Fast Data Plane (FDP)

Fast Data Plane (FDP) is a high-performance, deterministic Entity–Component–System (ECS) engine designed for large-scale, real-time simulations and data-intensive runtime environments.
It prioritizes **predictable performance**, **zero allocations on the hot path**, **determinism**, and **deep introspection** over convenience abstractions.

FDP is not a game engine in the traditional sense. It is a **simulation core** and **data execution engine** that can serve as the backbone for games, training simulators, digital twins, large-scale AI simulations, or distributed real-time systems.

Design Philosophy

FDP is built around several non-negotiable principles:

- **Performance must be predictable**, not just fast on average
- **Hot paths must be allocation-free**
- **Execution must be deterministic**
- **Data layout matters more than abstraction elegance**
- **Debugging and replay are first-class concerns**
- **Flexibility must not compromise the hot path**

These principles shape every architectural decision in the engine.

Core Features Overview

1. High-Performance ECS Core

FDP implements a **data-oriented ECS** optimized for modern CPUs:

- Components stored in **contiguous memory pools**
- Fixed-size memory chunks for cache efficiency
- SIMD-friendly iteration and filtering
- Extremely fast entity queries via bitmask matching
- No per-entity object overhead
- Constant-time component access by entity ID

Unlike archetype-based ECS designs, FDP uses **component-centric storage**, trading some per-entity contiguity for simpler, more flexible data management and faster random access.

---

2. Hybrid Managed / Unmanaged Data Model

A defining feature of FDP is its **explicit two-tier data model**:

Tier 1 – Unmanaged Components (Hot Data)

- Plain value types
- Stored in native memory
- Contiguous layout
- Zero garbage collection
- Intended for high-frequency updates (physics, simulation state, AI)

Tier 2 – Managed Components (Cold Data)

- Reference types
- Stored on the managed heap
- Accessed indirectly
- Intended for complex or infrequently changing data
    - Strings
    - Object graphs
    - Metadata
    - Configuration state

This separation allows FDP to:

- Keep the hot path extremely fast
- Still support rich, expressive data where needed
- Avoid forcing everything into unsafe or unnatural data representations

Most ECS engines force a single model. FDP embraces **controlled heterogeneity**.

---

3. Zero-Allocation Hot Path

FDP is designed so that **steady-state simulation produces no managed allocations**:

- No heap allocation during frame updates
- No iterator allocations
- No hidden boxing
- No LINQ
- No captured lambdas
- No implicit enumerators

All critical iteration is performed via:

- Value-type iterators
- Direct memory access

- Pre-allocated buffers
- Explicit pooling where needed

The result:

- No GC spikes
- Stable frame times
- Predictable latency
- Suitable for real-time and soft real-time systems

---

4. SIMD-Accelerated Entity Queries

Entity matching in FDP is based on **wide bitmasks**:

- Each entity carries a fixed-width component signature
- Queries are expressed as bitmask predicates
- Matching is performed using vectorized CPU instructions

This allows:

- Extremely fast filtering
- Branch-free inner loops
- Efficient scanning of very large entity sets

In practice, millions of entity checks per frame are feasible without query overhead becoming dominant.

---

5. Deterministic Phase-Based Execution Model

FDP enforces a **strict, explicit execution model** based on phases:

- Each frame is divided into well-defined phases
- Each phase defines:
  - What can be read
  - What can be written
  - What structural changes are allowed
- Illegal access patterns are detected and rejected (in debug builds)

Benefits:

- Deterministic execution order
- No hidden data races
- Lock-free parallelism
- Clear mental model for system interactions

This model replaces ad-hoc synchronization and implicit ordering with **architectural guarantees**.

---

6. Built-In Multithreading Without Locks

Parallelism in FDP is achieved through **structural guarantees**, not fine-grained locks:

- Read-only phases can run fully in parallel
- Write phases are isolated and synchronized
- Structural changes are deferred and applied at safe points
- No locking on component access in hot loops

This approach:

- Scales well across cores
- Avoids lock contention
- Preserves determinism
- Keeps systems simple and analyzable

---

7. Integrated Event System (Managed & Unmanaged)

FDP provides a **high-performance event bus** aligned with its data model:

- Unmanaged events for high-frequency signaling
- Managed events for complex or low-frequency communication
- Events are phase-aware
- Events can be recorded and replayed

This allows:

- Zero-allocation event processing where needed
- Expressive event payloads where performance is less critical
- Unified event handling across the engine

---

8. Flight Data Recorder (Deterministic Replay)

One of FDP's most distinctive features is its **Flight Data Recorder**.

It provides:

- Frame-by-frame recording of the entire ECS state
- Deterministic replay of simulation runs
- Event replay synchronized with state
- Delta compression between frames
- Optional keyframes
- Asynchronous recording with minimal runtime impact

Key Characteristics

- Records raw component memory, not high-level objects
- Sanitizes unused memory to ensure determinism
- Highly compressible snapshots
- Suitable for continuous recording in production builds

Use cases:

- Debugging hard-to-reproduce bugs
- Simulation validation
- Offline analysis
- Regression testing
- Networking and synchronization research

Few ECS engines treat replay as a **first-class architectural concern**. FDP does.

---

9. Change Tracking and Delta Propagation

FDP tracks state changes at multiple levels:

- Per-entity
- Per-chunk
- Per-frame

This enables:

- Efficient delta snapshots
- Selective replication
- Fast detection of modified data
- Reduced I/O and serialization overhead

Change tracking is deeply integrated into the ECS core, not bolted on afterward.

---

10. Determinism by Construction

FDP is designed so that:

- Given the same inputs
- In the same order
- On the same architecture

…it produces the same outputs.

Determinism is supported by:

- Fixed update ordering
- Explicit phases

- Controlled parallelism
- Explicit event sequencing
- Deterministic memory snapshots

This makes FDP suitable for:

- Replays
- Lockstep simulations
- Validation and certification environments
- Scientific and industrial simulations

What Makes FDP Unique

Compared to other ECS engines, FDP stands out in several areas:

Compared to Unity DOTS

- Less restrictive data model
- Built-in replay
- Explicit phase safety instead of implicit job contracts
- No dependency on a specific engine ecosystem

Compared to Traditional C# ECS Frameworks

- True zero-allocation hot path
- Native memory usage
- SIMD-accelerated queries
- Deterministic execution guarantees

Compared to Custom Simulation Cores

- ECS-based structure with strong architectural rules
- Integrated tooling for replay and debugging
- Balanced support for both performance and expressiveness

FDP occupies a niche where **raw performance, determinism, and debuggability** are equally important.

Intended Use Cases

FDP is particularly well-suited for:

- Large-scale simulations
- Training and military simulators
- AI and agent-based modeling
- Digital twins

- Deterministic multiplayer research
- Performance-critical game cores
- Real-time data processing pipelines

It is less suitable when:

- Rapid prototyping is the primary goal
- Maximum convenience is preferred over predictability
- Determinism is not required

Below is a **condensed, promotional-style summary section** suitable for an overview README.
It keeps the architectural intent and differentiators, but trims detail in favor of **clarity, impact, and scannability**.

Diagnostics & Inspectors (Overview)

Fast Data Plane includes a **built-in diagnostic and introspection layer** designed for data-oriented, high-performance systems.

Diagnostics are not external tools or ad-hoc debug helpers—they are **architecturally integrated** and operate directly on ECS data and event streams, without breaking determinism or performance guarantees.

Component Inspector

The Component Inspector provides structured visibility into ECS state:

- Inspect entities and their components
- View unmanaged (hot) and managed (cold) data distinctly
- Observe component values and structural composition
- Track state changes across frames

Inspection is phase-aware and safe, ensuring that runtime invariants are never violated.

Event Inspector

The Event Inspector exposes the engine’s event-driven behavior:

- Observe both unmanaged and managed events
- Inspect event payloads and ordering
- Correlate events with simulation frames
- Trace cause–effect relationships between events and state changes

This makes complex, event-driven logic transparent and debuggable.

Replay-Aware Diagnostics

Diagnostics integrate seamlessly with the Flight Data Recorder:

- Inspect live simulations or recorded replays
- Step through historical state deterministically
- Analyze rare or timing-sensitive issues offline

This enables **time-travel debugging** without special instrumentation.

Designed for Production Use

FDP diagnostics are:

- Allocation-free in read-only mode
- Deterministic and reproducible
- Phase-safe and non-intrusive
- Suitable for long-running and headless systems

They can be left enabled in performance-sensitive environments and form a foundation for monitoring, validation, and operational analysis.

Why It Matters

In data-oriented, parallel systems, understanding *what the data is doing* matters more than inspecting call stacks.

FDP's diagnostic tools provide:

- Deep observability
- Deterministic insight
- Confidence in correctness at scale

They complete FDP's vision of a simulation engine that is not only fast and deterministic, but also **transparent, debuggable, and operable**.

Architectural Analysis and Features

Key Features of the FDP Engine

- **Hybrid Managed/Unmanaged Component Storage:** FDP stores high-frequency data in unmanaged structs and low-frequency/complex data in managed objects. Unmanaged components (Tier 1) reside in contiguous native memory (using raw OS allocation) for maximum cache locality and SIMD-friendly layout, while managed components (Tier 2) are kept in sparse arrays for data that requires garbage collection[[1]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L12-L14). This hybrid model gives the performance benefits of struct storage while still allowing

flexibility for complex or reference-based data when needed[[2]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L14-L17).

- **AVX2-Optimized Queries:** Each entity's component composition is represented by a 256-bit mask. FDP leverages CPU intrinsics (AVX2) to compare these masks with query masks in a single instruction[[3]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L2-L5). This means systems can filter entities extremely fast (O(1) per entity check in practice), scanning millions of entities per second without branch-heavy logic[[3]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L2-L5). As a result, entity queries (e.g. "find all entities with components A and B") are highly efficient.
- **Zero-Allocation Hot Path:** The engine is designed such that the core game loop generates **no garbage collector pressure**. It uses a custom native memory allocator and object pooling so that once the simulation is running, no new heap allocations occur during frame updates[[4]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L2-L5)[[5]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L15-L16). Iterators for entities are implemented as stack-allocated structs (ref structs) instead of heap-allocating C# iterators, and practices like using foreach, LINQ, or lambda allocations in the hot path are explicitly prohibited[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45). This strict zero-GC philosophy ensures consistent performance and avoids GC pauses during critical real-time simulation frames[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45).
- **Built-in Flight Recorder (Deterministic Replay System):** FDP includes a **Flight Recorder** feature for recording and replaying simulation state deterministically. It can capture the entire state of the ECS at a high frequency (e.g. 60 Hz) without slowing down the simulation[[4]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L2-L5)[[7]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L17-L20). This is achieved with asynchronous, double-buffered snapshotting – one buffer is filled with snapshot data while the previous buffer can be compressed or written out, avoiding stalls. The recorder uses delta compression (recording only changes after a keyframe) and automatically serializes component data via JIT-compiled expression trees (i.e. it generates efficient serialization code at runtime)[[7]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L17-L20). The result is a replay system where every frame's state (and even events) can be logged and later reproduced exactly, which is invaluable for debugging or synchronizing state in distributed simulations.
- **Strict Phase-Based Execution Model:** FDP enforces a deterministic update order through defined phases (e.g. *Network Ingest*, *Simulation*, *Post-Simulation/Presentation*). Each phase has rules about what operations are permitted, ensuring no race conditions without using locks[[4]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L2-L5)[[8]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L21-L22). For example, one phase might allow only reading certain components, another allows writing, etc., according to configured permissions. The engine will throw an error (e.g. a WrongPhaseException) in debug mode if a system tries to, say, write to a component in a read-only phase[[9]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L20-L28). This phase system, combined with the component access permissions, guarantees **determinism and thread safety** – e.g. one phase can run multi-threaded physics updates with read/write separation, and a later phase applies any structural changes in a single-threaded "sync point". This lock-free phased approach ensures that parallel tasks don't collide, without the overhead of fine-

grained locking[[9]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L20-L28).

- **Multithreading and Parallel Iteration:** The engine is built to take advantage of multiple cores. Component data being laid out in chunks allows the scheduler to split work by chunks or other criteria so that multiple threads can iterate over different subsets of entities in parallel[[8]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L21-L22). Write operations are carefully controlled by the phase system (only occurring in phases where threads are synchronized) so that read-only phases can be fully parallelized. FDP also supports thread-safe command recording (deferring actions to the correct phase). Overall, the design allows scaling the simulation across threads while maintaining safety and determinism.

Internal ECS Architecture of FDP

Memory Model – Pools and Chunks: Internally, FDP's ECS uses a data-oriented design where each *component type* has its own contiguous storage pool. Rather than storing each entity's components together, the engine organizes memory by component type (a *sparse set* approach). For each component type, memory is divided into fixed-size chunks (pages) – for example, each chunk might hold up to 1024 components of that type[[10]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L33-L40). These chunks are allocated as needed from the OS (using low-level virtual memory allocation) so the storage can grow dynamically. Crucially, within a chunk, the components are laid out contiguously in memory, which maximizes cache locality for systems iterating that component. If a chunk is filled, a new chunk is allocated and linked to the pool's list[[11]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L35-L40). This means an entity is identified by an index, and that index maps to specific slots in each component pool's chunks.

Entity Composition and Indexing: Every entity in FDP has an ID/index and an associated **bitmask** of which components it contains. FDP uses a 256-bit mask for each entity to represent up to 256 distinct component types[[5]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L15-L16). This mask allows the engine to quickly check an entity's composition via bit operations. When an entity is created or destroyed or has components added/removed, the mask is updated. Systems can define a query mask (for the combination of components they are interested in) and use a fast bitwise comparison (augmented by AVX2 instructions) to filter entities in bulk[[3]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L2-L5). For example, if a system needs entities that have components A and B, the engine can compare the entity's 256-bit signature against the mask (A|B) extremely quickly in vectorized batches. This design differs from an archetypal ECS (like Unity DOTS) where entities with the same set of components are packed together; instead, FDP's approach is closer to a "sparse set" ECS where each component type array may have empty slots for entities that don't possess that component. The advantage is O(1) direct access to any entity's component by index and simpler memory management, at the cost of some unused space for absent components.

Deterministic Phased Execution: The execution of systems in FDP is split into distinct phases that correspond to different roles in the frame. A typical configuration might include phases such as **Network Ingest (Pre-Simulation)**, **Simulation (Game Logic/Physics)**, **Structural Sync (applying entity creation/destruction)**, and **Export/Presentation**. By design, write operations to certain data are only allowed in specific phases. For instance, during the Simulation phase, game logic systems may write to components they "own," while in the Presentation phase systems are only supposed to read state and output to rendering or audio (no writing to simulation state)[[12]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L22-L29)[[13]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L68-L70).

The engine's core tracks the current phase and can validate operations; any illegal access (like modifying a component in the read-only phase) will be caught in debug mode to prevent race conditions[[9]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L20-L28). This phase segregation is what enables FDP to run logic in parallel without locks – e.g. one phase might run multi-threaded without any writes (so threads don't conflict), and another phase might perform all writes in a single-threaded section (to apply queued changes). The phases act as implicit barriers, so by the time a parallel phase runs, any necessary data changes from prior phases have been safely applied. This design leads to a **deterministic update order** (every frame's operations occur in the same sequence relative to phases) and eliminates subtle timing bugs that could occur in an unstructured multithreading scenario.

Change Tracking and Versioning: (Related to the Flight Recorder and networking) FDP tracks modifications to components using version counters. Every time the global simulation tick advances, and when components change, the system updates version numbers per chunk and per entity[[14]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L124-L133). This dual-layer versioning (coarse chunk-level and fine entity-level) means systems or the networking layer can efficiently query "what changed since tick X" without scanning everything[[15]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L124-L132). Only chunks marked dirty (having any change in the last frame) and entities with updated version flags need to be considered for delta updates. This mechanism underpins both delta snapshots in the recorder and efficient network replication of state. By combining version checks with the component masks, the engine can skip entire blocks of memory when nothing in them has changed[[15]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L124-L132).

Comparison with Other ECS Systems

FDP's design can be contrasted with some well-known Entity-Component-System frameworks to highlight its unique pros and cons:

- **Unity DOTS (Data-Oriented Tech Stack) ECS:** Unity's ECS similarly emphasizes performance and contiguous data layout. Unity uses an *archetype chunk* model – entities with the same set of components are stored together in memory chunks. This yields excellent locality for groups of components used together by a specific entity type, whereas FDP's per-component pools give locality per component type. Unity historically allowed only unmanaged (blittable) components in its high-performance ECS; reference types were not stored in the main chunk. (Recent versions of Unity ECS introduced *managed components*, but these are stored separately in a global array and referenced by chunks, incurring extra lookup and GC overhead[[16]](https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Optimize%20managed%20components). Unity explicitly notes that managed components are slower and should be used sparingly[[17]](https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Unlike%20unmanaged%20components%2C%20Unity%20doesn%27t,less%20optimal%20than%20unmanaged%20components).) By contrast, FDP's *hybrid model* was built-in from the start: it cleanly separates hot, unmanaged data and cold, managed data into Tier 1 and Tier 2 storage. A benefit of Unity's pure-archetype approach is that all of an entity's core data is contiguous, which can make iterating over an entity's full state very cache-efficient and fast for tightly coupled data. FDP's approach trades some of that per-entity contiguity for flexibility – it can handle a wider variety of data types (e.g. strings, complex classes) and makes adding/removing components a simple flag change rather than moving an entity between archetype chunks. **Pros:** Both systems achieve near zero-allocation and high speed. Unity's ECS is deeply integrated with the Burst compiler and job system for optimal multithreading; FDP similarly attains parallelism through its phase system (without

requiring a separate code compilation step like Burst). FDP's advantage is in usability/flexibility – developers can use rich C# object types for infrequent components, whereas Unity ECS code is more restrictive. **Cons:** Unity ECS, being part of the Unity engine, benefits from tooling and a large ecosystem, whereas FDP is a custom engine library (likely used in a specific simulation context). Also, Unity's archetype model can be more memory-efficient for sparse components (no empty slots – entities simply don't appear in chunks for components they lack), while FDP's fixed slot indexing can lead to unused space when many entities don't have a given component type (though this is mitigated by chunking and isn't usually a major issue). Both systems require careful coding practices to avoid allocations; Unity enforces this by design (no GC types in components or jobs), and FDP enforces it via guidelines and runtime checks (e.g. disallowing certain operations in hot paths).

- **Entitas (and similar pure C# ECS frameworks):** Entitas is a well-known ECS framework in the Unity/.NET community that focuses on code generation to create efficient ECS boilerplate. It uses a more traditional approach with entities identified by an index and components stored in arrays or lists by component type (somewhat similar to FDP's pool-per-type) and bitmask flags for fast queries. Compared to FDP, Entitas operates entirely in managed memory – components are often classes or plain C# objects and may produce garbage unless carefully pooled. It doesn't have the sophisticated native memory management, so performance and cache locality are not as optimized. For example, adding a component in Entitas might involve creating a new object or resizing an internal array, which could trigger GC. FDP's use of unmanaged chunks and custom allocators gives it an edge in raw speed and consistent frame times (no GC hiccups). Also, Entitas does not natively provide features like a deterministic replay system or a strict phase-based execution; it leaves most of the game loop structure to the user. **Pros:** Entitas is high-level and easy to integrate – it generates code for getters/setters and events, making it pretty user-friendly. It was designed to reduce the friction of using ECS in Unity, and it's quite lightweight. FDP, on the other hand, is more complex but offers determinism and performance suited for large-scale simulations. **Cons:** Entitas and similar managed ECS libraries can incur runtime allocations and typically rely on the C# garbage collector (meaning you must be careful with memory if targeting performance). They also might not support multithreading out-of-the-box to the extent FDP does, whereas FDP's architecture is built for parallelism. In short, FDP sacrifices some simplicity for raw performance and advanced capabilities, while Entitas prioritizes ease of use over absolute performance.
- **Svelto ECS (or other data-oriented .NET ECS frameworks):** Svelto ECS is another modern C# ECS framework that, like FDP, emphasizes zero allocations and cache-friendly design. Svelto employs a "entity component engines" model and often uses structures of arrays internally; it even can use unmanaged memory for certain storages. In many ways, Svelto aligns with FDP's goals: it encourages splitting data into value types for performance and supports job-like parallel execution. However, FDP distinguishes itself with some unique features – notably the *Flight Recorder* and strict phase guarding – which typical ECS frameworks (Svelto included) generally do not have out-of-the-box. For instance, Svelto doesn't inherently record a frame-by-frame replay log of all state changes – that's a specialized feature of FDP. Also, FDP's two-tier component approach is somewhat more explicit; Svelto developers can certainly use reference-type components, but they won't be as optimized, similar to any managed data in .NET. **Pros:** Compared to FDP, Svelto ECS is an established open-source solution that can integrate into Unity or other C# projects, and it has a community around it. It provides patterns to avoid allocations (like using pools or pre-allocated native buffers), approaching the performance ideology of FDP. **Cons:** FDP's tightly integrated design (phases, event bus, replay, etc.) means that all parts of the simulation work in concert, potentially giving a more deterministic and complete solution for a custom engine. Generic ECS frameworks require the user to enforce determinism or recording if needed. Also, because FDP was inspired by Unity DOTS terminology[[18]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L6-L9), its API might feel familiar to those concepts,

whereas Svelto has its own patterns and terminology. One trade-off is that FDP being a hybrid system can handle high-frequency logic and low-frequency logic differently, which is a level of control many ECS frameworks don't expose as a built-in concept.

Summary of Pros and Cons: In summary, FDP stands out by combining **high-end performance techniques** (lock-free multithreading, SIMD acceleration, zero-allocation patterns) with **practical flexibility** (support for managed objects, built-in replay, dynamic event system). Compared to other ECS systems, FDP's pros include deterministic replay (a rare feature), strict safety checks (phase-based write protections), and the ability to integrate complex managed data without completely sacrificing performance. Its cons could include increased complexity – the programmer must understand phases and memory tiers – and a potentially higher memory footprint for sparse data (due to its slot-based storage). Other ECS frameworks might be simpler or have engine-specific tooling (as in Unity's case), but they may not achieve the same level of runtime performance or feature completeness for large-scale simulation use cases. For a system architect, FDP provides a robust, conceptually rich ECS core at the cost of being a more opinionated and specialized solution than generic ECS libraries.

Rationale for Managed vs Unmanaged Components (and Events)

FDP's **hybrid data design** is rooted in the observation that "not all data is equal"[[2]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L14-L17). Some components are fundamental to the simulation's tight loop – things like positions, physics state, or health values that update every frame. These are Tier 1 unmanaged components (structs) which are stored in native memory for speed. By keeping them unmanaged, the engine avoids any garbage collection overhead and guarantees these components have a contiguous memory layout. This makes iterating over, say, 10,000 positions and velocities very fast and predictable (data is tightly packed, enabling vectorized processing and cache prefetching).

On the other hand, some game data is inherently complex or used infrequently – for example, a player's inventory (which might be a list of item objects), or a mission description text, or other high-level state that doesn't update every frame. Trying to force such data into structs can be either impossible or counterproductive (e.g. you cannot put a variable-length list or a string with dynamic length into a fixed struct without indirection). For these cases, FDP provides Tier 2 managed components (classes or other reference types)[[2]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L14-L17)[[19]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L85-L93). They live in C#'s managed heap and the engine stores references to them in a special array (one per component type). This way, the system can still treat them as components attached to entities, but when you access a managed component you're doing an extra pointer lookup to the heap. The trade-off, as expected, is performance: accessing managed components is slower (an extra pointer chase and likely a cache miss) and can incur GC if you create or destroy these frequently[[20]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L68-L75). FDP mitigates this by expecting that Tier 2 components will be used for "cold" data that changes rarely or for data that simply must be reference types. By segregating the two tiers, the **hot path remains clean** – the presence of Tier 2 components doesn't significantly degrade the performance of the tight-loop simulation, because the high-frequency systems will mostly be working with the unmanaged tier. In short, **managed components exist to give developers flexibility for complex state without abandoning ECS structure**, but the engine encourages using unmanaged components for anything performance-critical.

The same philosophy extends to **events** in the FDP engine. The engine defines an Event Bus that can handle both unmanaged events and managed events, using a double-buffered approach. The reason is analogous: many events in a game/simulation (collisions, damage events, etc.) can be represented as small struct payloads (e.g. an event with an entity ID and a damage amount). These are Tier 1 events, which FDP stores in a native ring buffer or stream – they can be written and read with no allocations, and even processed in parallel using SIMD if needed[[21]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2684-L2686). However, some events might carry data that doesn't fit in a simple struct, such as a chat message event that contains a string and perhaps references to complex objects (or simply for ease, the user might want to define an event as a class). For these, FDP provides managed events (Tier 2 events) which are recorded and replayed via serialization. Managed events allow the system to capture things like "Player sent a chat: 'GG'" or other high-level messages that are not performance-critical. They do incur allocations (each event is a class instance, and during replay they have to be deserialized), but since such events are typically low-frequency (and often one-off per frame at most), the impact on the GC is minimal in practice. **Having both managed and unmanaged event streams gives the architect the choice**: time-critical, high-volume events can be done with zero GC cost, while complex events can still be included in the simulation flow at the cost of a bit of GC, all under a unified event bus interface[[22]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L50-L59)[[23]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L60-L68). The Flight Recorder actually captures both types of events in the replay log – it stores raw bytes for unmanaged events and uses a lightweight serializer for managed events[[22]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L50-L59)[[23]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L60-L68) – so that absolutely everything that happened in the simulation can be replayed deterministically.

In summary, **managed vs unmanaged components (and events)** exist because FDP acknowledges the need for a balance between **performance** and **functionality**. Purely unmanaged ECS designs (like earlier Unity DOTS versions) achieve great performance but at the cost of forcing all game state into bare-bones structs, which sometimes isn't feasible. FDP's approach gives a controlled escape hatch: you can use classes and other managed data where necessary, but you consciously pay a performance cost only for those specific components/events. Meanwhile, the core simulation loop remains as fast as if it were a pure unmanaged ECS. This design is especially useful in complex simulations or games where, for example, the core physics and gameplay are heavy-duty (and handled in Tier 1), but you still want to leverage higher-level .NET functionality (like complex object graphs, text processing, etc.) for the "glue" or peripheral systems at Tier 2. By splitting the data model, FDP keeps the **hot path "pure" and the cold path flexible**.

Enforcing Zero-Allocations in the Hot Path

The "Zero-Allocation" philosophy of FDP is not just a wish – it is actively enforced through both design and guidelines. At a design level, as discussed, all core data structures avoid runtime allocation: memory for components is pre-allocated in chunks, and growing those chunks (when needed) is done via unmanaged allocations outside of .NET's GC. Within a frame update, adding or removing components might mark some data or swap pointers, but doesn't allocate new objects on the managed heap. Iterating entities yields no garbage because of the custom iterators that are value-types[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45). The engine even avoids typical hidden allocations: for instance, using a C# foreach on a standard collection would allocate an enumerator object – FDP avoids this by providing its own for-like iteration tools (or by using Span<T>/pointers directly)[[24]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L26-L34). String manipulations in the hot path are avoided or done with fixed-

size buffers (e.g. the engine uses a custom FixedString32 struct for small strings like entity names to avoid allocating full String objects during gameplay[[25]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L95-L103)).

To help developers keep the hot loop allocation-free, FDP documentation explicitly lists forbidden practices in systems that run every frame. These include: creating new objects, using LINQ queries, using lambdas or delegates that capture variables (which allocate), boxing value types, and concatenating strings in-place[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45). Instead, it encourages using struct pools, static pre-allocated buffers, or recycling objects if needed. For example, if you need a large temporary list during a system update, you would use an object pool or a stack-allocated span rather than new List<>() each frame.

Are there any violations of zero-allocation on the hot path? By policy, core engine systems in FDP should not allocate in steady-state operation. After the initial setup, the expectation is that the GC will not be triggered during the regular 30–60 Hz simulation loop[[26]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L12-L19). In practice, certain things will cause allocations if used inappropriately – for instance, if a game developer continuously creates new entities or managed events every frame, those will allocate memory. FDP's stance is that such operations should either be limited to setup or handled via pooling. The engine's own features like the Flight Recorder run in a way designed not to interfere with the hot path: recording uses pre-allocated buffers and background threads for compression, so it doesn't allocate during the frame's critical section. One area to watch is **managed components** – if a system frequently updates a managed component or creates new managed component instances often, this can invoke the GC. The intent is that Tier 2 components are not churned every frame. Similarly, publishing a large number of managed events every frame would generate garbage (each event is a new object), so the design presumes that high-frequency events are kept unmanaged.

During development, any accidental allocations in the critical path can be detected by profiling (the engine's zero-GC goal means they likely profile for Gen0 collections during a run). The presence of the strict guidelines[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45) and the fact that FDP provides alternative patterns (like stackalloc, spans, and pooled structures) means a diligent team can indeed run the simulation with essentially **no GC interruptions**. If there is any violation, it would typically be regarded as a bug or a misuse of the API. For example, using a standard C# foreach on an IEnumerable in a hot loop would break the rule – but that would be visible and can be corrected to a zero-allocation pattern. The engine's philosophy is clearly to eliminate all avoidable allocations in update logic, and achieve determinism and performance thereby. In summary, **FDP's hot path, when used as intended, does not allocate**; maintaining this requires discipline (and the engine assists by offering the needed low-level tools). This approach contrasts with typical managed game logic, and even some ECS libraries, where small allocations can sneak in – FDP takes the stance that even small garbage is unacceptable in the core simulation loop[[26]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L12-L19).

Flight Data Recorder – How the Deterministic Replay Works

One of FDP's most advanced features is its **Flight Recorder**, which provides a deterministic replay of the entire simulation. Conceptually, the Flight Recorder operates by taking snapshots of the ECS state on a fixed interval (for example, every frame or every few frames) and recording any changes in a binary log. The goal is that given this log, one can replay the sequence of frames and end up with the exact same state transitions as the original run – effectively "record once, play back the game/simulation exactly."

Snapshotting via Memory Pages: FDP avoids the naive approach of serializing each entity one by one (which would be slow for large entity counts). Instead, it treats the internal memory chunks like pages of a database. When it's time to capture a snapshot, FDP will **copy the entire memory chunk** buffers directly for all unmanaged components[[27]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L50-L59). Because each chunk is a fixed-size block (e.g. 64KB) and mostly contiguous in memory, this copying can be done with highly optimized operations (memcpy, etc.), achieving extremely high throughput (tens of GB/s throughputs are possible for memory bandwidth)[[28]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L52-L60). For example, instead of iterating 10,000 entities and their components, the recorder might just copy 10 chunks of 64KB each, which can complete in a fraction of a millisecond on modern hardware[[28]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L52-L60).

Sanitization for Determinism and Compression: A challenge arises because not every slot in those chunks is actually in use – there may be "garbage" data in slots where an entity was destroyed or not yet created. Copying raw memory would include this noise, which not only could leak inconsistent data into the replay but also hurts compression (random garbage doesn't compress well). FDP solves this by performing a **sanitization pass** before copying a chunk[[29]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L63-L71). It consults the Entity index (which knows which entity IDs are alive) to get a **liveness map** of the chunk[[30]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L59-L67), and then uses SIMD instructions to rapidly zero-out any slots in the chunk that are not active[[31]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L65-L71). This means the snapshot of a chunk becomes deterministic (no leftover stale bytes from dead entities) and highly compressible (runs of zeros compress extremely well). In fact, a mostly empty chunk (say only 5 entities alive out of 1024 capacity) will turn into a memory page that is almost entirely zeros, which a compression algorithm like LZ4 can reduce to a tiny size (a 64KB page of zeros might compress to <1KB)[[32]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L69-L72). This approach addresses both **determinism** (the replay won't be thrown off by stray data) and **storage efficiency** (replay logs are smaller and faster to write).

Double-Buffered Asynchronous Recording: The Flight Recorder operates alongside the simulation without pausing it. FDP achieves this by double-buffering the output and aligning with the phase system. Typically, during the Post-Simulation (or a dedicated snapshot/export phase), the engine will take the snapshot of all chunks and events for that frame. It does so while other game logic is paused (ensuring no one is writing to the data mid-copy) – essentially a short "stop-the-world" in a safe phase[[33]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L9). However, because it's copying raw memory which is very fast, this can be done quickly. Once the data is copied out of the live memory into a snapshot buffer, the simulation for the next frame can proceed immediately. The snapshot buffer is then handed off to another thread (or deferred to after the frame) to compress and write to disk. Meanwhile, a second buffer is used for the next frame. This **async double-buffering** ensures the recorder's I/O or compression work never blocks the simulation loop[[33]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L9). In effect, the simulation thread spends a small, bounded time to clone the data, then the heavy lifting of compressing and saving that data happens off the main thread. By the time the next snapshot is needed, the engine either has another buffer ready or has swapped buffers (hence "double-buffer").

Recording Deltas and Keyframes: To keep the log size reasonable, FDP does not necessarily record the full state every single frame after the first. It uses a combination of keyframes (full snapshots) and delta frames. A **keyframe** (also called an I-frame in analogy to video) is a complete dump of all component data (after sanitization). A **delta frame** for subsequent ticks contains only the differences since the last keyframe – essentially which chunks or entities changed, plus any events or newly created/destroyed entities in that frame[[34]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L133-L140). FDP's internal change versioning helps identify those differences quickly. Additionally, the engine logs a **destruction log** of entities that were removed, because a replay needs to know not just state changes but also when an entity was deleted (and similarly creation is implicit in the data when a previously unused slot becomes used)[[35]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L137-L145). The combination of version-tracking and the destruction log allows the recorder to output a delta frame that says "Frame N: these 3 chunks had some changes (with their updated bytes possibly compressed) and these 2 entities were destroyed." On replay, the system will apply those changes to the last known state, remove those entities, and thereby reconstruct the new state for frame N.

The **binary format** of the Flight Recorder frames reflects this structure: each frame record might contain a header with the frame number (or tick count) and flags for delta or keyframe, plus a list of destroyed entity IDs[[36]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L44-L53). Then it will have a section for any events that occurred that frame (segregated by unmanaged events vs managed events as discussed)[[22]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L50-L59)[[23]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L60-L68). Finally, it has the component data – either the full chunk data (for a keyframe) or the modified chunk data for a delta[[37]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L68-L70). The log thus captures everything: state and events. During playback, the engine essentially reverses this process: it reads a frame, updates its internal state memory (applying any chunk data changes and zeroing where needed, creating or destroying entities to match), and injects the recorded events into the Event Bus for that frame (so that any game logic that would normally run on an event still runs in replay)[[38]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L86-L95)[[39]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L122-L128). Because the recorder also increments the same global tick counter and uses the same order of operations, the simulation proceeds just as it did originally.

Automatic Serialization for Managed Data: One challenge is how to record **Tier 2 (managed) components**, since those are reference types that can't be memcpy'd in the same way. FDP addresses this by using an **auto-serializer** system that uses expression trees to generate serialization code for any managed component or event type at runtime[[7]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L17-L20)[[40]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L10). Essentially, when the recorder encounters a class-type component that needs to be logged (for example, a inventory component), it will on-the-fly build an expression tree that accesses each field/property of that class and writes it to the binary stream (and similarly can generate a reader for replay). This is done once per type and cached. The result is that managed objects are serialized nearly as fast as hand-written code, without using slow reflection each time[[40]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L10)[[41]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L150-L158). The expression tree approach also ensures zero allocations during

serialization – it avoids boxing or creating intermediary objects by directly emitting the necessary store operations into a compiled delegate[[41]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L150-L158). In the Flight Recorder log, managed components' data will thus appear in the snapshot as a series of bytes produced by this custom serializer. On replay, the engine will allocate a new instance of the managed component class and use the generated deserializer to populate it. This way, even the complex objects in the world can be faithfully reconstructed.

Use Cases and Benefits: For a system architect, the Flight Recorder means you can achieve reproducibility and debugging capabilities usually found in specialized game replay systems or networking (rollback) systems, but here it's generalized for the entire ECS. If a bug occurs, the exact sequence of states leading to it can be replayed. For networking, deterministic snapshots can be compared or sent as deltas. And since it's built-in, developers don't have to manually instrument all their game logic to log state – the ECS data itself is captured. The careful design (zeroing dead data, compressing zeros, delta diffs) ensures that this is efficient enough to run continuously (recording every frame) if needed, which is remarkable given the amount of data modern simulations can produce. The combination of asynchronous processing and the high-speed memory copying means the performance impact is minimal (the docs indicate the entire serialization step can take well under 1 ms per frame for tens of thousands of entities[[42]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L94-L101)).

In summary, the Flight Recorder works by **treating the ECS memory as the source of truth and efficiently copying it**, rather than iterating object by object. It introduces smart steps (sanitization, compression, delta tracking) to reduce the data size and ensure determinism. By logging both state and events with proper ordering, it guarantees a replay will mirror the original run exactly. This system exemplifies FDP's overarching philosophy: leveraging low-level optimizations (memory layout, intrinsic operations) combined with higher-level architectural structure (phases, event streams) to achieve something that is both **high-performance and architecturally elegant**. The result is an ECS engine where not only is gameplay simulation extremely fast and scalable, but it can also be *rewound or replayed* for debugging, testing, or network synchronization with the same deterministic outcome – a feature that sets FDP apart from typical ECS implementations.

Summary

Fast Data Plane is an ECS engine built for engineers who care deeply about:

- Data layout
- Performance guarantees
- Deterministic behavior
- Debuggability
- Architectural clarity

It deliberately trades some convenience and abstraction for:

- Predictable execution
- Zero-GC hot paths
- Powerful introspection
- Long-term maintainability at scale

If your system needs to be **fast, correct, reproducible, and analyzable**, FDP provides a foundation designed precisely for that purpose.

Sources:

1. Fast Data Plane README – Overview and Key Features[[4]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L2-L5)[[43]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L12-L20)[[8]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L21-L22)
2. FDP Technical Specification – Core Architecture & Memory Model[[2]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L14-L17) [[10]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L33-L40)[[9]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L20-L28)
3. FDP Technical Specification – Iterator and Allocation Guidelines[[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45)
4. FDP Technical Specification – Event Bus and Managed/Unmanaged Events[[21]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2684-L2686)[[22]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L50-L59)[[23]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L60-L68)
5. Unity ECS Documentation – Managed vs Unmanaged component storage (comparison context)[[16]](https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Optimize%20managed%20components)
6. FDP Flight Recorder Design – Snapshotting and Replay Design Docs[[27]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L50-L59) [[29]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L63-L71)[[34]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L133-L140)[[40]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L10)

[[1]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L12-L14) [[4]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L2-L5) [[5]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L15-L16) [[7]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L17-L20) [[8]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L21-L22) [[18]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L6-L9) [[43]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md#L12-L20) README.md

https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/README.md

[[2]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L14-L17) [[3]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L2-L5) [[6]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L38-L45) [[9]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L20-L28) [[10]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L33-L40) [[11]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L35-L40) [[12]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L22-L29) [[13]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L68-L70) [[14]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L124-L133) [[15]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L124-L132) [[19]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L85-L93) [[20]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L68-L75) [[24]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L26-L34) [[25]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L95-L103) [[26]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md#L12-L19) Specification-all-in-one.md

https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Specification-all-in-one.md

[[16]](https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Optimize%20managed%20components) [[17]](https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Unlike%20unmanaged%20components%2C%20Unity%20doesn%27t,less%20optimal%20than%20unmanaged%20components) Managed components | Entities | 1.0.16

https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html

[[21]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2684-L2686) [[27]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L50-L59) [[28]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L52-L60) [[29]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L63-L71) [[30]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L59-L67) [[31]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L65-L71) [[32]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L69-L72) [[33]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L9) [[34]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L133-L140) [[35]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L137-L145) [[40]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L2-L10) [[41]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L150-L158) [[42]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md#L94-L101) FDP-FlightRecorder.md

https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/FDP-FlightRecorder.md

[[22]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L50-L59) [[23]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L60-L68) [[36]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L44-L53) [[37]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L68-L70) [[38]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L86-L95) [[39]](https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md#L122-L128) Event-Bus-Implementation-FINAL.md

https://github.com/pjanec/FastDataPlane/blob/724630871391e18f7148e0c2e56187caf0106602/docs/Event-Bus-Implementation-FINAL.md