

## ModuleHost Overview

The **ModuleHost** is an orchestration layer in the FastDataPlane (FDP) engine that allows you to run complex game features as decoupled **modules** alongside the core simulation. In a traditional ECS (Entity-Component-System) like FDP, all systems run on the main thread in lockstep with the frame rate. ModuleHost introduces a *hybrid architecture* where core simulation logic (physics, basic gameplay) remains on the main thread at 60 FPS, while additional features (AI decision making, analytics, networking, etc.) run in parallel as separate modules possibly at different frequencies <sup>1</sup>. This design lets you offload heavy or non-critical computations to background threads without freezing the main game loop. The result is a **sim-centric core** for high-frequency tasks and **module-centric features** for more asynchronous or lower-frequency tasks <sup>1</sup>. Modules are managed by the ModuleHost, which handles their scheduling, data access, and results integration into the main world state.

### What Is a Module and What Can It Be Used For?

A **module** represents a self-contained unit of game logic (AI behaviors, physics sub-systems, analytics, etc.) that can run outside the main simulation tick. Unlike a standard ECS system that runs *every* frame on the main world, a module can run at its own pace (for example, 10 times a second instead of 60) or even be triggered by specific in-game events. This makes modules ideal for tasks that are either too computationally expensive to do every frame or don't need to update that frequently. Common use cases include:

- **Artificial Intelligence (AI)** – e.g. strategic decision-making for NPCs, pathfinding algorithms, or behavior trees that can update at 10–20Hz instead of 60Hz. This allows complex AI logic to run in the background without slowing down rendering or physics.
- **Analytics & Game Stats** – e.g. tracking player performance, heatmaps of activity, logging events for telemetry. These can run at very low frequency (once per second or less) to aggregate data without impacting gameplay.
- **Background Simulation** – e.g. secondary physics or world simulation (crowd simulation, environmental simulations) that don't need to be in lockstep with core physics.
- **Network and Replay Recording** – e.g. a recorder module capturing game state every frame (or a network sync module sending state to clients). These might run at full frame rate but are isolated as modules for clarity and possible off-thread processing.

In essence, a module is like a *mini-system* with its own update loop, managed by ModuleHost. It operates on game state but doesn't execute on the main thread's critical path. This means, for example, your 60Hz physics and rendering aren't held back by a 10Hz AI planner – the AI runs concurrently and feeds results back when ready.

**Module Lifecycle and Frequency:** Each module defines how and when it runs via its **tier** and **update frequency** <sup>2</sup>. ModuleHost supports two broad tiers:

- **Fast Tier Modules:** High-priority modules that run *every frame* (synchronized with the simulation). These are used for time-critical features. For example, a networking module might run at 60Hz to

replicate state exactly each frame, or a “flight recorder” module might log every simulation step. Fast-tier modules prioritize low latency and use a strategy that keeps a continuously updated full copy of the world state (more on this below). By definition, fast modules have an update frequency of 1 (every tick) <sup>3</sup> <sup>4</sup> .

- **Slow Tier Modules:** Modules that run at a lower frequency, e.g. every  $N$  frames (for instance 10Hz means every 6th frame in a 60Hz game). These are for less urgent or heavier computations like AI or analytics. A slow module declares an `UpdateFrequency`  $> 1$  (e.g. 6 for 10Hz) and will skip frames between executions <sup>5</sup> <sup>6</sup> . Slow tier sacrifices immediacy for performance, and uses a more efficient snapshot strategy to avoid copying all data each frame.

Beyond purely time-based scheduling, ModuleHost also supports **event-driven triggers**. A module can be configured to “wake up” and run **ahead of schedule** if certain conditions occur, such as a specific component changing or an event being fired <sup>7</sup> . This reactive scheduling means modules don’t always have to poll for changes – they can respond instantly to important game events. For example, if an explosion event happens, an AI module could be signaled to run immediately (instead of waiting for its next 0.1s tick) so that AI characters can take cover without delay <sup>7</sup> . In practice, you specify which data a module cares about (certain component types or event types), and the ModuleHost monitors those. If a relevant change or event occurs, it interrupts the module’s timer and schedules it to run on the next frame <sup>7</sup> . This hybrid of *time-based* and *data-driven* scheduling ensures modules can be both performance-friendly **and** responsive when needed.

## How ModuleHost Schedules and Runs Modules

ModuleHost acts as a central **scheduler and dispatcher** for all modules. When you initialize ModuleHost, you register each module with it, optionally giving each module a custom snapshot strategy or else letting the host assign a default based on the module’s tier (fast vs slow) <sup>8</sup> <sup>9</sup> . Once configured, the host coordinates their execution every frame in the following general sequence (integrated into the game’s main loop):

**1. Main Simulation Step:** At the start of each frame, the normal ECS systems update the live world (e.g. handling input, running physics, updating game logic). ModuleHost ties into this via FDP’s standard *system phases*. In fact, a module *can* also register some synchronous systems to run on the main thread if needed (for example, if part of the module must update a component each frame). ModuleHost’s internal scheduler will ensure any such systems run in the correct phase and order, respecting dependencies like “UpdateBefore/After” relationships <sup>10</sup> <sup>11</sup> . However, by default most module logic is meant to run asynchronously, off the main thread, which happens after the core simulation step.

**2. Sync Point – Snapshot Creation:** Once the core simulation for the frame is done (e.g. after the physics and gameplay systems for frame  $N$  have run), ModuleHost enters a **synchronization point** in the frame <sup>12</sup> . At this moment, it prepares the data each module will need this frame: essentially taking a “snapshot” of the world state. Depending on each module’s strategy, this is done in one of two ways:

- **Global Double Buffer (GDB) Sync:** For fast modules, the host keeps a *persistent replica* of the entire world (often called “World B”). Each frame, it updates this replica with any changes that happened in the live world (World A) since last frame <sup>13</sup> <sup>14</sup> . This sync is highly optimized – using dirty chunk tracking, it only copies parts of memory that actually changed <sup>15</sup> <sup>16</sup> . The result is an up-to-date mirror of the live state without reallocating memory each time (essentially a continuously shadowed

world). This approach adds a small overhead each frame (e.g. <2ms for 100k entities with ~30% changes <sup>17</sup>) but ensures that whenever a fast module runs, it has a *complete*, ready-to-use world state available <sup>14</sup> <sup>18</sup>.

- **Snapshot-on-Demand (SoD):** For slow modules, instead of maintaining a full copy every frame (which would be wasteful if the module isn't running most frames), ModuleHost creates **on-demand snapshots** only when the module needs to run <sup>19</sup> <sup>20</sup>. At the sync point of a frame where a slow module is due, the host will allocate or reuse a smaller **snapshot** (World C) and copy just the relevant data that module cares about <sup>19</sup>. For example, an analytics module might only need a handful of components (Position, Health, etc.), so its snapshot will filter out all other component types <sup>19</sup>. This filtering drastically reduces data copying for slow modules (often copying only 50% or less of data) <sup>21</sup>. Snapshots are often pooled – reused across frames – to avoid constantly allocating memory <sup>22</sup> <sup>23</sup>. The snapshot-on-demand strategy means that if a module isn't scheduled to run on a given frame, no snapshot is created for it (saving work), and if it runs infrequently, it always sees a recent world state without having maintained a full mirror in the interim.

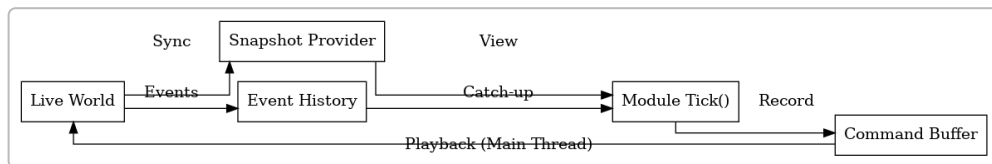
It's worth noting that ModuleHost can also optimize when *multiple slow modules* share the same update frequency. In such cases it can use a **Shared Snapshot Provider**, taking one snapshot and giving the same view to several modules that all need that data at the same time <sup>24</sup> (this is like a "convoy" of modules running together). This saves each from making redundant copies of similar data.

**3. Event Buffering:** During the sync point, ModuleHost also captures events from the game's **Event Bus** into an event history buffer <sup>25</sup> <sup>26</sup>. FDP's event bus accumulates events (such as "EnemyKilled" or "ExplosionOccurred") as they happen within the frame. For modules that are not running every frame, ModuleHost retains a history of events (by default ~3 seconds or 180 frames worth) <sup>27</sup> <sup>28</sup>. At each frame's sync point, it records the newest events and prunes older ones, ensuring a rolling window of relevant events is stored <sup>26</sup> <sup>29</sup>. These events will later be delivered to modules when they run, so a module can *catch up* on anything it missed since its last execution. For instance, if an AI module runs only every 6 frames, and a battle occurred in between, all those combat events (shots fired, damage taken, etc.) can be queued up for the AI module so it processes them on its next tick rather than missing them entirely <sup>30</sup>. This event accumulation guarantees **zero data loss** for modules – no matter how infrequent, they won't miss one-off events <sup>30</sup>.

**4. Module Execution (Async):** After updating snapshots and events, ModuleHost decides which modules need to run *this frame*. For each module, it uses the module's configured frequency or triggers to determine if it should execute now. Fast tier modules are scheduled every frame by definition <sup>31</sup>. Slow tier modules are scheduled when their frame counter reaches their `UpdateFrequency` (e.g. a module with `UpdateFrequency=6` will run on every 6th frame, with an internal counter tracking frames since last run) <sup>32</sup>. Additionally, if any event or watched component change has signaled a module, ModuleHost can force it to run on the current frame even if its frequency counter isn't due yet (this design was specified to react within <1ms to triggers) <sup>7</sup>. All modules that need to run will be dispatched **in parallel**. ModuleHost spawns a task/job for each module's `Tick` function, providing it with a **read-only simulation view** (the snapshot or replica) and the elapsed time since the module's last run <sup>33</sup> <sup>34</sup>. Importantly, the `deltaTime` given to a module is *not* just the frame time, but the total accumulated time since it last executed (e.g. a slow module that skips 5 frames might get ~0.083s if each frame was ~0.0167s) <sup>35</sup> <sup>36</sup>. ModuleHost automatically accumulates time per module to make this seamless. All module tasks run concurrently on background threads (from the .NET thread pool) so they do not block the main thread <sup>37</sup>. This effectively

utilizes multi-core systems: while the main thread moves on to rendering or the next frame's simulation, modules crunch data in parallel.

**5. Integrating Module Results:** Since modules operate on copies of the world, they cannot directly change the live game state during their background execution (doing so would be unsafe and could conflict with the main thread). Instead, modules follow a deferred write pattern using a **command buffer**. During its tick, a module can record intended changes (like "spawn an entity", "apply damage to enemy X" or "change component Y on entity Z") into a thread-local command buffer associated with its view <sup>38</sup>. This buffer collects all the module's outputs while the module is running, but doesn't apply them to the live world immediately (the live world is still busy with the next frame). Once all modules have finished running (ModuleHost waits for all their tasks to complete by the end of the frame), the host enters a **post-simulation phase** to **play back** all these buffered commands on the main thread <sup>39</sup> <sup>40</sup>. In other words, at a safe point, each module's effects are merged into the real world in a deterministic, single-threaded way. The sequence is: module reads snapshot → module enqueues commands → ModuleHost later applies those commands to the real `EntityRepository` <sup>41</sup> <sup>42</sup>. This design ensures thread-safety (no two threads ever write to the live world simultaneously) while still gaining parallelism on the read/compute part. It's implemented such that each module's command buffer is thread-local (no locks needed during record) and then ModuleHost simply iterates through all recorded commands at the end of the frame <sup>43</sup> <sup>44</sup>. For example, if an AI module decides to spawn new projectiles and update NPC velocities, it will call functions like "CreateEntity" or "SetComponent" on its command buffer (not directly on the snapshot). Those commands get applied after the module finishes, adding the new entities and updating components in the live world for the next frame <sup>43</sup> <sup>45</sup>.



*Figure 1: ModuleHost data flow.* **Live World** state is synced to a **Snapshot** (full copy for fast modules or filtered for slow modules). The module reads from this **View**, and any changes (entity spawns, component updates) are recorded to a **Command Buffer**. Later, on the main thread, the command buffer is played back into the Live World. Meanwhile, events from the Live World are buffered in an **Event History**, so when the module runs, it can consume any events that occurred since its last update <sup>46</sup>.

**6. Repeat Next Frame:** In subsequent frames, the same process repeats. Fast modules will run every time; slow modules will wait their turn. If a module did not run on a given frame, its snapshot from that frame (if any) is simply discarded or pooled, and it accumulates more elapsed time and event history until it runs next. The ModuleHost keeps track of how many frames since each module's last run (`FramesSinceLastRun`) to know when to schedule them <sup>33</sup> <sup>47</sup>. This continues as long as the simulation is running. ModuleHost can also be stopped or modules can be dynamically added/removed if needed (though typically modules are set up at initialization).

Throughout this loop, ModuleHost effectively plays the role of **mediator** between the ECS world and the modules: it provides *views* of the world to modules at the right time, and then reconciles their outputs back into the world in a controlled manner.

## Modules vs. Core Component Systems in FDP

It's important to understand how modules relate to the normal **component systems** of FDP. In FDP (FastDataPlane), the usual way to add game logic is to write systems that operate on entities/components and run in one of the fixed phases each frame (Input, Simulation, PostSimulation, etc.). These systems run on the main thread, directly on the live world data structure (the `EntityRepository`). By contrast, **modules** are a higher-level construct that **complement** this ECS model for certain use cases.

Key distinctions:

- **Synchronous vs Asynchronous:** Standard component systems execute synchronously every frame, whereas module logic (especially slow-tier modules) executes asynchronously and/or at a different rate. For example, a `PhysicsSystem` (regular component system) will update every frame without fail, but an AI module might run in parallel and only occasionally. Modules give you *temporal flexibility* – you don't have to squeeze every task into the per-frame budget if it can tolerate a delay.
- **Data Access:** Regular systems operate on the live world data directly (with write access), usually constrained by mutating only during certain phases to maintain stability. Modules, on the other hand, **never touch live data directly** – they go through the snapshot view which is read-only <sup>48</sup>. This is a conscious design to avoid the typical multi-threading hazards (race conditions, data tearing). It means modules are somewhat like *pure functions* for the game state: they take a snapshot as input and produce a set of commands as output. The live world is only changed in one place (the command buffer playback phase) which is easier to reason about and debug.
- **Integration with FDP Phases:** FDP component systems often use attributes like `[UpdateInPhase(Simulation)]` or dependency attributes `[UpdateAfter(X)]`. `ModuleHost` reuses this idea for any *synchronous part* of a module. A module can register an **IModuleSystem** – essentially a small system that also operates on the simulation view – if it needs to do something on the main thread each frame (for instance, a module might need a system in the Input phase to catch player input and store it for the module's background logic). These module systems get scheduled by `ModuleHost`'s internal scheduler alongside other systems, and can depend on or precede other systems as needed <sup>10</sup> <sup>11</sup>. In practice, most modules might not require any synchronous portion at all, but the option is there to integrate with the ECS loop. If `IsSynchronous` is set for a module (or in older terms, if a module provides systems via `RegisterSystems`), `ModuleHost` will treat it partly like a normal system on the main thread <sup>49</sup> <sup>50</sup>. Still, the *heart* of a module is its asynchronous `Tick` which runs off-thread with the snapshot.
- **Use Cases Fit:** Deciding whether to implement a feature as a module versus a standard FDP system boils down to how intensive and decoupled the task is. If the logic must run every frame and directly affects core gameplay (e.g. player movement or collision resolution), it's usually best as a regular ECS system (for immediate consistency). But if the logic can be done slightly later or at its own pace (like an AI thinking about its next move, which could feasibly happen a few frames later without issue), then a module is beneficial. Modules shine when a feature can tolerate *eventual consistency* (a slight delay in applying results) in exchange for not bogging down the main thread. They also help structure the code: complex subsystems can be encapsulated as modules with their own internal logic, rather than tangled in the frame-by-frame loop. In summary, **use modules for tasks that are heavy or optional enough to run in parallel**, and keep trivial or absolutely time-critical logic in regular component systems. Many games end up using a mix: core systems for tight loops, and `ModuleHost` modules for large-scale or background tasks. `ModuleHost` is essentially an extension of the ECS paradigm to better utilize multi-core processors and to organize code by feature.

## Snapshots and Simulation Views (State Access for Modules)

A fundamental concept enabling modules is the **Simulation View** – a consistent, read-only snapshot of the game state that a module operates on. FDP's `EntityRepository` (the world) is optimized for in-place updates and high-speed iteration, but it's not safe to share directly across threads. `ModuleHost` solves this by providing *views* onto the world, implemented in two forms as described earlier: a **replica** (for fast modules) or a **snapshot** (for slow modules). Both forms implement a common interface that lets modules query entities and components in a uniform way <sup>51</sup> <sup>52</sup>. From a module's perspective, it doesn't actually know or care if it's reading from a continuously synced replica or a one-time snapshot – it just sees an `ISimulationView` that behaves like an ECS world where you can look up entities, components, and even query sets of entities by component type.

**View Types:** The *Global Double Buffer view* is essentially an `EntityRepository` copy that is kept in sync every frame. The *On-Demand Snapshot view* might be a lighter object (possibly internally also an `EntityRepository` but one that's partially filled with only certain components). In code and design, both are often represented as an `EntityRepository` under the hood, to reuse all the query and component access functionality of FDP's world structure <sup>53</sup> <sup>54</sup>. The key difference lies in how they are produced and maintained:

- A **GDB (double-buffer) view** is *persistent*. When a fast module runs, `ModuleHost` calls something akin to `replica.SyncFrom(liveWorld)` to bring the replica up to date <sup>55</sup> <sup>56</sup>. This is done every frame regardless of whether the module actually runs (because fast modules run every frame by definition). The upside is that acquiring the view is extremely cheap – the module can be handed a pointer to this already-prepared replica immediately, with no allocation and minimal copy work (only incremental changes were memcp'd during sync) <sup>16</sup> <sup>57</sup>. The view is also *not* disposed or destroyed after each use – it persists across frames, always holding the last frame's state. That's why for GDB the `ReleaseView` operation is a no-op <sup>58</sup>; there's nothing to free, the replica just stays alive. This strategy is chosen for modules that need a lot of data at high frequency because it prioritizes speed and consistency over memory – essentially you pay a constant cost each frame to keep the copy ready <sup>48</sup> <sup>59</sup>.
- An **SoD (on-demand) snapshot view** is *transient*. When a slow module is about to run, `ModuleHost` will take a fresh snapshot from the live world. Under the hood this might involve taking a pre-allocated `EntityRepository` from a pool and copying only the necessary components into it <sup>60</sup> <sup>61</sup>. The module is given this snapshot as its view and, after the module finishes, the snapshot can be recycled back into a pool for future use <sup>62</sup> <sup>63</sup>. Because snapshots are created only when needed, a slow module that runs, say, once per second does very little work in the 59 frames between runs (just gathering events). When it does run, there is a short spike to copy relevant data. However, because you can filter by component type, this cost is controlled by the module's actual needs. For example, an analytics module might request only Position, Health, and a few other components – so the snapshot copy ignores all other data like graphics components or physics colliders. This targeted copying makes SoD efficient for sparse data <sup>48</sup> <sup>64</sup>. The snapshot is disposed or returned to a pool after use.

Modules can influence what data ends up in their snapshot by declaring **requirements** (depending on how the `ModuleHost` API is used). In the current design, modules can expose which component types and event types they need to see. The `ModuleHost` can use this to build a bitmask of components for that module's snapshot <sup>65</sup> <sup>66</sup>. If unspecified, the host might default to giving slow modules *all components* (as seen in

code where a mask is set to all bits 1 by default) <sup>67</sup>, but for efficiency it's expected you'd specify only what's needed, especially if your world has many component types. For fast modules using GDB, typically the full world is copied (since they're likely to need broad access), so filtering is often not applied in that case <sup>68</sup>.

No matter the strategy, the view that a module gets provides a **consistent snapshot of the world at a specific frame**. This means even if the main world is changing (the next frame might be simulating), the module's view is static – it sees the world state as of the sync point when the snapshot was taken. The view interface only allows read-only operations: e.g. `GetComponentRO<T>(entity)` to get a value type component, `GetManagedComponentRO<T>` for reference-type components, checks like `IsAlive(entity)` to see if an entity exists, and the ability to build entity queries (to iterate over all entities with certain components) <sup>69</sup> <sup>52</sup>. The module cannot (and should not) attempt to directly alter any component through this interface. This immutability is crucial for thread safety – since multiple modules and the main thread might all be looking at various versions of the world, only the main thread's live world is ever actually modified (via the command buffer applications). By designing the view as read-only, ModuleHost ensures a module can freely read without locks and never worry that the data it's looking at will be half-way changed by another thread.

From a developer's perspective, using the view in a module feels similar to using the `EntityRepository` API in a normal system, except you're limited to reads. For example, you might do: "for each enemy entity, get its position and health from the view, then decide an action". You can use the query builder to gather entities of interest efficiently, and use the provided accessor methods to fetch components. The module doesn't know if it's a replica or snapshot – that's transparent. This means you can write module logic in a generic way and let ModuleHost figure out the optimal data provisioning (GDB or SoD) under the hood <sup>70</sup> <sup>71</sup>. As the ModuleHost documentation emphasizes, *modules are agnostic to the strategy – they just use the unified view interface* <sup>70</sup> <sup>72</sup>. This simplifies module code and allows the system to evolve (for instance, switching a module from slow to fast tier or vice versa is mostly a configuration change, not a rewrite of the logic).

## Event Handling in Modules

Events in FDP are typically delivered via the Event Bus to systems in the same frame they occur. However, modules that run less frequently would naturally miss events that happen in between their ticks. ModuleHost's solution, as touched on above, is to accumulate events and attach them to the simulation view given to modules <sup>30</sup>. When a module obtains its snapshot view for a tick, that view comes with an **event history** – basically a backlog of events since the module last ran. The module can then *consume* these events and process them as needed. The view interface provides a method like `ConsumeEvents<T>()` for this purpose, returning all events of type T that occurred in the interim <sup>73</sup> <sup>74</sup>. Modules typically call this at the start of their tick to handle any relevant happenings. For example, an AI module might consume `ExplosionEvent` and `PlayerDeathEvent` streams so it can immediately respond (maybe adjust strategy if a big explosion happened nearby, or choose a new target if a player died). Once consumed, those events are cleared from that module's perspective (each module keeps its own "last seen" tick to know what events are new) <sup>26</sup> <sup>29</sup>.

The event buffering is designed to be lossless within a sliding window. By default, ModuleHost was configured to keep ~3 seconds of events (e.g. 180 frames at 60Hz) in memory <sup>28</sup>. If a module hasn't run for longer than that, older events might be dropped (to avoid unbounded memory growth), but under normal operation modules will run often enough to consume events before they expire. Developers can typically

configure how many frames of event history to retain (for instance, a module's `ModuleDefinition` can specify `MaxEventHistoryFrames`) to suit the needs of the game <sup>75</sup>.

It's also possible for modules themselves to raise events. If a module's logic generates some notable game event (say, the AI module detects "EnemySighted"), it can enqueue that as a game event via the command buffer or a special call. The event would then be dispatched on the next frame's event bus like any other event. There isn't a built-in automatic propagation of module-raised events back into other modules except through the normal event system (i.e. one module could output an event that another module's next tick would consume via the buffer). Typically though, modules communicate with the game mainly by state changes (components) or by producing events for the main simulation to handle.

**Event-Driven Module Activation:** A significant advanced feature is that modules can declare certain events (or component changes) that should trigger them to run immediately. Under the hood, `ModuleHost` can inspect incoming events each frame and if a module subscribed to that event type, the host will mark that module as ready to run on that frame. For example, our AI module could register interest in a "PlayerSpottedEvent"; normally it ticks every 0.1s, but if a `PlayerSpottedEvent` comes in, it could be run on the same frame as the event to update AI behavior promptly. Similarly, a module can watch for changes in specific components (say a "Health" component change) – perhaps an analytics module wants to run whenever any entity's Health drops to 0 to immediately record a kill. This capability essentially treats events as interrupts <sup>7</sup>. It was documented that the scheduler's check for triggers is extremely fast (on the order of 100ns per module to check, versus the cost of waking a thread ~50 microseconds) <sup>76</sup>, so it's efficient to do this every frame. In practice, to use this you'd configure the module's definition with lists of event types or component types to watch (e.g. `WatchEvents = [ typeof(KillEvent) ]` for an analytics module) <sup>77</sup> <sup>78</sup>. The `ModuleHost` then knows to inspect the event buffer for any new `KillEvent`; if found, it bypasses the usual frequency delay and runs the analytics module immediately next frame. This ensures that even infrequent modules can have *real-time reactions* to critical events. The design guarantee was to trigger within <1 frame of the event <sup>27</sup>, which for game design means near-instant responses.

To illustrate, consider the **AnalyticsModule** from a battle royale example: it runs at 1 Hz to aggregate stats, but it could watch for a "MatchEndEvent". If the match ends (event fired), you'd want the analytics module to run one last time immediately to report final stats rather than waiting up to a second. With event-driven scheduling, the `MatchEndEvent` can poke the analytics module to execute right away on that frame. This combination of periodic and reactive scheduling makes modules very flexible.

In summary, modules integrate with the event system in two ways: they reliably *receive* past events when they run, and they can be *triggered* by events to run sooner than scheduled. Together, these ensure modules are both up-to-date and responsive to the game's happenings.

## Debugging and Diagnostic Capabilities

Working with modules introduces new dynamics (multi-threaded execution, decoupled timing), so `ModuleHost` provides several diagnostic tools to aid development and debugging:

- **Per-Module Profiling:** The `ModuleHost` keeps track of performance metrics for module systems and module ticks. Each module system (the synchronous bits) and each module tick can be timed and recorded. Internally, the scheduler profiles how long each system or module's `Execute` took and

can record statistics like total time, average, max execution time, etc. <sup>79</sup> <sup>80</sup> . This data can be exposed to the developer (e.g. via a profiling UI or logs) to identify any modules that are taking too long. Because modules run in parallel, one misbehaving module could starve others or delay the command buffer merge if it's still running when others finished. By monitoring execution time, you can tune a module's `MaxExpectedRuntimeMs` (a config field) and set up alerts or mitigations if it exceeds that <sup>28</sup> <sup>81</sup> . In a robust setup, you might even have **watchdog** timers (as hinted by the architecture's Resilience layer) that can cancel or flag modules that hang or consistently overrun their budget.

- **Simulation Views Inspection:** Since each module works on a snapshot, it can be useful to inspect those snapshots during debugging. `ModuleHost` allows you to query how many snapshots (shadow buffers) are active and how much memory they are using <sup>82</sup> <sup>83</sup> . For instance, `GetActiveShadowBufferCount()` and `GetTotalShadowBufferBytes()` are provided to see if your modules are consuming a lot of memory with snapshots <sup>82</sup> . This is helpful for verifying that on-demand snapshots are being reused properly (the count should ideally not keep growing) and that memory usage is within expected bounds. You can use these diagnostics to catch leaks (e.g. if a snapshot was not released back to the pool) or to tune component filtering (if your snapshot is unnecessarily large, the memory totals will show it).
- **Event Log Monitoring:** `ModuleHost`'s event accumulator could also be instrumented to see how many events are being buffered for modules. For debugging, you might log when a module runs how many events it consumed (the example analytics module prints how many kill events it processed in the last second) <sup>84</sup> <sup>85</sup> . This helps confirm that the module isn't missing events and that the event system is working as intended. If you expected an event to trigger a module and it didn't, you can check whether the event was buffered or whether the module's watch list is correctly set.
- **Diagnostic Hooks in Modules:** The module interface itself can include debug hooks. For example, modules can implement a method like `DrawDiagnostics()` <sup>86</sup> . This can be used to output module-specific debug info – for instance, an AI module could draw the AI's planned path or current target on the screen for developers, or output internal state to the console. Because modules run off-thread, you wouldn't directly call debug drawing from within `Tick` (which is background), but `ModuleHost` could invoke `DrawDiagnostics()` on the main thread context (perhaps after commands are played back, or during a debug frame step) to allow the module to present anything useful. This method can be hooked up to a game's debug rendering routines. It's an intentional separation: `Tick()` is for logic and runs asynchronously, `DrawDiagnostics()` can be called synchronously to visualize that logic's results without race conditions <sup>86</sup> .
- **Logging and Exceptions:** Modules can use normal logging (e.g. writing to console or a file) just like any other code. In the example, the AI module prints a summary every 60 ticks <sup>87</sup> , and the analytics module logs counts each second <sup>85</sup> . These logs help verify that modules are running at the expected frequency and producing expected outcomes. If a module throws an exception during its `Tick` (due to a bug), `ModuleHost` will catch it when waiting for the task. Typically, the host will aggregate exceptions from module tasks and could either propagate them (potentially crashing the game if unhandled) or log them with the module's name. It's good practice to have error handling around `ModuleHost` updates so that one module's failure doesn't take down the whole simulation – e.g. you might catch exceptions around the `Task.WaitAll` and report which module failed. In a test environment, having 200+ tests covering modules (as mentioned in documentation) helps ensure modules behave, but at runtime, robust logging of exceptions is the safety net <sup>88</sup> .
- **Resilience Features:** The architecture documentation references a *Resilience & Safety* layer (watchdogs, breakers). This implies features like timing out a module that exceeds its worst-case

runtime or detecting a module that consistently errors out and perhaps disabling it. For example, if `MaxExpectedRuntimeMs` for a module is 50ms but it's taking 200ms, a watchdog could log a warning or even attempt to restart that module's logic if possible. Similarly, a circuit breaker could stop triggering a module that has hit repeated exceptions to prevent spamming failures. These are more advanced and game-specific, but `ModuleHost` is designed to allow plugging in such monitors.

All these diagnostic capabilities mean that while modules introduce concurrency, you have visibility into their operation. You can profile their performance impact, verify they receive the data they need, and ensure they behave correctly. In practice, when developing with `ModuleHost`, you'd run the game in a debug mode where you can see module timings each frame (similar to how you profile normal systems). If something is off (say an AI module not running when expected or running too slowly), the `ModuleHost`'s data (execution counts, last run tick, event triggers) is there to troubleshoot.

## Using Modules for Common Game Features

To ground these concepts, let's consider how one would implement typical game features using `ModuleHost` modules:

- **AI Behavior Module:** Imagine an NPC AI module in a strategy game. You could mark it as a slow module with `UpdateFrequency = 6` (running at 10Hz). In its tick, it would use the simulation view to gather necessary info: for example, query all AI-controlled entities and relevant targets (players or points of interest) <sup>89</sup> <sup>90</sup>. It would then run decision logic – e.g. choose a new waypoint for each AI, or decide to attack if an enemy is in range. Using the command buffer, it enqueues the results of those decisions: maybe setting a new `Velocity` component for each AI to move them toward a target, or spawning projectile entities to represent shots fired <sup>91</sup> <sup>92</sup>. When the module's commands play back, the main world gets updated with these AI actions (so on the next frame, those projectiles actually exist and AIs have new velocity). The AI module might also consume events like “heard a noise” or “took damage” to influence its behavior on its next run. Because it runs at 10Hz, it lightens CPU load compared to doing all AI logic at 60Hz, yet by using event triggers (e.g. on a loud noise event, run immediately) it remains responsive. This is a classic use-case: complex AI without frame-by-frame overhead, enabled by snapshots and event buffering. The provided **AIModule** example follows exactly this pattern – at 10Hz it finds nearest players for bots and issues movement and shooting commands <sup>93</sup> <sup>45</sup>.
- **Pathfinding Module:** Pathfinding (like A\* searches for multiple NPCs) can be very CPU-intensive. With `ModuleHost`, you can create a `PathfindingModule` that runs continuously but asynchronously. Perhaps it's a fast-tier module if you want it updating paths in real-time, or slow-tier if slight delays are okay. The module's view might include the navigation mesh or relevant component data (positions, obstacles). Each tick, it could take one or two pathfinding requests off a queue (populated by gameplay when a unit needs a new path) and compute those paths without blocking the main game. It would then output the results by setting a component on the unit with the new path or issuing move orders via events. This way, pathfinding work is distributed over many frames and threads. The main ECS might just handle the final movement following the path. The module approach prevents pathfinding from ever spiking the frame rate.
- **Analytics/Stats Module:** A module can periodically aggregate game statistics, which is useful both for gameplay (e.g. dynamic difficulty adjustment) and for telemetry. For instance, an `AnalyticsModule` might run once per second (frequency 60) and count various entities or events. The example analytics module counts players, bots, items, and projectiles, and processes all

`KillEvent` occurrences in the last second to update a “heatmap” of kill locations <sup>94</sup> <sup>95</sup> . It then prints out a summary to the console <sup>85</sup> . In a real game, instead of printing, it could send this data to a UI overlay or to a file. Running this as a module ensures that the potentially expensive counting of hundreds or thousands of entities is done infrequently and off the main thread, avoiding any impact on frame rate.

- **Replay Recorder or State Archive:** For making replays, you might have a module that runs every frame (fast tier) but on a background thread. Each tick, it could copy the essential state (or even use the GDB replica directly) and write it to a buffer or disk. Because the GDB provides a full world copy, the recorder module can literally serialize that snapshot without touching the live world (ensuring it's capturing a consistent state). Doing this asynchronously means the main game doesn't pause to write to disk. Another module could be a network sync, which at 60Hz diffs the live state (via the replica) and sends updates to remote clients. By isolating it as a module, the networking code stays separate and can be swapped or disabled easily (ModuleHost just wouldn't run that module if in offline mode).
- **Game Mechanics in Modules:** Some gameplay features can even be implemented as modules if they don't require immediate feedback. For example, a complex **weather simulation** or **economy simulation** in a strategy game could run as a slow background module updating climate or market conditions occasionally and pushing changes to the world (like changing weather effects or prices). A **background music manager** could run as a module reacting to events (like intensifying music on boss encounter events). Even certain **UI systems** (like a map that needs heavy processing to update) could be modules, as hinted by “UI modules” – though usually UI runs on the main thread, heavy computations for UI (say layout or analytics for UI) might be offloaded.

Overall, modules are well-suited to anything that can be done *eventually* and in parallel. By using them, you maintain the main thread's performance for what truly needs to be immediate (rendering, physics, tight input loops), and you utilize the full CPU for everything else in a structured way.

## When to Use ModuleHost (Modules) vs Plain FDP

While ModuleHost is powerful, it isn't a silver bullet for all game logic. There are scenarios where using modules is ideal and others where it's unnecessary or even undesirable:

### Use ModuleHost modules when:

- You have expensive computations that would cause frame stutters if done in the main loop. These could be made asynchronous to keep frame times smooth.
- A task doesn't need to run every frame. If doing it less often is acceptable, a module with a lower frequency is a cleaner solution than a main-thread system that internally skips work (why burden the main loop at all if it can be on a timer?).
- The work can be isolated to read-only game state and produce results later. Modules work best when they don't need continuous two-way interaction with the live simulation during their processing. If the problem can be solved with “take a snapshot, think, then output commands,” it fits the module model well.
- You want to encapsulate a subsystem. ModuleHost provides a clear API boundary – the module gets a view and must output commands/events. This enforces a good separation of concerns. If you find a piece of your game logic can be treated as a black box (inputs: current world, outputs: changes), that's a hint it could be a module. This often applies to things like AI, which takes the world state and outputs actions, or to subsystems like an inventory simulation that might be decoupled from frame updates.
- You need concurrency and scalability. On modern hardware, parallelizing tasks is key to using all CPU

cores. Instead of writing your own threads and worrying about locks, ModuleHost gives a framework that automatically parallelizes modules (via tasks) and handles syncing data. If you foresee needing to scale up logic (say having hundreds of AI agents each planning), splitting them into modules (or multiple module instances) can leverage multiple cores.

- The game is structured for potential distributed or networked environments. ModuleHost's design aligns with concepts like authoritative simulation and even distributing load (for example, a module could potentially run on a separate process or machine if the snapshot can be exported – though that's beyond current scope, it's conceptually possible). Even within one process, keeping things modular can ease turning features on/off (e.g. enable the Analytics module only in development builds).

#### **Stick with plain FDP (core systems) when:**

- The logic absolutely must run every frame and/or must run in a specific order relative to other gameplay systems. For example, anything to do with player input response (like movement, shooting) typically goes into the main update so the game feels responsive. If a player presses fire and your firing system is a module that runs 2 frames later, that latency might be noticeable and unwelcome.

- The computation is lightweight. Not everything benefits from being a module – if something takes 0.1ms on the main thread, it might be overkill to make it a module (the overhead of snapshotting and merging might outweigh any gain). For trivial or very fast systems, keep them simple. ModuleHost introduces some overhead (maintaining snapshots, task dispatch, etc.), which is minimal but not zero. Use it where there's a clear payoff (like moving a 5ms task off the frame).

- The system needs continuous real-time interaction with the simulation. For instance, a character controller reading player input and moving an avatar has to read input and write position within the same frame; splitting that into a module would complicate things because the module would act on an older snapshot. In general, if the logic can't tolerate acting on slightly out-of-date data, it probably belongs in the immediate ECS loop. Physics is a good example – it's usually kept in the main simulation step for deterministic, instant integration. (That said, secondary physics effects or predictions could be modular, but core collision resolution is not something you'd offload.)

- Debugging complexity outweighs benefits. While ModuleHost provides tools, debugging multi-threaded, delayed-effect systems is harder than debugging linear frame-by-frame code. If a gameplay feature is simpler to implement synchronously and it meets performance budgets, there's no need to complicate it by making it a module. For instance, a simple damage-over-time system that just decreases health each frame is fine as a normal system; turning it into a module that runs every frame would gain nothing and just add indirection.

In short, use modules judiciously: for the big stuff that *needs* them. Keep using FDP's core ECS for the straightforward or truly time-critical parts of gameplay. A rule of thumb might be: start with regular systems (keep it simple), and if you identify a system that's consuming too much time or could naturally run less often, then refactor it into a ModuleHost module. The design of ModuleHost is such that FDP itself remains intact – ModuleHost sits on top as an optional layer. You can perfectly well have a game that doesn't use ModuleHost at all (just FDP systems), and it will run normally. ModuleHost becomes beneficial as your game grows in complexity and you want that extra structure and performance.

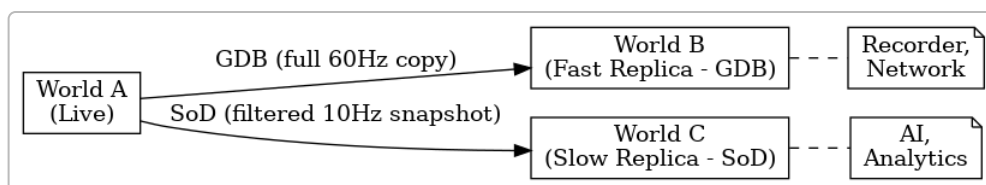


Figure 2: “Three-World” architecture with ModuleHost. The **Live World (A)** is the authoritative game state updated by normal ECS systems. ModuleHost maintains additional world states: **World B** is a fast replica (using Global Double Buffering) that is updated every frame from A and used by high-frequency modules (e.g. Network sync, Recorder). **World C** is a slow replica or on-demand snapshot used by low-frequency modules (e.g. AI, Analytics) <sup>96</sup> <sup>97</sup>. Fast modules always read from the constantly synced World B, while slow modules get snapshots of World C when needed. This topology illustrates how ModuleHost forks the simulation state to feed background tasks without disturbing the live world. (Dashed lines indicate which modules use which world.)

## How to Use Modules in Practice (Developer Guide)

Using ModuleHost in your game project involves a few steps, but it’s designed to be straightforward for anyone familiar with ECS patterns. Here’s a quick guide:

**1. Define Your Module Class:** Create a class that implements the module interface (often `IModule`). At minimum, you’ll give your module a name and specify its tier and update frequency. For example:

```
public class AIModule : IModule {
    public string Name => "AI";
    public ModuleTier Tier => ModuleTier.Slow;
    public int UpdateFrequency => 6; // e.g. run every 6 frames (10Hz)
    ...
    public void Tick(ISimulationView view, float deltaTime) { ... }
}
```

If using an updated API, you might instead return a `ModuleDefinition` object with these settings <sup>98</sup> <sup>81</sup>, but the concept is the same – configure how the host should schedule this module. In the `Tick` method, you will write the core logic that should execute when the module runs. This logic should use the `view` (snapshot) to read any game state needed and use some form of `command buffer` to record changes. For example, in pseudo-code inside `Tick`:

```
public void Tick(ISimulationView view, float deltaTime) {
    // Example: find all enemies and apply some effect
    foreach (var e in view.Query().With<EnemyTag>().Build()) {
        var health = view.GetComponentRO<Health>(e);
        if (health.Value < 20) {
            // Low health: enqueue an AI action or buff
            commandBuffer.SetComponent(e, new DefenseMode { Enabled = true });
        }
    }
}
```

If the API provides `view.GetCommandBuffer()`, you’d call that to obtain a buffer to record your `SetComponent` or entity creation commands <sup>43</sup>. The module should **not** directly modify

`health.Value` or other components on the view – those calls should only read data. All writes go through commands. This discipline ensures thread safety.

Optionally, if your module has any part that must run on the main thread each frame, you can implement the synchronous registration. For instance, your module class might implement a method `RegisterSystems(ISystemRegistry registry)` where you call `registry.RegisterSystem(new SomeSystem())` to add a system. That system could be a small class implementing `IModuleSystem` (which has an `Execute(ISimulationView, float dt)` method) and marked with an `[UpdateInPhase(...)]` attribute to place it in the frame. Most modules won't need this, but it's there for things like hooking into the Input phase or doing debug drawing.

**2. Declare Snapshot Requirements (Optional):** If your module doesn't need the entire world state, you should inform `ModuleHost` what subset of data to snapshot. Depending on the version of the API, this can be done by overriding methods or fields. For example, your module might override `GetSnapshotRequirements()` to return a mask or list of component types, and `GetEventRequirements()` for event types it cares about <sup>65</sup> <sup>99</sup>. In some implementations, you might specify this in a `ModuleDefinition` or attributes. By doing this, you optimize the snapshot process – `ModuleHost` will only copy those components and will filter the event history to just those event types for this module. If you skip this, `ModuleHost` might default to a safe choice (all components, all events), which works but could be less efficient. So it's good practice to be explicit. For example, an AI module might require `Position`, `Health`, `AIState` components and maybe wants `ExplosionEvent` and `DamageEvent` from the event bus – you'd list those so the snapshot includes them.

Similarly, if you want event-driven triggers, you'd list the `WatchComponents` or `WatchEvents` in the module's definition. E.g. *"Run immediately on ExplosionEvent"*. This way, when you register the module, `ModuleHost` knows to set up those triggers.

**3. Initialize ModuleHost:** Elsewhere in your game setup (perhaps in a `GameScene` initializer or the main Program), you need to create a `ModuleHostKernel` (or similar entry class) and register modules with it. For example:

```
var moduleHost = new ModuleHostKernel(liveWorld, eventBus);
moduleHost.RegisterModule(new AIModule());
moduleHost.RegisterModule(new AnalyticsModule());
// ... register more modules as needed
moduleHost.Initialize();
```

You pass the `ModuleHost` your live `EntityRepository` (so it knows the source of truth) and the event bus (so it can capture events). When registering each module, you can also specify a custom snapshot provider if you want (e.g. if you have a special one), but usually you let it pick the default based on tier <sup>8</sup> <sup>9</sup>. The host will internally create the necessary snapshot buffers or replicas now or at first use. Calling `Initialize()` on the host finalizes the setup: it will ask each module to register any synchronous systems (calling their `RegisterSystems`) and build the scheduling graph for those systems <sup>100</sup>. It also validates there are no dependency cycles among systems. Essentially, after `Initialize()`, the `ModuleHost` is ready to start running modules.

**4. Integrate into Game Loop:** Each frame of your game, you need to call the ModuleHost to do its work. Typically, you'd insert `moduleHost.Update(deltaTime)` (or a similarly named method) at the end of your frame update, after the regular ECS systems have updated the world. This `Update()` call corresponds to the steps described earlier: it will run any module systems in the "BeforeSync" phase, capture events, sync snapshot providers, dispatch module tasks, wait for their completion, and then play back their command buffers, and finally run any PostSimulation systems <sup>101</sup> <sup>102</sup>. All of that is abstracted in the ModuleHost's loop. From your perspective, it's just one call per frame. The `deltaTime` you pass is the frame time – the host uses it for timing any systems and for reference, but recall it accumulates actual module timing separately. If you have multiple tick rates (like physics at 60Hz, game at 30Hz), you'd call ModuleHost update whenever the simulation step happens to coincide with a frame. Most straightforward scenario: you call it every rendered frame with the simulation timestep.

**5. Run and Test:** With this integration, when you run the game, your modules should start executing at their defined intervals. Monitor the logs or console to see their debug outputs (like the example modules printing stats). You should see that their effects take place in game: e.g. AI characters start moving or shooting as determined by the AI module (with slight nuance that their actions appear with the aforementioned one-frame latency due to command buffering). Test that their frequency feels right and that event-triggered behavior happens immediately when expected. Use the diagnostics to ensure performance is as intended (e.g. check that the AI module actually ran 10 times a second, not slower due to overload, and that no frames had surprising delays).

**6. Debugging Tips:** If something seems off, enable additional logging in ModuleHost. For instance, you might instrument the ModuleHost to log when each module starts and ends a tick, or how many events were in its queue. This can reveal if a module is starved or running too often. You can also attach a debugger and put breakpoints in your module's Tick logic – since it runs on another thread, you might need to allow cross-thread breakpoints, but you can step through the logic as it processes the snapshot. Because the snapshot is a copy, you can safely poke around its data without affecting the live game. One thing to be careful of is thread safety with external resources: if your module interacts with file I/O or external systems, ensure those are handled properly (just as you would with any multi-threaded code).

**7. Evolving Modules:** As your game develops, you might adjust module parameters. For example, if the AI is too slow to react, you could raise its frequency or add an event trigger for urgent situations. Or if performance is an issue, you might lower a module's frequency or refine its component mask to shrink snapshots. The nice part is these changes are usually one-liners in the module config rather than sweeping code changes.

In practice, adding a new module to the game is relatively low friction: create the class, register it, and you've got a new concurrent system. ModuleHost takes care of the boilerplate of copying data and merging results. This encourages experimenting with offloading tasks – you might prototype a feature as a module to see if it helps performance and only keep it if it proves worthwhile.

**Example:** To solidify, let's walk through adding a hypothetical **"Loot Drop Module"**. Suppose whenever enemies die, we want to run some complex random generation algorithm to decide what loot drops, which could involve querying a lot of data or performing heavy computations (like simulating rarity rolls). We don't want to do this on the main thread at the moment of death because it could cause a hitch. So we make a `LootModule : IModule`. We set it to slow tier, maybe update frequency = 1 (meaning it *could* run every frame, but we rely mainly on triggers). In `WatchEvents`, we put `DeathEvent` so it triggers on any death.

The `Tick` of this module, when invoked, will consume all `DeathEvents` from its event buffer (there could be multiple if many enemies died in one frame) <sup>84</sup>. For each death event, it will look at the event details (maybe which enemy died, their level, etc., available through the event data) and perform the loot roll logic. It might then use the command buffer to create new entities for the loot items and spawn them into the world (with appropriate components like `Item`, `Position` at the death location). After the module finishes, those commands go into effect and the loot appears in the game world. The player experiences a slight delay – the enemy dies (frame *X*), and perhaps by frame *X*+1 or *X*+2 the loot pops out – but this is usually imperceptible and well worth the performance gain of not doing it immediately on the death handling system. This module can be tested independently (trigger some fake `DeathEvents` and see if it drops loot correctly) and adjusted (if loot drops feel delayed, we ensure the event trigger is working so it runs next frame after death).

Finally, if you ever need to run the game without modules (for example, comparing performance or running in a mode where all logic is single-threaded for determinism), you could disable `ModuleHost` or not initialize it. The game would then rely purely on core systems. You'd lose the parallelism and perhaps some fidelity (the module-driven features wouldn't function unless reimplemented in the main loop), but the point is the rest of FDP can operate normally. This indicates that `ModuleHost` and modules are an *opt-in layer* – very powerful for many scenarios, but your game's base can still function without it if needed (useful for things like deterministic lockstep modes or certain console certification runs where multi-threading might be toggled off).

**Conclusion:** `ModuleHost` provides a structured way to harness multi-threading in an ECS-based game engine, splitting the simulation into a live core and a constellation of background modules. By understanding its scheduling, snapshot, and event mechanisms, you can design game systems that are both performant and cleanly separated. Use modules for what they're best at – concurrent, decoupled tasks – and you'll keep your frame rates high and your code organized as your game scales in complexity <sup>1</sup> <sup>103</sup>. With careful use of its debugging features and thoughtful choice of which features to modularize, `ModuleHost` can significantly extend the capabilities and scalability of the FDP engine while maintaining a solid ground truth in the live world <sup>104</sup>. Enjoy building with it, and happy modular gaming!

---

1 2 22 23 24 30 37 38 43 44 46 48 59 64 103 **ARCHITECTURE.md**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/docs/ARCHITECTURE.md>

3 4 5 6 **IModule.cs**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/ModuleHost.Core/Abstractions/IModule.cs>

7 12 13 14 18 19 20 21 27 28 49 50 51 52 53 54 55 56 58 60 61 62 63 65 66 68 69 70 71 72  
73 74 75 76 77 78 81 82 83 86 96 97 98 99 104 **IMPLEMENTATION-SPECIFICATION.md**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/docs/IMPLEMENTATION-SPECIFICATION.md>

8 9 25 31 32 33 34 35 36 39 40 41 42 47 67 100 101 102 **ModuleHostKernel.cs**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/ModuleHost.Core/ModuleHostKernel.cs>

10 11 79 80 88 **SystemScheduler.cs**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/ModuleHost.Core/Scheduling/SystemScheduler.cs>

15 16 17 26 29 57 **detailed-design-overview.md**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/docs/detailed-design-overview.md>

45 87 89 90 91 92 93 **AIModule.cs**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/Examples/Fdp.Examples.BattleRoyale/Modules/AIModule.cs>

84 85 94 95 **AnalyticsModule.cs**

<https://github.com/pjanec/ModuleHost/blob/5b29c029566bc3f854497f11d590d33f8f7a2d18/Examples/Fdp.Examples.BattleRoyale/Modules/AnalyticsModule.cs>