NovaEngine — Master Plan & Build‑Out Guide (v2025-08-15)

Goal: Build a modern, extensible, production‑grade game engine and editor with an optional integrated AI assistant. The engine must be fully usable without AI and support Windows first, then Linux/macOS, with a Vulkan renderer and an abstraction layer for future DX12/Metal backends. This guide describes every step from repo bootstrap to shipping games, including architecture, implementation steps, QA, performance, and long‑term roadmap.

Principles: Data‑Oriented Design • Deterministic builds • Local‑first (no secrets/telemetry) • Modular interfaces • Hot‑reload everywhere • Testability • Strict approvals for AI‑proposed changes.

# 1. High‑Level Architecture

Core Modules:

• Core (logging, time, config) — small, dependency‑free. • Platform (windowing, file system, threads). • Renderer (abstraction + Vulkan backend; future DX12/Metal). • ECS (archetype/SoA registry, systems scheduler). • Assets (importers, caching, hot‑reload, GUIDs, dependency graph). • Scripting (Lua for gameplay; optional Python for tools). • Physics (Jolt recommended for MIT license and performance; Bullet optional). • Audio (miniaudio default; FMOD via IAudio abstraction optional). • Animation (skeletal, retargeting, IK; motion matching roadmap). • Navigation/AI (navmesh, pathfinding, behavior trees). • Networking (client/server, replication, rollback netcode for action games). • Editor (ImGui docking; Hierarchy, Inspector, Console, Profiler, AI Command Palette [optional]). • Tooling (profiler, frame capture hooks, unit/integration tests, CI/CD).

Data: JSON/TOML for config/scene; binary packed formats for runtime streaming; GLTF/FBX import; compressed texture formats (BCn/ASTC).

Safety: Assertions in Debug, robust error reporting, device‑lost handling, crash minidumps, reproducible builds.

# 2. Repository & Build System

• Language: C++20 (minimal STL). • Build: CMake ≥ 3.21. • IDE: VS2022 on Windows. • Options:

\

-DNOVA\_BUILD\_EDITOR=ON

-DNOVA\_FETCH\_DEPS=ON

-DNOVA\_WITH\_AI=OFF|ON

-DNOVA\_ENABLE\_TESTS=ON

-DNOVA\_ENABLE\_PROFILER=ON

Third‑party via FetchContent: volk, GLFW, ImGui (docking), glm, Lua (or LuaJIT), Jolt, miniaudio, cgltf (or Assimp), stb, spdlog (or custom Log).

Targets: NovaEditor (GUI), NovaRuntime (headless/game), NovaTools (CLI). Artifacts: zipped builds with required DLLs and assets.

# 3. Exact Step‑By‑Step Implementation

## Phase 0 — Bootstrap (Day 0–1)

0.1 Initialize repo, license (MIT), .editorconfig, .gitattributes (LF normalization), CODEOWNERS.

0.2 CMake skeleton with options above. Set VK\_NO\_PROTOTYPES globally; add volk as loader. Configure warning levels and sanitizers (ASan/UBSan on non‑Windows).

0.3 FetchContent: glfw, imgui (docking), glm, lua, volk. Wire include paths and compile definitions.

0.4 Create minimal app: initialize Log, create a GLFW window, init volk, create VkInstance/Device/Swapchain, clear color loop, shutdown cleanly.

0.5 Add ImGui context + docking, backend bindings (imgui\_impl\_glfw/vulkan), draw demo window to validate inputs and rendering.

## Phase 1 — Editor + ECS + PBR Cube (Day 1–3)

1.1 Implement IRenderer and a VulkanRenderer PIMPL. Add renderpass, depth buffer, descriptor pool, command buffers, sync (fence + semaphores).

1.2 Shaders: create Assets/Shaders/pbr.vert and pbr.frag with a single directional light, camera UBO, and roughness/metallic parameters.

1.3 Mesh: create a unit cube vertex/index buffer; support interleaved positions/tangents/UVs; upload to device local memory with staging.

1.4 ECS: simple archetype or sparse-set with Transform, MeshRenderer, Camera, Light components. Systems: TransformUpdate, RenderSystem.

1.5 Scene: Assets/Scenes/hello.json (camera + cube + light). Auto-load at startup. If missing, create it.

1.6 Lua VM: init once; on Play, run Scripts/rotate\_cube.lua and tick on every frame (expose Time.delta\_seconds and an API to rotate an entity).

1.7 Editor UI: Hierarchy (tree of entities), Inspector (component fields), Console (logs), Stats (FPS, CPU/GPU timings). Persist docking layout.

1.8 Input: WASD + mouse-look editor camera, capture/release mouse; configurable sensitivity; save camera settings to Config/editor.toml.

1.9 Resize Handling: implement swapchain recreation; keep pipelines that depend on extent.

## Phase 2 — Asset System + Materials + Hot‑Reload (Week 1)

2.1 Asset GUID registry; path ↔ GUID DB (Assets/assetdb.json). Dependency tracking (materials reference textures; meshes reference materials).

2.2 Importers: cgltf for GLTF2; stb\_image for images; normal/tangent generation if needed; sRGB handling; mipmap generation at import or runtime.

2.3 Material system: physically‑based material with textures (baseColor, roughness, metallic, normal, emissive). JSON material files with references.

2.4 Hot‑reload: file watcher triggers recompile of shader variants and reload of textures/materials/meshes; non‑disruptive pipeline state hot‑swap.

2.5 GPU Instancing: per‑instance matrices + material index; frustum culling on CPU; HZB for occlusion (roadmap).

## Phase 3 — Lighting, PostFX, and GI Options (Week 2)

3.1 Shadow mapping for directional lights (cascaded), point/spot (cube/atlas).

3.2 PostFX: HDR, tonemapping (ACES), bloom, vignette, exposure auto.

3.3 GI options roadmap: DDGI (probe volumes), SSGI, RTGI (if ray tracing enabled). Add toggles and fallbacks.

3.4 Temporal AA with history management; optional dynamic resolution; upscalers (FSR2 path with plugin architecture for DLSS/XeSS when available).

## Phase 4 — Physics & Character (Week 3)

4.1 Integrate Jolt (preferred) via IPhysics interface; components: Rigidbody, Collider (Box/Sphere/Capsule/Mesh), Trigger, CharacterController.

4.2 Sync ECS ↔ Physics in fixed timestep; interpolation for render; raycast queries in editor (gizmos).

4.3 Physics materials and layers; events (on\_collision, on\_trigger) piped to Lua callbacks.

## Phase 5 — Animation (Week 4)

5.1 Skeletal animation loader from GLTF; GPU skinning; animation state machine assets (JSON).

5.2 Retargeting setup; basic IK (two‑bone, CCD/FABRIK).

5.3 Blend spaces; additive layers; animation events to gameplay scripts.

## Phase 6 — Terrain, Foliage, and Procedural Tools (Week 5)

6.1 Heightmap terrain with sculpt/paint layers, splat textures; streaming tiles; impostors for far LOD.

6.2 Foliage/instancing system with density/biome masks; GPU culling; wind animation.

6.3 Procedural MeshGen module (cube, sphere, capsule, cylinder, cone, plane, torus) + CSG (union/diff/intersect) as roadmap.

## Phase 7 — Audio (Week 6)

7.1 miniaudio backend via IAudio; 2D/3D playback, listener, spatialization; effects bus; basic occlusion. FMOD plugin optional.

7.2 Asset types: sound banks, events. Lua bindings for play/stop/parameters.

## Phase 8 — Scripting & Tools (Week 6–7)

8.1 Lua bindings for core components and math; hot‑reload of scripts with error surfacing in Console.

8.2 Optional Python tools for batch import/validation/build; tools/ai\_cli.py for memory search (no network).

8.3 Visual scripting (roadmap) with node graph generating Lua; versionable JSON graphs.

## Phase 9 — Networking (Week 8)

9.1 NetTransport abstraction (ENet or similar) with reliability classes. Entity replication with prioritized channels; delta compression.

9.2 Deterministic simulation mode and rollback netcode for action games (lockstep or GGPO‑like), with input prediction and reconciliation.

9.3 Dedicated server build target and headless sim.

## Phase 10 — Editor Power Features (Week 9)

10.1 Prefabs with overrides; variant system. Undo/redo command stack with snapshot‑diff strategy.

10.2 Sequencer/Timeline for cutscenes and scripted events; keyframe tracks for transforms/material params.

10.3 Node‑based Material Editor; shader graph → SPIR‑V generation pipeline (intermediate JSON).

## Phase 11 — AI Assistant (Optional; Always ON by Default for Usability)

11.1 Build flag NOVA\_WITH\_AI (default ON) to compile AI features.

11.2 Memory subsystem (local JSONL) writing to /.ai/memory/conversation.jsonl, session summaries, and graph.json.

11.3 Proposal flow: Natural language → CHANGE PROPOSAL (summary + unified diff) written to /.ai/proposals/<id>/. Requires APPROVE: <id> before applying.

11.4 Proactive suggestions (≤3) persisted in /.ai\_issues/. Editor AI Palette lists and applies by generating proposals.

11.5 Deterministic NL→Actions mappers for scene edits (create primitive, add logic, terrain paint) that emit assets + scene diffs. No network calls.

## Phase 12 — Build/Export/CI (Week 10)

12.1 Export wizard: choose target (Win), build type, include assets, strip debug symbols (optional), write Builds/<Project>/<timestamp>/.

12.2 CI: GitHub Actions multi‑config builds (Debug/Release), cache deps, publish artifacts; unit tests and smoke tests on boot and scene load.

12.3 Deterministic build manifest with hashes for assets and shaders; reproducible archives.

## Phase 13 — QA, Telemetry (Local), and Compliance

13.1 Unit tests (math, ECS, serialization), integration tests (scene load), GPU capture test recipes (RenderDoc/Vulkan validation).

13.2 Local‑only metrics (frame times, memory) saved to Logs/metrics.jsonl—no external telemetry. Privacy first.

13.3 Accessibility: color‑blind friendly UI themes, remappable controls, scalable fonts, high‑contrast mode.

# 4. Detailed Command Cookbook (Windows)

\

# Configure & build (editor without AI)

cmake -S . -B build -G "Visual Studio 17 2022" -A x64 ^

-DNOVA\_BUILD\_EDITOR=ON -DNOVA\_FETCH\_DEPS=ON -DNOVA\_WITH\_AI=OFF

cmake --build build --config Release

.\build\bin\Release\NovaEditor.exe

# Enable AI features later

cmake -S . -B build -G "Visual Studio 17 2022" -A x64 -DNOVA\_WITH\_AI=ON

cmake --build build --config Release

# Run tests

ctest --test-dir build -C Release --output-on-failure

# Package

cpack -B dist -C Release

# 5. Directory Layout (Target)

\

/ (repo)

CMakeLists.txt

/src

/app # NovaEditor, NovaRuntime

/engine

/core # Log, Time, Config

/platform # Windowing, FS, Threads

/renderer # IRenderer + backends (vk, dx12, metal)

/ecs # Registry, Systems

/assets # Importers, DB, hot-reload

/scripting # Lua (gameplay), Python (tools optional)

/physics # IPhysics (Jolt/Bullet)

/audio # IAudio (miniaudio/FMOM)

/animation

/navigation

/network

/editor # Panels, gizmos, AI palette (optional)

/tools # CLI tools, ai\_cli.py

/Assets

/Scenes, /Materials, /Meshes, /Textures, /Shaders, /Audio

/Samples/HelloWorld3D

/Config

/Scripts

/.ai (optional when AI on)

/memory, /proposals, /.ai\_issues

/Tests

/Docs

/Builds

# 6. Rendering Deep‑Dive (Modern Features)

• Clustered/Forward+ lighting, bindless descriptors, material/texture arrays, GPU culling, HZB occlusion, meshlets (roadmap), ray tracing (optional).

• Shader compilation pipeline: HLSL/GLSL → SPIR‑V; reflection → auto‑binds; permutation keys for features; hot‑reload; shader cache on disk.

• Temporal stability: TAAU, velocity buffers, reactive masks for upscalers; history management.

# 7. Performance & Memory

• Frame graph orchestrates passes; transient resources; aliasing; GPU/CPU timelines. • Job system with work‑stealing for ECS systems; jobified animation/skin/IK; async streaming for assets.

• Budgets per domain (CPU, GPU, VRAM, RAM); live HUD; capture/playback micro‑benchmarks.

# 8. Multiplayer & Determinism

• Network‑ready components with serialization; snapshot + delta; interest management. • Deterministic mode for rollback; avoid non‑deterministic math; fixed‑step sim; input prediction.

# 9. Editor UX & Productivity

• Command Palette, search‑everything, multi‑select editing, prefab overrides diff view, safe mode (no scripts), crash‑recovery autosave, source control integration.

• Import pipelines with rule presets per folder, build‑time validation, ‘Fix It’ buttons for common issues.

# 10. AI Assistant (Optional) — Guardrails

• Local‑first memory and proposals only. No auto‑apply. All changes go through a diff + explicit approval. • Redaction/CLEAR/FORGET commands supported via cli. • Deterministic action mappers for scene/terrain/object edits; zero network usage.

# 11. Roadmap Beyond v1

• USD (Universal Scene Description) import/export; collaborative editing (CRDTs) with local peer‑to‑peer sync; path tracing mode; motion matching; cloth and fluids; node‑based gameplay scripting; console platform abstraction (subject to SDK licenses).

# 12. Acceptance Criteria Checklist (Ship v1)

• Editor opens, loads HelloWorld3D, renders lit PBR cube; Play toggles Lua‑driven rotation; logs show subsystem init. • Asset hot‑reload works for textures/materials and scripts; swapchain resize is stable. • Physics demo scene with character controller; audio 3D demo works; export wizard creates a runnable build. • All features functional with AI turned OFF; when ON, proposals and memory files are persisted and searchable via CLI.

# Appendix A — Exact Lua Example

\

-- Scripts/rotate\_cube.lua

ROTATE\_DEG\_PER\_SEC = 90

function on\_update(dt)

local yaw = ROTATE\_DEG\_PER\_SEC \* dt

Entity("PlayerOrb"):add\_yaw(yaw)

end

# Appendix B — Minimal Shader Stubs

\

// Assets/Shaders/pbr.vert (GLSL)

#version 460

layout(location=0) in vec3 inPos;

layout(location=1) in vec3 inNormal;

layout(location=2) in vec2 inUV;

layout(set=0,binding=0) uniform Camera { mat4 VP; mat4 M; } cam;

out gl\_PerVertex { vec4 gl\_Position; };

void main(){ gl\_Position = cam.VP \* cam.M \* vec4(inPos,1.0); }

// Assets/Shaders/pbr.frag (GLSL)

#version 460

layout(location=0) out vec4 outColor;

void main(){ outColor = vec4(1.0,0.8,0.6,1.0); } // placeholder

# Appendix C — Build Scripts (PowerShell)

\

$ErrorActionPreference='Stop'

cmake -S . -B build -G "Visual Studio 17 2022" -A x64 -DNOVA\_BUILD\_EDITOR=ON -DNOVA\_FETCH\_DEPS=ON -DNOVA\_WITH\_AI=OFF

cmake --build build --config Release

$exe = (Get-ChildItem -Recurse -File -Filter NovaEditor.exe .\build | Select-Object -First 1).FullName

& $exe

# NEW: Always-On AI, Local Models (No API), and Memory Contract

This section adds concrete instructions for: (A) telling the AI how to remember and behave in every session; (B) enabling a fully local AI runtime with no API keys; (C) keeping AI enabled by default while preserving full parity for manual workflows.

## A) "Tell the AI" — Memory & Behavior Contract (Copy‑Paste into your AI)

Paste the following in the beginning of a new chat with your AI (or into your engine’s built-in AI ‘System Prompt’ slot). Use it whenever you upload this DOCX and the project ZIP. It makes the AI persist everything locally and keep approvals/diffs strict:

SYSTEM / ROLE CONTRACT

You are my engine-integrated assistant. You must:

1) Load Context

- When I upload the NovaEngine\_MasterPlan.docx and project ZIP, extract their contents mentally and acknowledge the version/date.

- Build a local knowledge index from them (sections, headings, file paths). Do NOT rely on network calls.

- Maintain a rolling 'Memory Pack' summary for this project.

2) Persistent Memory (Local, Git-Versioned)

- For every message and action, append a JSONL entry to /.ai/memory/conversation.jsonl with timestamp, role, summary, and referenced files.

- Maintain /.ai/memory/summaries/<session-id>.md and /.ai/memory/graph.json (entities ↔ subsystems ↔ files).

- Provide a local search interface: tools/ai\_cli.py --search "<query>".

- Support commands: EXPORT\_MEMORY, CLEAR\_MEMORY, FORGET:<range|id>, REDACT:<pattern>.

3) Proposals & Approvals

- Natural language requests → CHANGE PROPOSAL (short rationale + unified diff) under /.ai/proposals/<id>/{summary.md,patch.diff}.

- Never auto-apply. Wait for explicit APPROVE: <id>. Respect DENY with a short reason.

- Keep ≤3 proactive suggestions in /.ai\_issues/; include risk, rollback, and tests.

4) Dual-Path Parity (AI Always Enabled, Manual Fully Capable)

- Every AI feature must have an equivalent manual flow (UI panel or CLI). Provide clear, step-by-step manual instructions when asked.

- Default to AI-enabled behavior, but never block manual control. Always expose the underlying engine operation(s) and file changes.

5) Deterministic NL→Actions

- Map commands like “Create a red sphere radius=1 at (0,1,0) named PlayerOrb” into concrete engine diffs (scene JSON, assets, scripts). Return deterministic patches.

- Provide migration notes when public APIs change.

6) Non-Goals & Privacy

- No cloud calls. No secrets. Do not reveal chain-of-thought; provide concise reasoning and diffs only.

## B) Local AI (No API) — Reference Architecture

Run models locally so the engine does not depend on any external API. Recommend an abstraction shaped like IAgent with pluggable backends:

IAgent (abstract)

- generate(prompt, tools, system) -> {text, tool\_calls}

- embeddings(texts[]) -> float[dim][] (optional for retrieval)

- tokenize()/detokenize() (optional)

Backends (no API):

• llama.cpp / gguf runtime (C/C++) with quantized models (Q4\_K, Q5\_K, Q8\_0) for offline inference.

• Local inference servers (e.g., ollama) with UDS/localhost calls (no keys) — still considered local.

• Optional small local embedding model for search over /.ai/memory and repo files.

Memory & Retrieval

• conversation.jsonl + summaries + graph.json as ground-truth memory.

• Optional SQLite/FAISS index for quick vector search (fully local).

Tool Calling

• Deterministic function registry that maps AI tool calls → engine actions (create entity, edit component, terrain paint, etc.).

• All tools must be executable via CLI/UI without AI as well.

Ship with a small set of local models and a model selection UI (Model, Quantization, Context Length). Provide per-project defaults in Config/ai.toml.

## C) Always-On AI With Manual Parity — UX Rules

• AI is ON by default (NOVA\_WITH\_AI=ON), but every operation is mirrored by a manual path. • For each AI command, display the exact manual steps and the patch to be applied. • Provide a ‘Explain & Show Diff’ button before applying any change. • Allow ‘Record Macro from Manual Steps’ to create re-usable commands without AI.

## D) Optional “Other AIs Without API Keys”

Integrate additional local backends through the same IAgent interface (e.g., other offline runtimes). They must operate fully offline and never require a cloud API key. Use adapters so the editor can hot-swap backends at runtime.

## E) Startup Checklist (Enable AI by Default, User Can Disable)

\

- CMake: -DNOVA\_WITH\_AI=ON by default; expose a menu toggle in the editor: Settings → AI → Enable (checkbox).

- On first launch, create /.ai/ folders and write a quickstart memory entry.

- Show a one-time info panel: “AI is enabled. Every AI action has a manual equivalent.”

- If disabled, hide AI panels but keep manual workflows intact.

## F) What To Say When You Upload the DOCX and ZIP (Short Version)

Use this as your first message to the AI:

"Load the attached NovaEngine\_MasterPlan.docx and project ZIP. Build a project knowledge index and persist memory locally under /.ai/.

From now on:

- Log every message/action to /.ai/memory/conversation.jsonl, keep session summaries and a graph map.

- For any request, produce a CHANGE PROPOSAL with unified diff and wait for APPROVE:<id>.

- Keep AI always enabled, but every AI capability must have a manual path with step-by-step instructions and the same end result.

- Never call the network or require API keys. Use only local tools and deterministic mappings."

## G) Implementation Tasks To Add To Backlog

\

1. CMake: NOVA\_WITH\_AI default ON; menu toggle + config persist.

2. Implement IAgent abstraction and a llama.cpp-backed LocalAgent.

3. Implement ToolRegistry mapping tool calls to concrete engine operations.

4. Build ai\_cli.py with SEARCH / EXPORT\_MEMORY / CLEAR\_MEMORY / FORGET / REDACT.

5. Ensure every AI action emits a patch.diff and a manual-steps.md.

6. Add model selection UI and Config/ai.toml; ship a default local model pack.

— Appended on 2025-08-15.

# Appendix D — AI Starter Patches & Files

Included patch bundle: NovaEngine\_AI\_Starter\_Patches.zip. It contains unified diffs for CMake and Editor AI settings, plus full source files for IAgent, LocalAgent, and ToolRegistry, and a sample Config/ai.toml.

## Apply Instructions (PowerShell)

# 1) Unzip into your repo root

Expand-Archive -Path .\NovaEngine\_AI\_Starter\_Patches.zip -DestinationPath .\\_ai\_patches -Force

# 2) Apply diffs (requires git installed)

git apply .\\_ai\_patches\patches\0001-cmake-ai-toggle-and-sources.diff

git apply .\\_ai\_patches\patches\0002-editor-ai-toggle-settings.diff

# 3) Copy new files (if not already present)

Copy-Item .\\_ai\_patches\files\src\engine\ai\IAgent.h -Destination .\src\engine\ai\ -Force

Copy-Item .\\_ai\_patches\files\src\engine\ai\LocalAgent.h -Destination .\src\engine\ai\ -Force

Copy-Item .\\_ai\_patches\files\src\engine\ai\LocalAgent.cpp -Destination .\src\engine\ai\ -Force

Copy-Item .\\_ai\_patches\files\src\engine\ai\ToolRegistry.h -Destination .\src\engine\ai\ -Force

Copy-Item .\\_ai\_patches\files\src\engine\ai\ToolRegistry.cpp -Destination .\src\engine\ai\ -Force

New-Item -ItemType Directory -Force -Path .\Config | Out-Null

Copy-Item .\\_ai\_patches\files\Config\ai.toml -Destination .\Config\ -Force

# 4) Reconfigure & build

cmake -S . -B build -G "Visual Studio 17 2022" -A x64 -DNOVA\_BUILD\_EDITOR=ON -DNOVA\_FETCH\_DEPS=ON -DNOVA\_WITH\_AI=ON

cmake --build build --config Release

# 5) Run editor and toggle AI in Settings → AI; adjust model path; Save AI Settings.

# Deterministic AI Tool Example + CLI Memory Search

Below is a ready-to-use deterministic AI tool implementation for 'Create Plane 20x20 at (0,0,0)'. This maps natural language into a concrete engine change proposal with a scene JSON diff.

// ToolRegistry.cpp (excerpt)

tools["create\_plane"] = [](const ToolParams& p) {

// Hardcoded example: plane 20x20

std::string entityName = p.get("name", "GroundPlane");

SceneChange change;

change.description = "Add plane 20x20 at (0,0,0)";

change.diff = R"({"entities":[{"name":")" + entityName + R"(","mesh":"plane20x20","position":[0,0,0]}]})";

return change;

};

In practice, replace the hardcoded values with parameters extracted from the AI's parsed NL command.

\*\*CLI Tool:\*\* Below is ai\_cli.py snippet to search memory JSONL for a given query.

#!/usr/bin/env python3

import sys, json, pathlib

memfile = pathlib.Path('.ai/memory/conversation.jsonl')

q = " ".join(sys.argv[1:]).lower()

for i, line in enumerate(memfile.read\_text().splitlines(), 1):

if q in line.lower():

print(f"[{i}] {line}")

Usage:

python tools/ai\_cli.py --search "rotating logic"

# FINAL: AI Startup Prompt (with Multiple ZIPs + DOCX)

When starting a new session with the AI, paste this exact text:

Load the attached files:

1. NovaEngine\_MasterPlan.docx (full design and implementation plan)

2. NovaEngine.zip (current project source)

3. NovaEngine\_AI\_Starter\_Patches.zip (AI integration patches)

Instructions:

- Merge the plan from DOCX into your working memory, resolving any conflicts so all requirements are consistent.

- Build a project knowledge index from both zips (source + patches) and link them to the plan.

- Persist all memory locally under /.ai/ (conversation.jsonl, summaries, graph.json).

- For any request, produce a CHANGE PROPOSAL (short rationale + unified diff) under /.ai/proposals/<id>/ and wait for APPROVE:<id> before applying.

- Keep AI always enabled by default, but every AI action must have a fully capable manual UI/CLI equivalent with step-by-step instructions.

- Map natural language commands deterministically to engine changes (scene JSON, assets, Lua, terrain edits) and return patches.

- Never use network calls or API keys; all models and tools are local.

# Permanent Free Use & Licensing Rules

NovaEngine will remain permanently free to use, modify, and distribute for both personal and commercial projects, with no royalties, subscription fees, or hidden costs. The following rules ensure this policy is preserved:

1. Core Engine License: All original NovaEngine source code is released under the MIT License. This grants unlimited commercial and personal use, modification, and redistribution, with attribution but no royalties.

2. Third-Party Dependencies: Only include libraries with permissive, OSI-approved licenses (MIT, BSD, Apache 2.0, Zlib) that allow commercial use without fees. Avoid GPL/LGPL unless you intend to open-source all linked code.

3. AI Models: Only integrate and distribute open-weight AI models with permissive licenses allowing commercial use and redistribution. All inference must run locally without API calls or per-token costs. Suggested runtime: llama.cpp or equivalent.

4. Assets: All bundled textures, models, sounds, and other default assets must be CC0 or similarly permissive, royalty-free licenses. Maintain a CREDITS.md in /Assets listing sources and licenses.

5. Distribution Policy: Users of NovaEngine can build, modify, and ship projects without paying fees or royalties. State clearly in README.md that NovaEngine is free and open-source.

6. Compliance Tracking: Maintain a LICENSES.md in the repo root to track all third-party code/assets and their licenses.