THE UNREAL RADAR · PRACTICES RADAR

The Unreal Dev Practices Radar

How to actually build an Unreal project: which version-control, build, architecture and tooling practices we'd adopt on a new production, which to trial, which to assess, and which to leave behind.

Practices Radar · Edition 2026.1

2026-06-13

Edited by Phil, MythicLemon

mythiclemon.com/radar

Editor’s note

Welcome to the first edition of the Unreal Dev Practices Radar. The two radars we publish alongside it rate *things* — Epic's engine subsystems, and the shape of the marketplace. This one rates *habits*: the version-control choices, build and cooking pipelines, code-architecture boundaries, and profiling and testing disciplines that, in our experience, decide whether a UE5 project glides toward ship or grinds to a halt at integration time.

A deliberate rule governs every blip here: we rate **classes of practice and tool**, never a named product. "Centralised VCS for binary assets" is a blip; no specific server is. "Distributed-build CI" is a blip; no specific build farm is. This keeps the radar honest — we sell developer tools ourselves, so naming and ranking rival products would be marking our own homework. Where a practice's tool-class overlaps something MythicLemon ships, you'll see a one-line disclosure on that blip. Read those as a conflict-of-interest flag, not a sales pitch: the verdict is the verdict regardless.

The rings are the familiar four, used here as adoption advice rather than maturity labels. **Adopt** means we'd set this up on day one of a new UE5 production and consider its absence a smell. **Trial** means it's worth committing to on a project that can absorb a little setup cost or a learning curve. **Assess** means understand it and prototype with it, but don't make it load-bearing yet. **Hold** means it's a habit to retire — fine where it's already entrenched, but don't start new work depending on it. These are opinions, not measurements; reasonable teams will disagree, and the smallest projects can rationally skip some of the heavier machinery.

Because this is the first edition, nothing carries a movement arrow — there's no prior radar to have moved from. The throughline, if there is one: Unreal punishes teams who treat it like a generic codebase. Its content is enormous and binary, its build is heavy, and its two-language Blueprint/C++ split has no equivalent elsewhere. The practices below are the ones that take those facts seriously.

— Phil, MythicLemon

> **This is our view, not a measurement.** The rings are a considered production-readiness opinion — not a quality score. We rate Epic’s engine, asset categories and practices, never a named third-party product. Where a category overlaps something we sell, we say so.

The radar at a glance

Source control & collaboration · Build, CI & cooking · Architecture & code · Profiling, testing & AI-assist

Adopt

We'd set this up on day one of a new production.

1. Centralised VCS for binary assets
2. One File Per Actor workflow
4. Distributed-build & graph-driven CI
5. Incremental & cached cooking
6. Blueprint / C++ boundary discipline
8. Data-driven design with DataTables & DataAssets
9. Profiling discipline (frame-level instrumentation)

Trial

Worth committing to where you can absorb some setup cost or a learning curve.

3. Git-LFS-style VCS for UE
7. Modular Gameplay Feature plugins
10. Automated functional & smoke tests
11. Source-control-friendly asset review
12. Lightweight in-team task tracking

Assess

Understand it and prototype with it — not yet load-bearing.

13. AI-assisted coding in UE

Hold

A habit to retire; don't start new work depending on it.

14. Monolithic level files & manual one-machine builds

The radar at a glance

Practice / tool-class	Quadrant	Ring	In one line
Centralised VCS for binary assets	Source control	**Adopt**	Locking + on-demand fetch; assets can't be merged.
One File Per Actor workflow	Source control	**Adopt**	Per-actor files make big maps collaborative.
Git-LFS-style VCS for UE	Source control	**Trial**	Fine for code-led teams; strains under heavy content.
Distributed-build & graph-driven CI	Build, CI & cooking	**Adopt**	Reproducible, parallelised builds as a graph.
Incremental & cached cooking	Build, CI & cooking	**Adopt**	Share cook/derived-data caches; stop re-cooking.
Blueprint / C++ boundary discipline	Architecture & code	**Adopt**	Decide on purpose what lives where.
Modular Gameplay Feature plugins	Architecture & code	**Trial**	Self-contained features; benefit scales with size.
Data-driven design (DataTables/DataAssets)	Architecture & code	**Adopt**	Designers tune data without recompiling.
Profiling discipline (instrumentation)	Profiling & testing	**Adopt**	Profile, don't guess — instrument early.
Automated functional & smoke tests	Profiling & testing	**Trial**	Catch regressions in CI; scope deliberately.
Source-control-friendly asset review	Profiling & testing	**Trial**	Make content changes reviewable + validated.
Lightweight in-team task tracking	Profiling & testing	**Trial**	Visible, low-friction board over heavy process.
AI-assisted coding in UE	Profiling & testing	**Assess**	Great for boilerplate; risky on UE-specific APIs.
Monolithic levels & manual one-machine builds	Build, CI & cooking	**Hold**	The old default; retire it for collaborative work.

Rings are adoption advice — our opinion on a new UE5 production, not a measurement. The smallest projects can rationally skip some of the heavier machinery.

The verdicts

Source control & collaboration

How a team shares a huge, binary-heavy project without trampling each other. The single biggest source of friction on most Unreal teams.

1 Centralised VCS for binary assets ADOPT

A centralised, file-locking version-control class is the default for any team sharing Unreal content.

The single most consequential setup decision an Unreal team makes is how it version-controls its content. A centralised VCS class — one that supports exclusive **file locking** and streams large binaries on demand rather than cloning the whole history — is what Unreal's own workflows assume, and it's the default we reach for the moment more than one person touches the same .uasset or .umap.

The reason is structural, not fashionable: Unreal assets are large binary blobs that **cannot be merged**. Two people editing the same Blueprint or material is a lost-work event, not a three-way merge. Exclusive checkout turns that from a recurring disaster into a non-issue, and on-demand fetching keeps multi-hundred-gigabyte projects workable. Adopt this on day one; retrofitting it onto a team that started elsewhere is painful.

Sources: Epic · Source control & collaboration

2 One File Per Actor workflow ADOPT

Storing each actor in its own file is what makes large worlds collaborative.

One File Per Actor (OFPA) stores each placed actor as its own small file rather than baking everyone's edits into one monolithic level file. It's the workflow that makes World Partition's large worlds genuinely team-friendly: two designers can dress different parts of the same map at the same time without colliding, because they're touching different files.

We **Adopt** it for any project of meaningful scope, but adopt it *consciously*. It changes what your version-control history looks like — thousands of tiny files instead of a few big ones — and your team needs to understand that a 'level' is now a directory of actors. Pair it with a centralised locking VCS and your collaboration story on big maps is largely solved.

Sources: Epic · World Partition & OFPA

3 Git-LFS-style VCS for UE TRIAL

Distributed VCS plus large-file storage and locking — viable for the right team, with caveats.

A distributed-VCS-plus-large-file-storage approach, with a locking layer bolted on for binary assets, is a legitimate way to run an Unreal project — especially for small, code-heavy, or open-source-adjacent teams who already live in that ecosystem and want one tool for code and content.

We place the class in **Trial** rather than Adopt because the seams show as content grows. Large-file storage and asset locking are add-ons here, not the native design, and very large repositories or big art teams tend to strain the model. If your project is code-led with modest content, trial it happily; if it's a content-heavy game with a large art team, lean toward the centralised-locking class above and treat this as the fallback, not the default.

Sources: Epic · Source control in UE

Build, CI & cooking

Turning source and content into a running game repeatably and automatically. Slow and manual by default; transformative when you invest.

4 Distributed-build & graph-driven CI ADOPT

Automated, distributed builds described as a dependency graph — the backbone of a sane Unreal pipeline.

A distributed build system feeding a graph-driven CI pipeline — where the steps to compile, cook, package and test are described once as a dependency graph and farmed out across machines — is the backbone of any Unreal project that ships more than once. Unreal builds are heavy; doing them by hand on one machine doesn't scale past a tiny team, and 'it built on my laptop' is not a release process.

We **Adopt** the class because the payoff is so large: reproducible builds, parallelised cook and compile, and a single source of truth for what 'a build' even means. The honest caveat is that standing it up is real work and overkill for a solo weekend project. For anything with a team and a ship date, the investment pays for itself within the first integration crunch.

Sources: Epic · Build automation & BuildGraph

5 Incremental & cached cooking ADOPT

Treat cooking as a cacheable, incremental step — not a from-scratch ritual.

Cooking — converting your content into platform-ready data — is one of the slowest steps in the Unreal pipeline, and the default mental model of 'cook everything from scratch every time' wastes enormous amounts of machine and human time. The practice we **Adopt** is treating cook output as a shared, cacheable artefact: derived-data and cook caches shared across the team and CI so that unchanged content is never re-processed.

Done well, this is the difference between a fifteen-minute iteration and a two-hour one. The discipline is mostly hygiene — keep the cache warm, share it, and don't blow it away reflexively when something goes wrong. It's cheap to adopt relative to how much time it returns.

Sources: Epic · Cooking & derived data

14 Monolithic level files & manual one-machine builds HOLD

The old default — one big map file, one person building by hand — is a habit to retire.

The legacy default for many teams was a small number of large, monolithic level files edited one-at-a-time, with builds and packages produced by hand on a single machine when someone remembered to. It works for one person on a small project, and that's exactly where it should stay.

We place it on **Hold**. For anything collaborative it's actively harmful: the big level file becomes a contention bottleneck that forces serialised editing, and the manual build is unreproducible and breaks the moment the one person who 'knows how to build it' is away. The modern replacements are elsewhere on this radar — One File Per Actor for the content side, distributed graph-driven CI for the build side. Don't start new collaborative work on the old way; migrate off it where you can.

Sources: Epic · World Partition & build automation

Architecture & code

The structural decisions — language boundaries, modularity, data-driven design — that keep a project legible as it grows.

6 Blueprint / C++ boundary discipline ADOPT

Deciding, on purpose, what lives in C++ versus Blueprint is the highest-leverage architecture call in Unreal.

Unreal's defining architectural fact is that you write in two languages at once. The teams that thrive treat the Blueprint/C++ boundary as a deliberate design decision rather than letting it accrete: systems, performance-critical loops, core data types and anything that needs real testing live in **C++**; tuning, composition, designer-facing logic and rapid iteration live in **Blueprint**, calling into the C++ below.

We **Adopt** the discipline because the failure mode of ignoring it is so common and so expensive: deep 'spaghetti' Blueprint graphs that implement core systems, can't be diffed in source control, can't be unit-tested, and bring a deep call stack to its knees at runtime. Decide where the line sits early, write it down, and hold it. This is the practice most likely to separate a project that scales from one that seizes up.

Sources: Epic · Blueprint & C++ workflow

7 Modular Gameplay Feature plugins TRIAL

Packaging features as self-contained, optionally-loaded plugins keeps a big project legible.

Structuring a project as a set of modular Game Feature plugins — each feature self-contained, with its own code, content and the ability to be activated or deactivated at runtime — is how Epic's own large projects stay maintainable. It enforces clean dependencies, lets teams own a feature end-to-end, and keeps a growing codebase from collapsing into one undifferentiated module.

We say **Trial** rather than Adopt because the up-front structure is real overhead and genuinely overkill for a small, single-team project. The benefit compounds with size and team count. For an ambitious or multi-team production, trial it early — retrofitting modularity onto a monolith later is far harder than starting that way. For a small game, a clean module layout may be all you need.

Sources: Epic · Game Features & modular gameplay

8 Data-driven design with DataTables & DataAssets ADOPT

Push tunable values into data so designers iterate without recompiling.

Driving gameplay from data — DataTables, DataAssets, curves and config rather than hard-coded values scattered through Blueprints and C++ — is a foundational Unreal practice we **Adopt** without reservation. It separates the *shape* of a system (code) from its *tuning* (data), so designers can balance, retune and add content without a programmer or a recompile in the loop.

The compounding benefits are real: tunables live in one inspectable place, content scales without code changes, and much of your game becomes describable as data you can validate and review. The discipline is to resist the easy hard-coded constant and ask 'should a designer be able to change this?' — and the answer is usually yes.

Sources: Epic · Data-driven gameplay

Profiling, testing & AI-assist

Knowing what your project is actually doing: performance discipline, automated tests, review habits, and where AI tooling helps.

9 Profiling discipline (frame-level instrumentation) ADOPT

Profile early, profile often, with proper frame-level instrumentation — never guess at performance.

Treating performance as an ongoing instrumented discipline rather than a pre-ship panic is one of the clearest dividing lines between projects that hit frame rate and projects that don't. The class of tooling here — frame-level instrumentation that captures CPU, GPU, memory and load timelines so you can see *where* the time and memory actually go — should be wired into your workflow from early, not bolted on at the end.

We **Adopt** the discipline. The recurring lesson in Unreal is 'profile, don't guess': the bottleneck is almost never where intuition says it is. Capture traces against your target hardware regularly, watch for regressions as content lands, and make a captured frame — not a hunch — the thing you optimise against.

> ⚖ MythicLemon ships an in-editor profiling/metering tool, so we have a commercial interest in this tool-class. We'd place the practice in Adopt regardless — it's the standard Unreal discipline — but the overlap is worth flagging.

Sources: Epic · Unreal Insights & profiling tools

10 Automated functional & smoke tests TRIAL

Automated functional and gameplay tests catch regressions a human pass never will.

Building a layer of automated tests — functional tests, gameplay smoke tests that boot a level and exercise core flows, and unit tests over your C++ systems — is a practice we want every serious Unreal team to grow toward. It turns 'did we break the main menu again?' from a manual ritual into something CI answers on every change, and it pairs naturally with the build pipeline above.

We place it at **Trial** rather than Adopt with eyes open: Unreal's automation framework has real rough edges, content-heavy games are genuinely hard to test end-to-end, and over-investing in brittle tests can cost more than it saves. Start with smoke tests on critical paths and unit tests on the C++ that the Blueprint layer can't reach (another reason to keep that boundary clean), then expand where the regressions actually hurt. Commit to it — just scope it deliberately.

Sources: Epic · Automation & functional testing

11 Source-control-friendly asset review TRIAL

Make content changes reviewable — naming standards, validation rules, and human eyes before it lands.

Code gets reviewed; content usually doesn't, and that asymmetry is where a lot of Unreal project rot comes from. The practice here is to make asset changes reviewable: enforced naming and folder conventions, automated asset-validation rules that fail a check-in when something is mis-set-up, and a lightweight human review step before binary content lands in the mainline.

We say **Trial** because the tooling is more bespoke than code review and asks for team buy-in to stick. But it pays back fast — validation rules catch the broken-reference and wrong-import-setting mistakes that otherwise surface days later as a mysterious cook failure, and conventions keep a growing content tree navigable. Pair it with data-driven design, where so much of your 'content' is reviewable data in the first place.

> ⚖ MythicLemon ships in-editor developer tools, including in-editor documentation and a charting/visualisation utility that teams use to surface this kind of project hygiene — so this tool-class overlaps our line. The verdict stands on the practice's merits.

Sources: Epic · Asset naming, validation & data validation

12 Lightweight in-team task tracking TRIAL

A visible, lightweight task board beats heavyweight process for most game teams.

Keeping work visible on a lightweight task board — a simple kanban-style flow that a small team can actually maintain — is the project-management practice we'd reach for over heavyweight, ceremony-laden process. Game development is exploratory and content-heavy; a board that shows what's in flight, what's blocked and what's done, without demanding hours of upkeep, is usually the right weight of tooling.

We say **Trial** because the right amount of process is genuinely team-dependent and easy to overdo — the failure mode here is *too much* tracking, not too little. Adopt the principle of visible, low-friction tracking; calibrate the exact tool to your team's size and rhythm rather than imposing a process built for a much larger org.

> ⚖ MythicLemon ships an in-editor kanban tool, so this tool-class overlaps our line directly. We're rating the practice, not our product — and we deliberately keep the verdict at Trial precisely because the right amount of process is team-dependent.

13 AI-assisted coding in UE ASSESS

Useful for boilerplate and exploration; not yet something to depend on for Unreal-specific work.

AI-assisted coding — generative help for writing and explaining C++, scaffolding boilerplate, and navigating an unfamiliar codebase — has clearly earned a place in the toolkit, and for general-purpose code and learning we already find it genuinely useful. The trajectory is steep and this entry is the one most likely to climb the radar in future editions.

We hold it at **Assess** for *Unreal-specific* work specifically. The engine's APIs are large, version-sensitive and full of conventions that general models still get subtly wrong — confidently generating code against an API shape that doesn't exist in your version, or missing Unreal idioms around reflection, garbage collection and the Blueprint boundary. Use it to accelerate and to explore, but keep a human and a compiler firmly in the loop, and don't let generated Unreal code into the mainline unreviewed. Assess it actively; we expect to revisit this verdict soon.

Sources: Epic · UE 5.7 release notes

FAQ

Why don't you name specific tools — the version-control server, the build farm, the AI assistant?

Because we sell developer tools on Fab, naming and ranking specific products would be marking our own homework. So this radar rates classes of practice and tool, not individual products. Where a class overlaps something we ship, we disclose it on the blip.

Isn't all this overkill for a solo developer or a tiny team?

Some of it, yes — and we say so on the blips. Distributed CI, modular Game Feature plugins and heavy test suites earn their keep with team size and project scope. A solo project can rationally run lighter. The architecture and profiling habits, though, pay off at every scale.

These are opinions — what are they based on?

This is an editorial edition: the verdicts come from building and shipping on Unreal ourselves and from the working consensus of experienced UE teams, anchored to Epic's own documented workflows where one exists. They're adoption advice, not measurements — which is exactly why the rings here are advice-flavoured rather than data-derived.

How often does this radar come out?

Roughly twice a year, alongside our engine and market radars. From the next edition, movement arrows will mark where we've changed our minds about a practice — separate from any change in Epic's tooling.

HOW TO CITE

```
MythicLemon, "The Unreal Dev Practices Radar", edition 2026.1 (2026-06-13). The Unreal Radar.
https://www.mythiclemon.com/radar/dev-practices-radar-2026-1
```