior, or game logic are tightly bound to engine internals, this architecture aims to externalize game behavior and system definitions into composable data structures—processed by interpretable, reactive pipelines.

At the core lies a sparse-based Entity Component System (ECS), which favors flexible runtime transformations of entities without requiring archetype reallocation. This contrasts with archetype-based ECS, which enforces structure at the cost of runtime flexibility. The sparse ECS is tightly integrated with a reactive data streaming layer inspired by concepts from ReactiveX (Rx)—an open standard for composing asynchronous and event-based programs using observable streams. By combining both, entities become stream-reactive, and systems can respond to changes in real-time without polling or frame-bound evaluation.

Game logic, UI, and behavior are all represented through visual node graphs, where nodes act as transformation units, consuming and emitting typed data streams. These graphs are defined via modular files (e.g., Alchemy Files) and compiled into runtime-executable code—either directly in a target language like C++, or via a binary intermediate format similar to WebAssembly (WASM) or .NET IL (Intermediate Language). This enables high-performance execution while retaining platform flexibility and introspectability in editor environments.

In practice, the development environment supports:

- Visual logic authoring through node-based graphs
- Reactive UI definition using data flows and state transformations
- Toolchain integration via a modular build system based on simple but powerful package management like npm (Node Package Manager) or cargo
- CLI ↔ GUI Interoperability through Process Interop, allowing both programmatic and interactive usage of tools
- Semantic metadata support for describing implicit relationships (e.g., materials, physics interactions, sensory feedback)

Modern game engines offer an impressive set of tools—but they are bound by legacy assumptions:

- Unreal Engine, while modular on paper, enforces a monolithic structure that makes custom pipeline or system replacements impractical
- Unity has introduced modular packaging through the Unity Package Manager, yet its core runtime and editor remain closed-source and static in architecture

This system is designed to solve what these engines cannot:

1. Full system composability – Swap out rendering, physics, or logic modules as data-driven components
2. Developer-centric toolchains – Empower teams to build tools, pipelines, and systems in the languages and workflows they know (e.g., CLI, node graphs, peer-dependencies)
3. Future-facing extensibility – Built-in support for metadata and reactive patterns enables emergent gameplay mechanics, AI integration, and sensory simulation for VR, AR, and beyond

By shifting the paradigm from "code-bound mechanics" to "data-defined behavior", this framework enables a new category of game development—one that's flexible, modular, and inherently expressive. Whether used as a standalone engine or a high-level plugin for existing runtimes like Unreal, its purpose is to bridge creativity and execution through data

2. Design Principles

The foundation of this framework is built on five guiding principles: data-centricity, modularity, reactive composition, visual accessibility, and future readiness. Each of these principles is designed not only to solve specific limitations of existing engines, but to empower developers with a more expressive, flexible, and maintainable way to build interactive worlds.

Data-centricity means that the behavior, state, and transformation of systems are defined through structured data rather than hardcoded logic. Instead of writing thousands of lines of engine-bound script or C++, developers work with declarative data structures that describe what should happen, not how to do it. This approach makes logic portable, editable, and introspectable—by both machines and humans. It also lays the groundwork for automated validation, introspection, and transformation tooling.

Modularity ensures that every subsystem—rendering, animation, physics, AI, UI—can be developed, replaced, or extended without modifying the core. By separating concerns into self-contained modules that communicate through well-defined data contracts, the system achieves a level of composability that monolithic engines lack. This separation also enables teams to build isolated feature stacks and experiment without risking overall stability.

Reactive composition ties these pieces together into a living dataflow. Inspired by frameworks like ReactiveX (Rx) and state propagation systems found in frontend development (e.g., React, Svelte), the system allows game state to be modeled as a set of interconnected streams. These streams respond automatically to changes—whether input events, physics updates, or AI decisions—and propagate their effects through the game world. This eliminates the need for polling loops, tightly coupled systems, or global state managers.

Visual accessibility is key to bridging the gap between technical implementation and creative iteration. The development environment allows logic, UI, and data pipelines to be authored through node-based graphs. These visual editors offer the full power of the underlying system without requiring deep programming expertise, making it easier for designers, artists, and hybrid roles to engage with system behavior directly. Nodes are modular and reusable, defined in external files that describe input/output types, transformation rules, and (optionally) associated runtime code.

Future readiness speaks to the framework's long-term vision: supporting not only today's development workflows, but also those of tomorrow. This includes rich metadata for simulating emergent behavior (e.g., sound propagation, material interaction), modular runtime translation into target languages (C++, C#, WASM), and openness to evolving input/output technologies such as haptics, neural input, or environmental simulation. By abstracting behavior and content into portable data, games built with this system are inherently more adaptable to new hardware and new platforms.

These principles are not isolated—they reinforce each other. Modularity enables reactive pipelines; reactive behavior benefits from declarative data; visual editors emerge naturally when systems are data-defined. Together, they form a cohesive foundation for building expressive, future-ready game systems with precision and creative freedom

3. System Architecture Overview

The architecture is divided into three primary layers: Authoring, Pipeline, and Runtime. Each is designed to be modular and interchangeable, enabling development teams to tailor the system to their needs.

1. **Authoring Layer**: This is where all creative input happens. Developers, designers, and artists use visual editors, structured files (Alchemy Files), and reactive templates to define game logic, UI, components, metadata, and behavior. Graphs and component definitions are version-controlled and

human-readable

2. **Pipeline Layer**: The build pipeline transforms the authored data into executable modules. It validates node graphs, resolves dependencies, and compiles graphs and definitions into intermediate or target code. The build process uses modular CLI tools, compatible with GUI environments through Process Interop

3. **Runtime Layer**: At runtime, compiled modules are loaded and executed. The ECS processes entity data, while reactive systems handle event propagation and state updates. The runtime can be standalone or bridge into existing engines via compiled modules and adapters

Each layer is interchangeable. Developers can swap out the pipeline tools, embed only parts of the runtime, or create custom authoring environments. This architecture favors long-term extensibility, multi-platform support, and deep tooling integration across the development lifecycle

4. Visual Data Modeling

At the heart of the authoring experience lies the node-based visual editor, designed for constructing reactive data pipelines, defining logic, composing systems, and binding components—all without writing code. These visual graphs are not just UI elements—they define real, executable behavior.

Each graph consists of nodes representing:

- **Inputs**: data sources like user input, entity components, timers

- **Transformers**: processing steps (e.g., math, filtering, mapping, branching)

- **Actions**: side-effects like modifying entities, playing sounds, triggering animations

- **Outputs**: data sinks or events passed to other systems or pipelines

Graphs can define full systems (e.g., combat handling, quest logic), pipelines (e.g., inventory filters, UI rendering flows), or reactive behaviors (e.g., animation state transitions). Graphs can be nested and reused, supporting modularity and composition.

Each node is defined in a file format (Alchemy File) that describes:

- Input and output types
- Data validation rules
- Runtime bindings or generated code
- UI metadata (for editor representation)

Node libraries are created by bundling multiple node definitions into a portable archive. Developers can create custom node libraries, share them via packages, and extend the editor's capabilities dynamically.

Because graphs are stored in structured, serializable formats, they can be validated, transformed, and compiled into performant, optimized code for any target language. This enables consistent behavior across platforms and development environments.

To accelerate development and reduce friction, pipelines and systems authored in graphs can also be represented textually or extended via embedded scripting—similar to how Warcraft III's World Editor allowed visual and script-based trigger definition. This empowers developers of all backgrounds to contribute to logic creation, whether visually, textually, or through custom tooling

5. Reactive Systems

In traditional game architectures, behavior and state changes are typically evaluated in tight, imperative loops—where every update cycle involves checking for input, evaluating AI, applying physics, and rendering the frame. This model, while effective in certain cases, often leads to bloated update code, complex interdependencies, and non-deterministic bugs.

This framework replaces the imperative loop model with a reactive stream-based system. At its core, the engine combines a sparse-based Entity Component System (ECS) with principles from Reactive Programming, particularly the ReactiveX (Rx) family of libraries. The ECS is responsible for modeling the structure and relationships of entities, while reactive streams manage how data flows through those entities and systems in real time.

Every component property and input source becomes part of a data stream—an observable unit that can be transformed, combined, filtered, or delayed using a standard set of operators. For example:

- A stream representing player input can be filtered and debounced before triggering a movement system

- A health component can publish a stream of value changes, which UI components subscribe to for live updates

- Sensor nodes, such as triggers or vision cones, emit detection events to downstream AI systems

Streams are composed declaratively using operators such as map, filter, merge, debounce, sample, withLatestFrom, etc., enabling precise control over system logic without creating brittle, monolithic logic blocks. This also allows events and state to be propagated across systems without explicit coupling.

Reactive systems are defined visually as well as in code. Pipelines created in the node editor translate directly into reactive dataflows, and developers can observe live data as it flows through the system—debugging, modifying, and re-routing without halting execution. This drastically reduces iteration time and opens the door to novel gameplay architectures such as declarative AI, real-time data analytics, or collaborative sandbox worlds

6. Component & Entity Authoring

Entities are not predefined objects but constructed compositions of components. Each component is a structured definition of data that expresses a particular aspect of an entity: position, inventory, material, faction, sensory attributes, etc. Components are fully user-definable and described in Alchemy Files using a structured format with typed fields.

Each field can declare:

- Data type (primitive, enum, vector, custom)

- Default value
- Metadata (e.g., UI description, editor hints)
- Reactive bindings (optional)

Components can also include metadata tags for simulation purposes. For example, a MaterialComponent might include acoustic, thermal, or olfactory properties used in emergent systems.

Entities are assembled by combining components and assigning them an ID (or UUID). Collections of entities can be grouped into prefabs, which are reusable templates composed of components, initial values, and optionally, logic attachments.

The entity authoring workflow is fully visual, but under the hood, entities and components are stored in source-controlled, structured files and compiled into ECS modules. The system ensures type safety, efficient memory layout, and runtime reflection, enabling advanced features like save states, modding, and data-driven instancing

7. Implicit Gameplay & Metadata

Games have traditionally relied on explicitly programmed interactions: a door opens if the player has the key; a fire burns only if a script tells it to. But much of real-world behavior emerges from the properties of objects and their relationships—not from imperative rules. This framework embraces implicit gameplay by enabling metadata-driven world logic.

Components can carry semantic metadata—descriptions that help systems infer behavior without hardcoding it.

Examples:

- A door tagged as Material:Wood and Burnable:true will react to nearby fire
- A surface tagged Material:Metal and Hollow:true will echo footsteps
- An item tagged Smell:Strong will affect stealth mechanics via olfactory AI

These metadata fields are not tied to any specific system. Instead, systems subscribe to relevant metadata patterns and process them reactively. For example, a fire system subscribes to Burnable:true components within range; a sound propagation system analyzes Material:Resonance properties to generate echo effects.

This design allows new gameplay to emerge organically from data, and systems to scale and evolve without code duplication. Developers can create new interactions simply by combining and tagging components appropriately.

Metadata also supports future-facing features:

- AI systems interpreting environmental cues
- Environmental storytelling through dynamic ambiance
- Accessibility adjustments (e.g., translating smell or haptic cues)
- VR and sensory simulation based on material semantics

Implicit gameplay reduces scripting overhead, improves immersion, and future-proofs content by enabling general-purpose systems to respond to new entities dynamically

8. Tooling & Build Pipeline

The framework includes a robust, modular build system powered by simple but yet powerful package management environments like npm (Node Package Manager) or cargo, and custom tooling.

This build pipeline is responsible for:

- Resolving peer dependencies (plugins, node libraries, component sets)
- Validating and transforming visual graphs
- Compiling graphs into source or intermediate code
- Bundling runtime modules
- Emitting platform-specific artifacts (e.g., C++, C#, WASM)

Tooling is structured as independent CLI packages that can also be embedded in GUI environments. This dual-mode architecture is enabled by Process Interop, a system for running CLI processes as embedded services with exposed APIs.

For example:

- A component editor GUI calls the CLI validator to check schemas
- A build pipeline GUI invokes the bundler to export for multiple platforms

This approach encourages modularity, testability, and IDE-agnostic tooling. Developers can work in Visual Studio (Code), a full-featured GUI, or even headless CI environments—using the same toolchain.

Publishing and sharing modules is handled via custom registries like an npm server via GitHub Packages or self-hosted cargo instance. Projects can declare exact versions or semantic ranges, and builds are fully reproducible. Artifacts such as compiled pipelines, prefabs, or node libraries are cached and portable

9. Runtime Integration

The runtime might be integrated into existing engines (e.g., Unreal) or run standalone. The integration layer consists of runnable modules compiled from the data graph and component definitions. These modules are lightweight and reactive—they do not require polling loops or tight engine coupling.

For integration:

- Visual pipelines are compiled into engine-specific code (e.g., C++/Blueprint bindings for Unreal)
- Reactive systems interface with engine events (e.g., input, collision)
- Entities are synchronized via a runtime ECS bridge

The architecture supports live-reloading of pipelines and hot-swapping of system logic. This enables designers to iterate in real-time, without full engine rebuilds.

In standalone mode, the runtime executes compiled systems in a headless or windowed environment with modular render, audio, and physics backends

10. Use Cases & Scenarios

This framework supports a wide range of development scenarios, each benefiting from its data-driven architecture and reactive composition:

- **Game Development**: Whether building 2D pixel art games, 3D narrative experiences, or physics-based simulations, the framework enables rapid iteration and emergent gameplay mechanics. Designers can author systems visually, while engineers can fine-tune or extend behavior at the code level. Modular pipelines allow teams to separate concerns between gameplay, UI, AI, and simulation

- **Simulation & Training**: Industrial or educational simulations often require precise data modeling and reactive systems for replicating real-world interactions. The component metadata system supports fine-grained physical, acoustic, or chemical modeling, while reactive pipelines allow scenario logic to respond dynamically to trainee input or external data feeds

- **Tooling & Editors**: Developers can build custom authoring tools (e.g., dialogue editors, quest designers, inventory configurators) using the same node-based architecture and ECS foundation. These tools can be embedded in larger applications or run as standalone visual editors

- **Plugins for Existing Engines**: The runtime can serve as a logic and data-processing layer inside other engines (e.g., Unreal). Projects can replace or augment internal systems—such as AI, inventory, dialogue, or environmental logic—using modular, reactive graphs without touching core engine internals

- **Live Prototyping & Rapid Design**: The combination of hot-reloadable modules, live-editable graphs, and observable streams allows for real-time iteration. This is ideal for exploring new mechanics, balancing game loops, or conducting user testing during early development stages

- **Collaborative Design Workflows**: Because visual logic and systems are stored in structured, version-controllable files, cross-functional teams (e.g., designers, programmers, artists) can work simultaneously without merge conflicts or dependency errors. Visual graph diffs, schema validation, and type-safe composition support efficient collaboration

- **Academic & Experimental Research**: For researchers working on interaction design, generative systems, or new input methods (e.g., brain-computer interfaces), the framework provides a flexible, low-friction environment for integrating novel systems without having to build a full engine stack from scratch

11. Roadmap & Extensibility

The long-term roadmap for this framework focuses on deepening system maturity, expanding platform support, and enabling next-generation interaction models. Key milestones include:

- **Advanced Input Modalities**: Integration of haptic feedback, neural input (e.g., EEG/BCI), eye-tracking, and gesture systems will allow developers to model richer sensory feedback and user control. This benefits both accessibility and immersion—especially in XR and accessibility-sensitive projects

- **Generative Systems Integration**: Connecting the framework to procedural content generation (PCG) or AI-assisted design tools (e.g., GPT-powered narrative generators, terrain synthesizers, or visual design assistants) will enable new workflows. Developers could define constraints in metadata and let systems auto-generate content or assist in logic design

- **Online Collaboration**: Cloud-hosted authoring environments, shared graph editing, and project sync tools will allow distributed teams to work on the same logic graphs or ECS systems concurrently. This supports modern pipelines that mirror tools like Figma or GitHub Codespaces.

- **Reusable Modules & Shared Asset Graphs**: A marketplace or registry of public node libraries, component definitions, and system templates would make common patterns (e.g., RPG inventory, 2D platformer physics, quest logic) reusable across projects and studios

- **Performance Profiling & Visualization**: Integrated profiling tools for inspecting reactive data flows, ECS performance, and pipeline latency. This includes heatmaps, per-frame diagnostics, and stream-level debugging for optimizing system performance across devices

- **Custom DSL Support**: A domain-specific language (DSL) for defining node behavior, visual graph logic, or system rules. This DSL would compile to graph data or code and could be embedded in visual editors to allow hybrid visual/textual authoring

- **Extended Export Targets**: Compilation targets for additional runtimes beyond C++/C#/WASM, including embedded platforms (e.g., microcontrollers for physical installations), or cloud-execution models for multiplayer simulations and metaverse-style experiences

- **Plug-and-Play Editor Templates**: Ability to create fully customized editor environments for specific game genres, teams, or workflows—tailoring the visual editing experience through configuration and theming rather than source modification

Each roadmap item is designed to reinforce the core principles of the framework: modularity, data-centricity, and creative accessibility. By building toward a fully decoupled and extensible ecosystem, the framework aims to evolve into a flexible foundation that can power games, tools, and simulations for years to come.

Future development includes:

- Support for haptics, neural input, and alternative controllers
- Integration with generative systems (e.g., AI content authorship)
- Online collaboration and cloud-synced authoring
- Shared asset graphs and reusable templates
- Performance profiling tools for reactive pipelines
- DSL authoring for node definitions and graph composition

All features will remain modular. Core packages can be replaced, extended, or embedded in domain-specific solutions.

12. Summary & Vision

The goal of this framework is to democratize and modernize game creation—removing barriers between code and creativity, and offering a path toward smarter, more dynamic content. By turning logic, structure, and interaction into portable, introspectable data, it empowers creators to build systems that scale, adapt, and evolve.

In a future where hardware, input methods, and player expectations change rapidly, this architecture lays a foundation for resilient, expressive, and intuitive world-building—where behavior emerges not from code alone, but from the relationships between things