

UStudio Pro
User Guide and Reference

Version 1.22.0

The C++ and Blueprint workflow for Unreal Engine 5, in VS Code and Cursor: IntelliSense, references, build and debug, the live Unreal Editor, Blueprints as C++ (and converted to C++), classes made with nodes, wizards, Python, source control and AI agents.

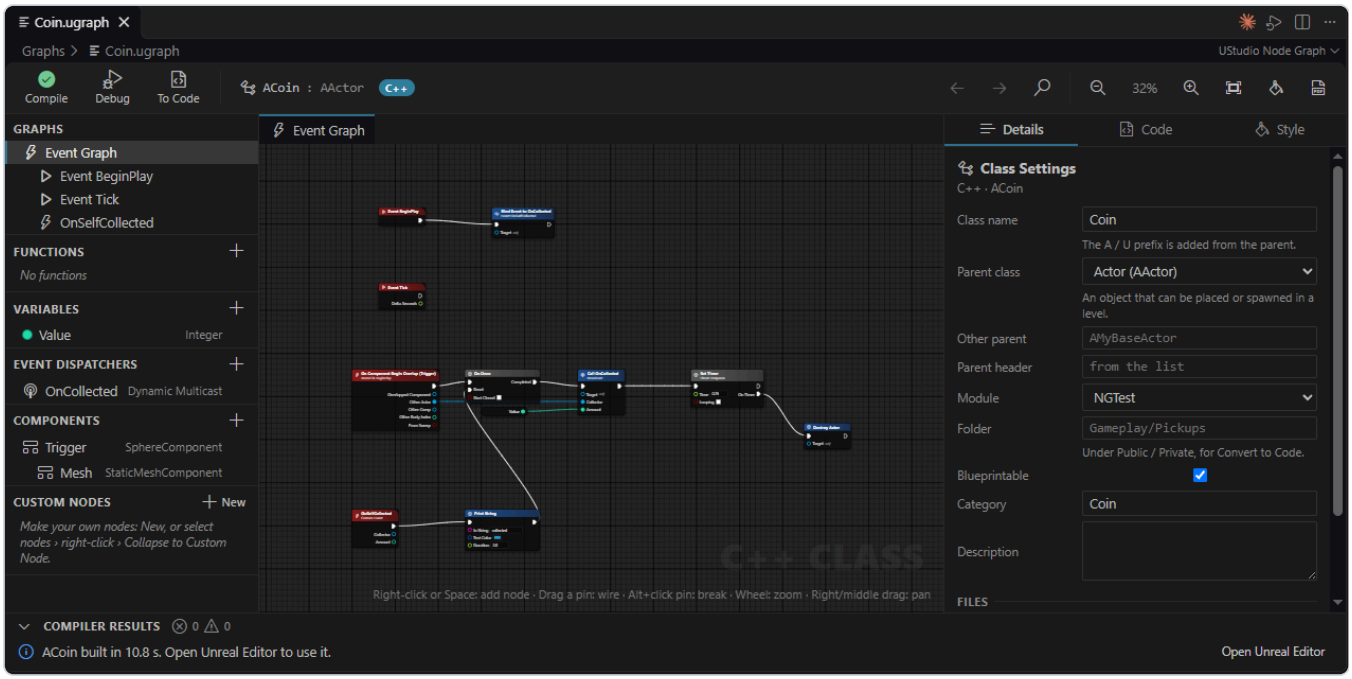
November 7, 2026 · Unreal Engine 5.0 – 5.8 · Windows 10 and 11 · discord.gg/4f6VpWpzp5

Contents

1. Node Graphs: classes made with nodes
2. Converting Blueprints to C++
3. Getting started
4. Code intelligence
5. Build and debug
6. Live Unreal Editor
7. Blueprints and assets
8. Node Graphs
9. Wizards and templates
10. Python for Unreal Editor
11. Source control and history
12. Navigation and workspace
13. AI agents
14. Cursor support
15. Free vs Pro
16. Requirements
17. Privacy
18. Support
19. License
20. Release notes 1.15.2 – 1.22.0

Node Graphs: classes made with nodes

New in 1.19.0, extended in 1.20.0, 1.21.0 and 1.22.0. Build C++ and Python classes with Blueprint-style nodes in VS Code, compile them into the running Unreal Editor to use them at once, watch the nodes run, and turn the graph into regular project code when it is done.

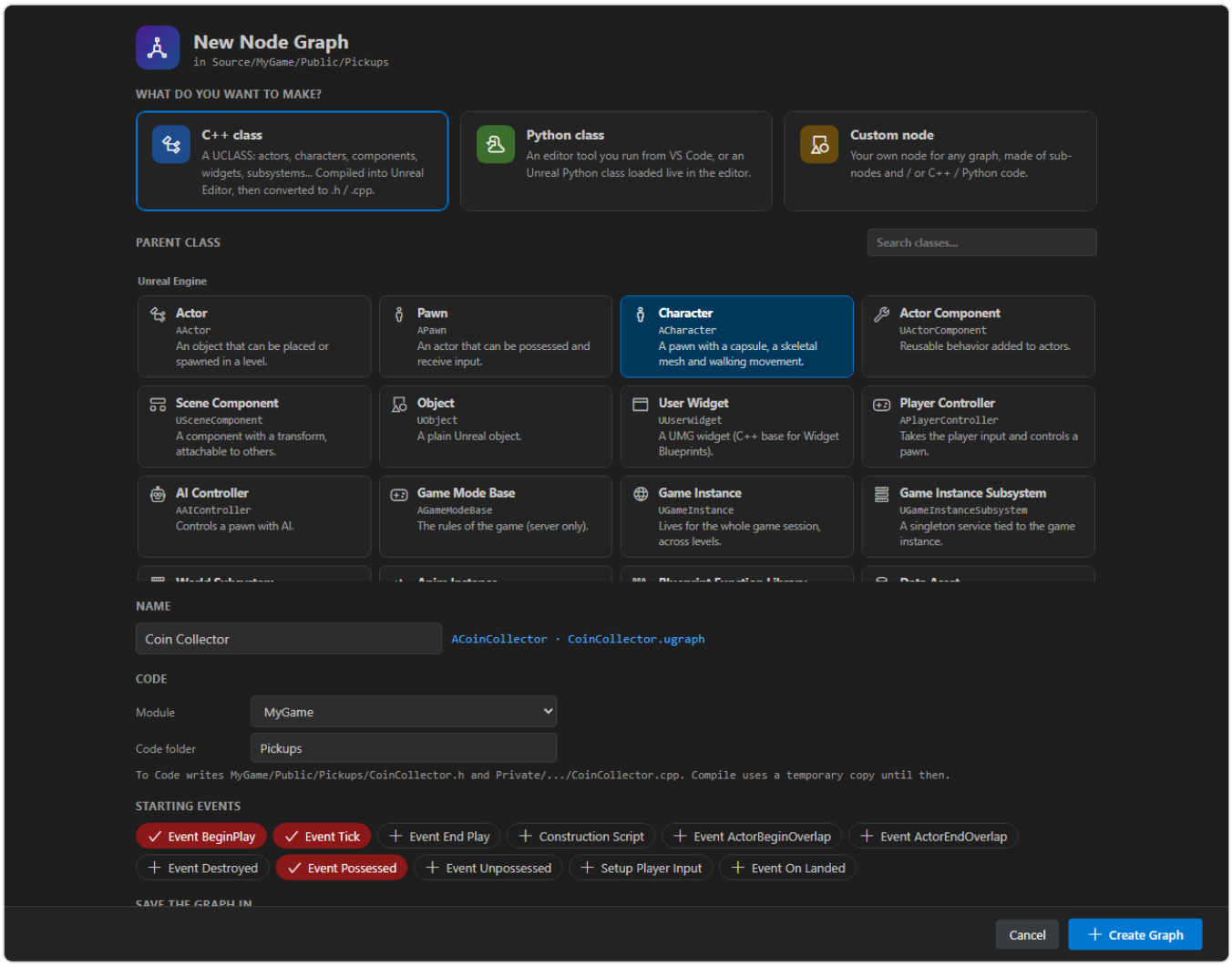


A C++ actor made with nodes in VS Code, just compiled: components, a dispatcher, a component overlap event, Do Once and a timer.

Create a graph

Right-click a folder of your source code (Explorer, Solution Explorer or Favorites) › **New Node Graph (C++ / Python)...**, click **Node Graph** in the UStudio panel, or run **UStudio: New Node Graph (C++ / Python)...**. A wizard opens:

- Cards for what to make: a C++ class, a Python class or a custom node.
- The parent class as cards with a search (engine classes and the project's own).
- The name, with a preview of the class and file names (`ACoinCollector · CoinCollector.ugraph`).
- The module and code folder, taken from the folder you clicked (`Source/MyGame/Public/Pickups` gives the module `MyGame` and the folder `Pickups`).
- The events the graph starts with, and where it is saved.



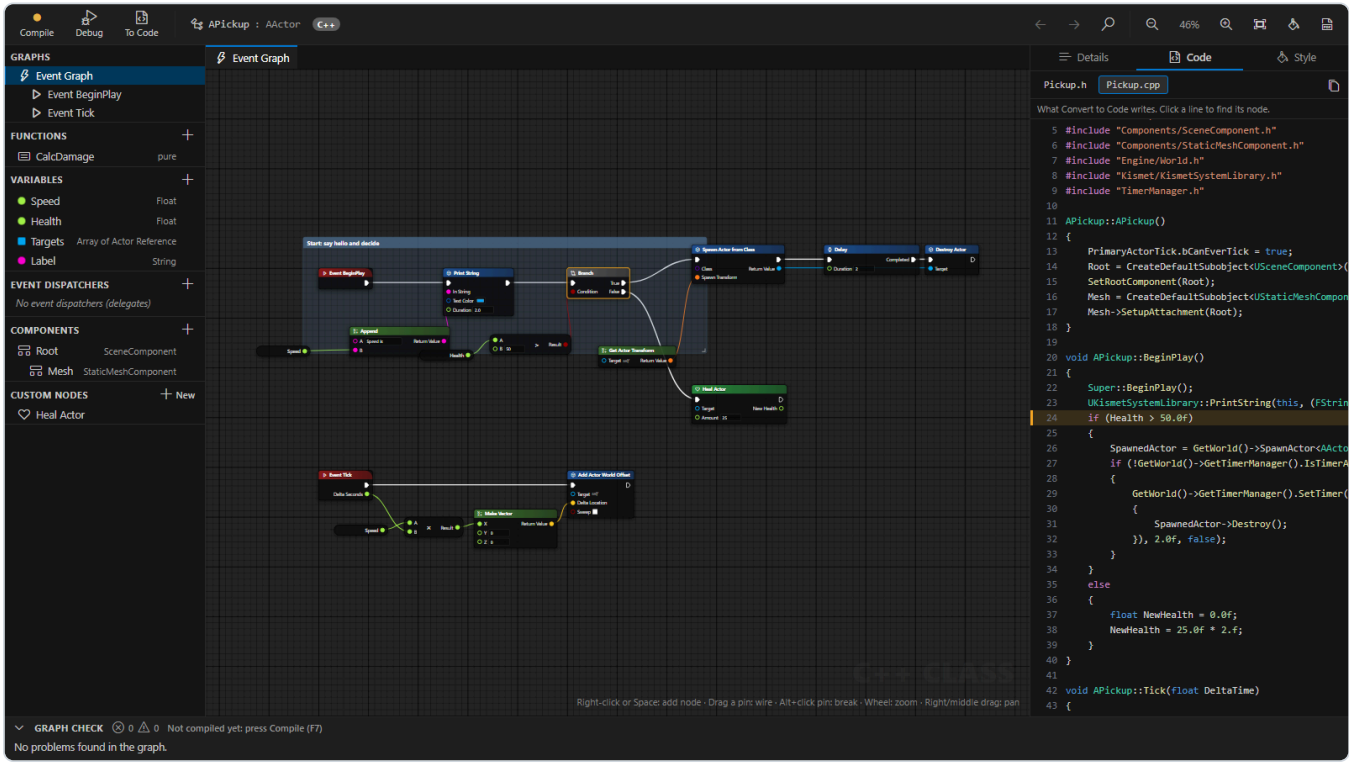
The New Node Graph wizard, opened from Source/MyGame/Public/Pickups: a Character with BeginPlay, Tick and Possessed.

Parent classes:

Language	Parent classes
C++	Actor, Pawn, Character, Actor / Scene Component, Object, User Widget, Player / AI Controller, Game Mode Base, Game Instance, Game Instance / World Subsystem, Anim Instance, Blueprint Function Library, Data Asset, or any class of the project.
Python	A Python class with a **Run Script** event (an editor tool), `unreal.Object`, `unreal.Actor`, `unreal.ActorComponent` or `unreal.BlueprintFunctionLibrary`.

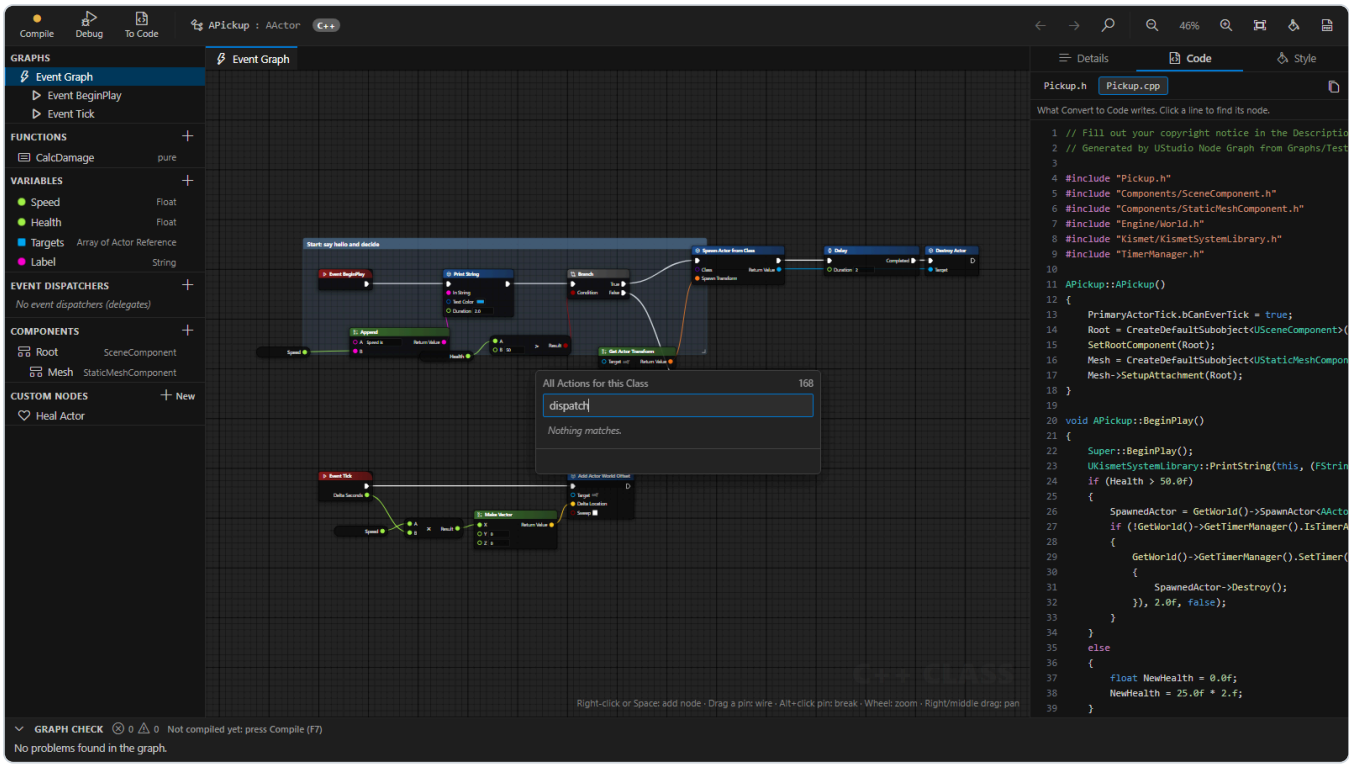
The graph is saved as `Graphs/<Name>.ugraph` (setting `unrealStudio.nodeGraph.folder`). It is a JSON file of the project: `Ctrl+S`, undo / redo and source control work as for any file.

The editor



The Branch node is selected: its lines are highlighted in the generated .cpp of the Code tab.

- **Add nodes:** right-click the canvas or press `Space` for the searchable list. Drag a wire from a pin and drop it on empty space: the list only shows nodes that can connect, and the new node comes wired.
- **Wires:** drag from a pin to another. `Alt+click` a pin or a wire breaks it; double-click a wire adds a reroute; right-click a wire to delete it or give it its own color.
- **Pin types** are checked like in Blueprints. Two pins connect only when they have the same type, when Blueprint converts between them automatically (numbers, values to string, string / name / text), or when the object is of the pin's class or a child of it. A parent class needs a **Cast**, and the message says so. An Actor can't go into a Branch.
- **Arrange like in Unreal:** `Q` straightens the wires of the selection, `Shift+W` / `S` / `A` / `D` align top / bottom / left / right, `Alt+Shift+W` / `Alt+Shift+S` align middle / center, `L` arranges the selection or the whole graph, arrow keys nudge (`Shift` : four steps). Also under right-click › **Alignment** and **Arrange** on the toolbar.
- **Select and move:** click, `Ctrl+click` , or drag a box on the canvas. `Delete` , `Ctrl+D` (duplicate), `Ctrl+C` / `Ctrl+V` , `C` (comment around the selection), `F` (frame the selection), `Home` (fit everything). The wheel zooms; the right or middle button pans.
- **My Class** (left): graphs and events, functions, variables, event dispatchers, components and your custom nodes. Drag a variable to the canvas for Get / Set (`Ctrl` = Get, `Alt` = Set), a function to call it, a dispatcher for Call / Bind / Unbind.
- **Details** (right): the selected node, member or the class settings (name, parent, module, folder). Values of unconnected pins can be typed on the node itself.
- **Code** (right): what Convert to Code writes, updated as you edit. Click a line to find the node that produced it. **Compiler Results** (bottom): the graph checks and, after compiling, the compiler's messages.



The node list, filtered as you type.

Nodes

Group	Nodes
Events	The parent class's events (BeginPlay, Tick, End Play, Construction Script, overlaps, Possessed, Setup Player Input, widget Construct / Tick, subsystem Initialize...), custom events (with replication) and component events.
Flow control	Branch, Sequence, For Loop, For Each Loop, While Loop, Switch on Int, Do Once, Do N, Gate, FlipFlop, Cast To, Return Node.
Async	Delay, Retriggerable Delay, Set Timer (named, looping), Clear Timer, Is Timer Active, Async Task (background thread, then Completed on the game thread), Async Load Asset. C++.
Event dispatchers	Call, Bind Event to, Unbind Event from, Unbind all Events from, Is Bound. C++.
Values	Variables, components, self, math (integer, float, vector, boolean, comparisons, clamp, lerp, interpolation, random), vectors, rotators, transforms, strings, arrays.
Game and actors	Print String, logs, Spawn Actor from Class (its output has the class you pick), Get Player Pawn / Character / Controller / State / Camera Manager, Get Game Instance / Game Mode / Game State, Get Controlled Pawn, Get Controller, Get HUD, Possess, Get All Actors Of Class, actor and component transforms, visibility, Destroy Actor, console commands.
Subsystems	Get Game Instance Subsystem, Get World Subsystem, Get Local Player Subsystem, Get Engine Subsystem (and Get Editor Subsystem in Python): pick the class on the pin, yours or the engine's, and the output has that type.
Spawning	Play Sound at Location / 2D, Spawn Sound at Location / Attached, Spawn System at Location / Attached (Niagara), Spawn Emitter at Location (Cascade), Spawn Decal at Location, Create Widget, Add to Viewport, Remove from Parent, Construct Object from Class, Add Component by Class. Asset pins take a path: `/Game/Audio/Hit`.
Other classes	Functions, custom events, variables and event dispatchers of another graph's class or of a C++ class of the project or its plugins, with a **Target** pin. Static library functions need no target. See below.
Editor scripting	Python: selected and level actors, spawn in the level, labels, selected assets, load / list / save assets, Blueprint classes, message dialogs.
Custom code	**Custom Code** (statements) and **Custom Expression** with your own inputs: any C++ or Python.

A pin connected to another type converts automatically when Blueprints would (integer to float, numbers and names to string...). A node that can't be generated (missing variable, a C++-only node in Python) shows a red mark with the reason.

Event dispatchers and delegates

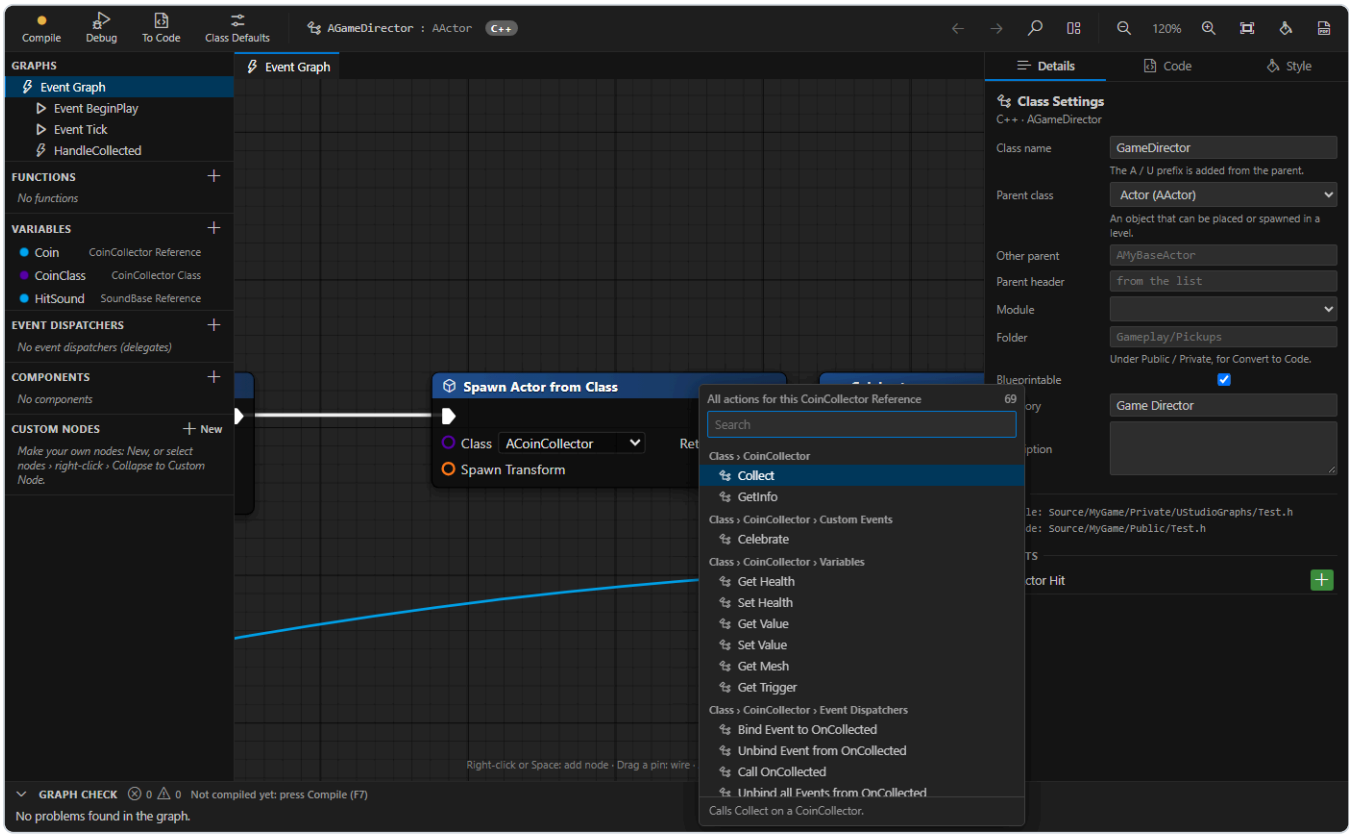
Add an **Event Dispatcher** in My Class and choose its kind:

Kind	Generated C++	Use
Dynamic Multicast	`DECLARE_DYNAMIC_MULTICAST_DELEGATE_*` and a `BlueprintAssignable` property	The Blueprint event dispatcher: several listeners, Blueprints can bind it.
Multicast	`DECLARE_MULTICAST_DELEGATE_*`	Several listeners, C++ only, faster.
Dynamic	`DECLARE_DYNAMIC_DELEGATE_*`	One listener (a UFUNCTION).
Single	`DECLARE_DELEGATE_*`	One listener, C++ only.

Bind Event to needs a custom event with the same inputs: pick it in Details, or click **Create a matching event**. Select a component with collision (box, sphere, capsule, mesh): its **Events** appear in Details with **+** buttons, as in Unreal. The event node gets the delegate's outputs, and UStudio writes a `UFUNCTION` handler with the engine signature, bound in `BeginPlay` with `AddUniqueDynamic`.

Using other classes

Graphs use each other like Blueprints. The class of every `.ugraph` in the project works right away, without converting it: as a variable or parameter type, in Cast To, on class pins (Spawn Actor from Class, subsystems) and as the parent of another graph. The project's C++ classes, including the plugins in its `Plugins` folder, work the same way. Their public `UFUNCTION`s and `UPROPERTY`s are read from their headers.

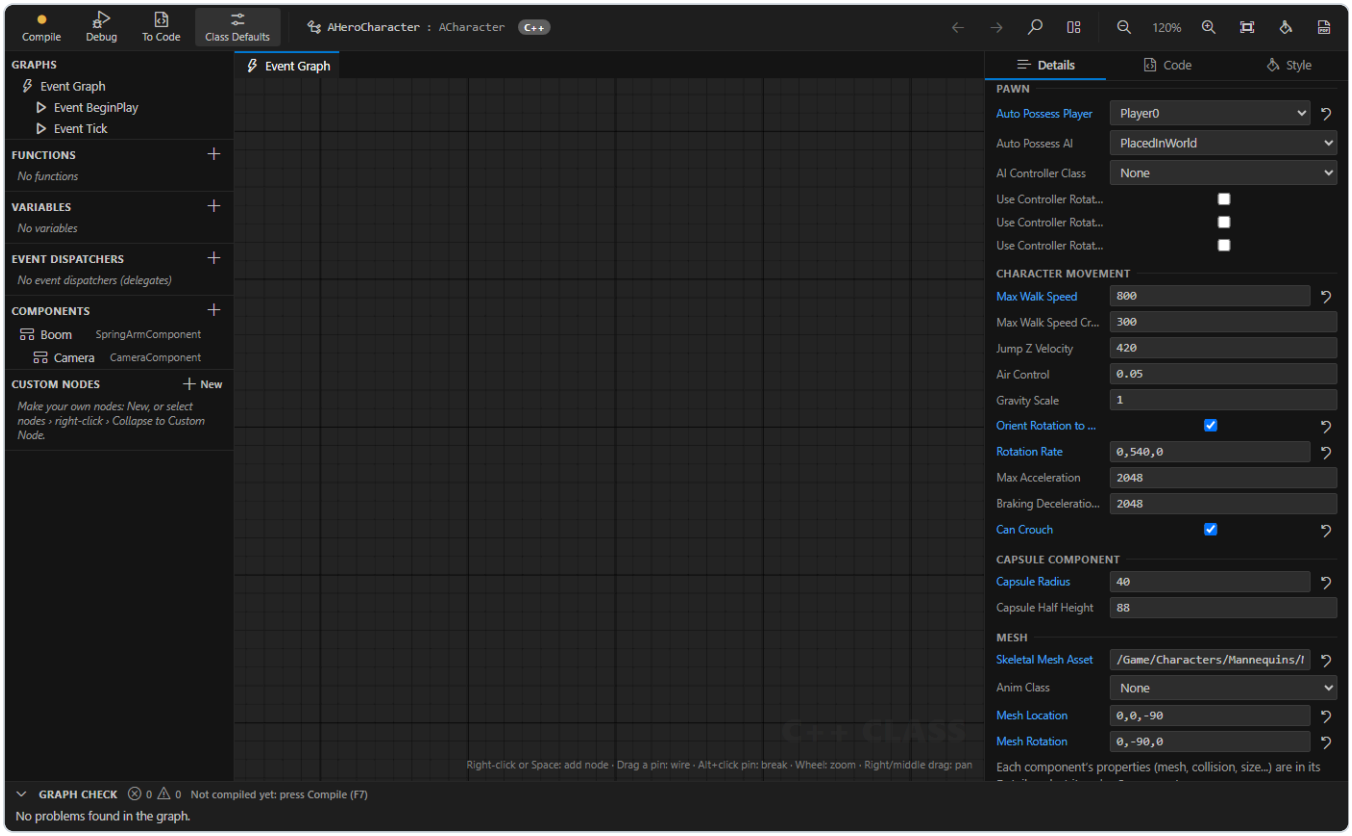


A wire dragged from a CoinCollector pin: that class's functions, custom events, variables and event dispatchers come first.

- **Members.** Drag a wire from an object pin and the list starts with its class's members, from the class and its parents. They give nodes with a **Target** pin: call a function, Get / Set a variable, Bind / Unbind / Call an event dispatcher. With Target unconnected the node acts on self, when this class is one of those. A `WorldContextObject` left unconnected is self, as in Blueprints.
- **Libraries.** Static functions of function libraries (yours or a plugin's) are always in the list.
- **Renames.** Renaming a function or variable in its graph doesn't break the nodes that use it. Deleting it shows the error on them. **F12** opens the other graph at that member, or the C++ header.
- **Compiling.** Compiling a graph first writes the compiled copy of the graph classes it uses (and of the ones those use). Compiling or converting a class rewrites the code of the graphs that use it, so they follow renames and include its final header.
- **Modules.** A class from another module (a plugin, for example) adds that module to the Build.cs. If that class is a graph that is only compiled, convert it to code first: its compiled copy is private to its module.

Class Defaults

Class Defaults on the toolbar shows, like in Blueprints, the default value of every variable and the properties the class inherits. Asset variables take a path (`/Game/Audio/Hit`) and class variables a class. Inherited properties are written in the constructor only when they differ from the engine's value. Changed values are highlighted, with an arrow to reset them.

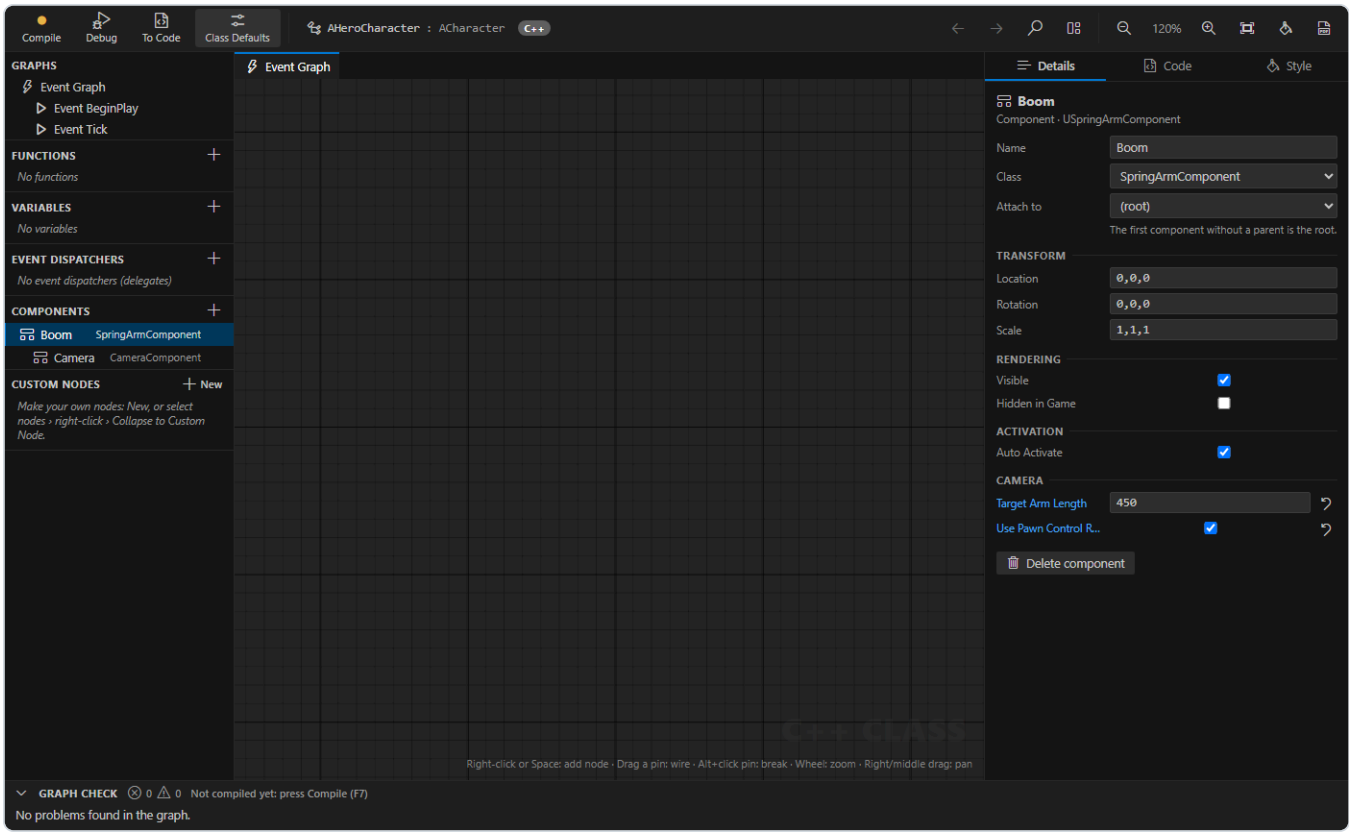


Class Defaults of a character: Auto Possess Player is Player0 (changed, with its reset arrow).

Parent	Properties
Every actor	Replicates, Replicate Movement, Always Relevant, Only Relevant to Owner, Net Load on Client, Net Update Frequency, Net Priority, Net Cull Distance, Net Dormancy; Can Ever Tick, Start with Tick Enabled, Tick Interval, Tick Group; Initial Life Span, Hidden in Game, Can be Damaged, Spawn Collision Handling.
Pawn, Character	Auto Possess Player / AI, AI Controller Class, Use Controller Rotation Yaw / Pitch / Roll.
Character	Max Walk Speed, crouched speed, Jump Z Velocity, Air Control, Gravity Scale, Orient Rotation to Movement, Rotation Rate, acceleration and braking, Can Crouch; capsule size, Skeletal Mesh Asset, Anim Class, mesh location and rotation.

Game Mode	Default Pawn, Player Controller, Game State, Player State, HUD and Spectator classes (yours, from a graph or C++), Start Players as Spectators, Use Seamless Travel.
Player Controller	Show Mouse Cursor, Enable Click Events, Enable Mouse Over Events.
Components	Replicates, Auto Activate, Tick Interval.

Each component in **Components** shows its own properties in Details: location, rotation and scale; visibility; Static Mesh, Skeletal Mesh and Material assets; Collision Presets, Generate Overlap Events, Simulate Physics; box, sphere and capsule size; text, light, sound, spring arm, camera, projectile and rotating movement settings.



A spring arm component: Target Arm Length and Use Pawn Control Rotation changed.

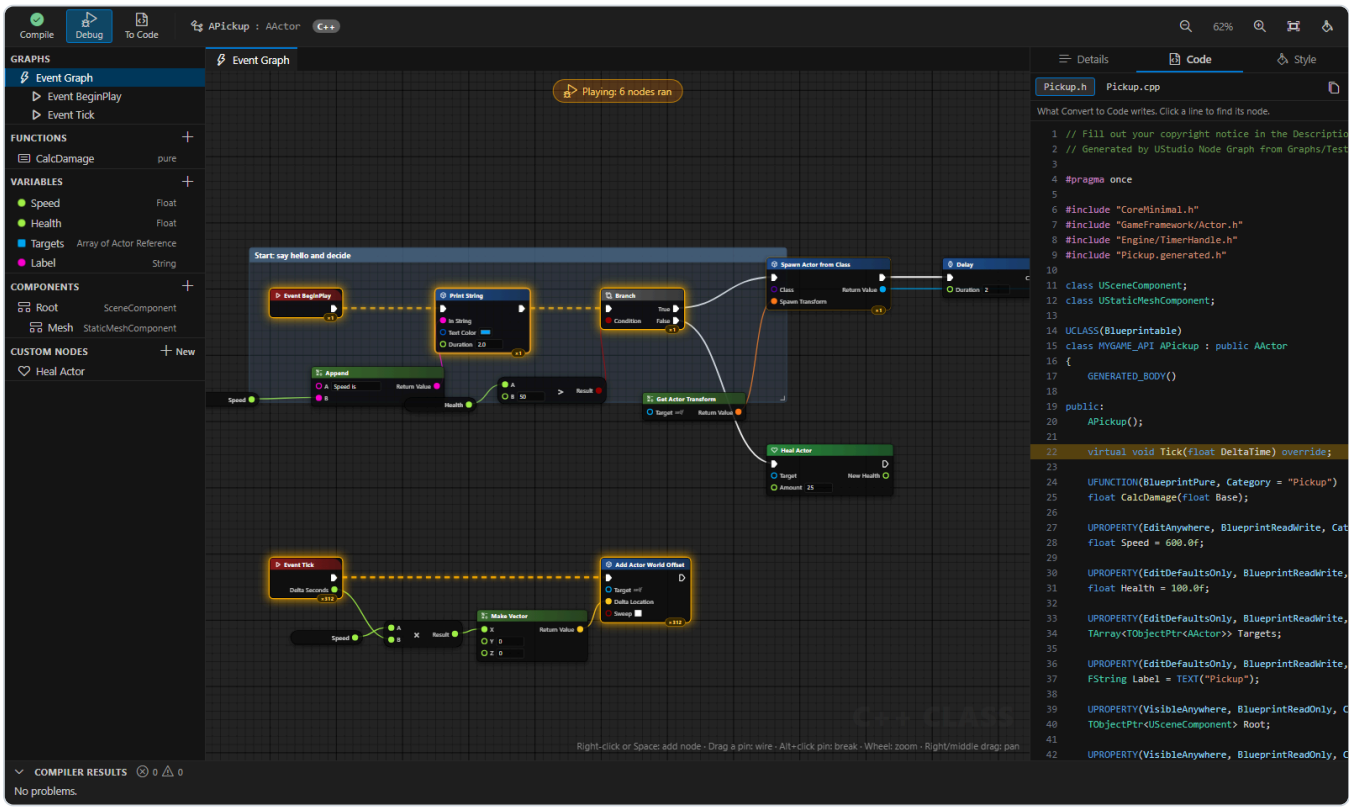
Compile PRO

Compile (`F7` , or the toolbar) builds a temporary copy of the class and loads it into the running editor, so it can be placed, spawned or made into a Blueprint right away:

	C++	Python
Where the copy goes	`Source/<Module>/Private/UStudioGraphs/<Class>.h / .cpp`	`Intermediate/UStudio/NodeGraphs/ustudio_graph_<name>.py`
Unreal Editor open	Live Coding through the Live Bridge plugin. A new class needs Live Coding reinstancing (Editor Preferences › Live Coding).	Loaded (or reloaded) through Python Remote Execution. **Run** calls the Run Script event.
Unreal Editor closed	A build of the editor target.	A syntax check with the engine's Python.

The toolbar button shows the state: yellow (changed since the last compile), green (compiled), yellow warning or red error. Compiler errors and warnings are listed in **Compiler Results** on the node that produced the line; click one to go to it.

Live debugging PRO



Debug on, playing in the editor: the nodes that ran are lit, the exec wires flow and Tick shows 312 runs.

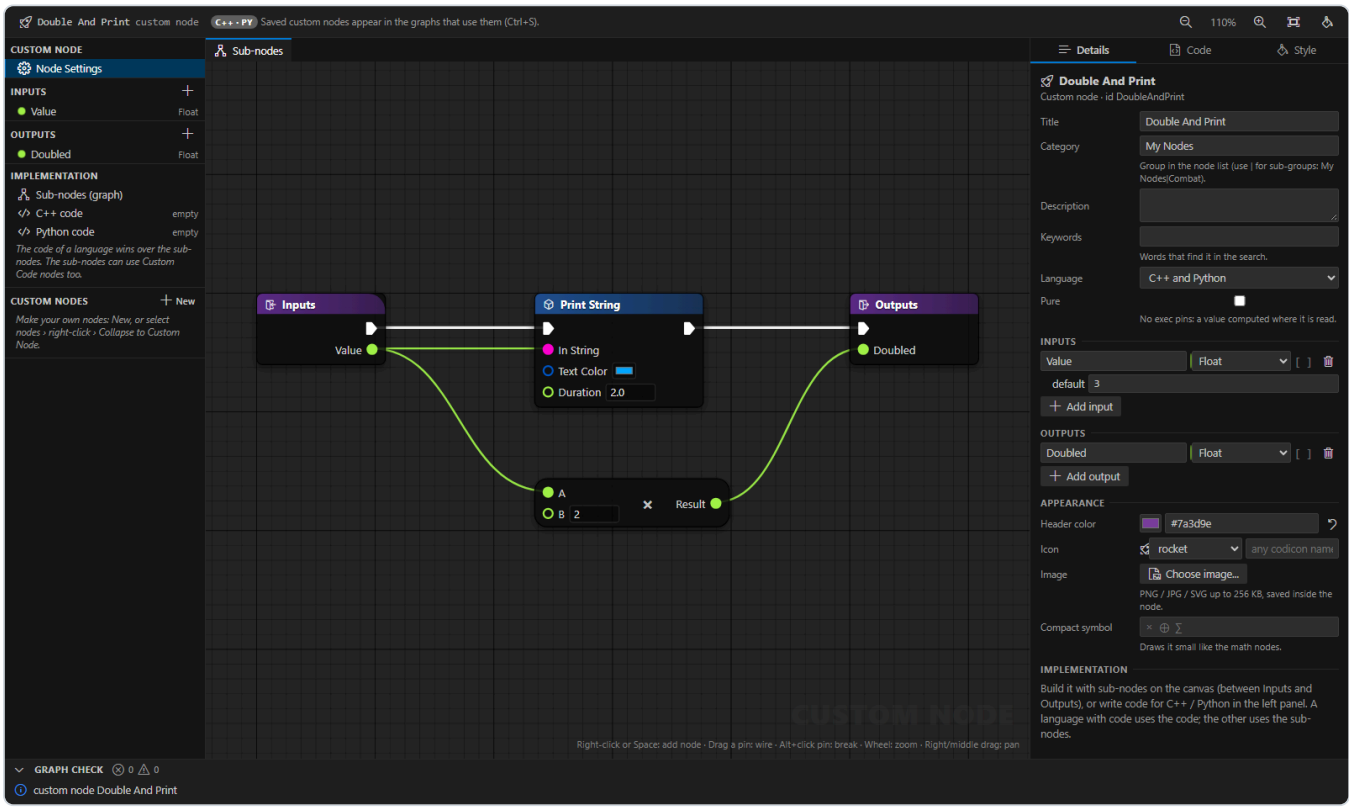
Click **Debug** and compile. While Unreal Editor runs the class (Play in Editor, or the editor itself for tools), the nodes that run light up, the exec wires between them flow, and each node shows how many times it ran. The Code tab highlights the running lines too. C++ uses the Live Bridge plugin's `USTUDIO_TRACE()` (the plugin and the `UStudioTrace` module are added the first time; then one build with Unreal Editor closed). Python counts the runs in the module.

Convert to Code PRO

To Code writes the final class as regular source: `Public/<Folder>/<Class>.h` and `Private/<Folder>/<Class>.cpp` in the module (or `Content/Python/<name>.py`), adds the modules it needs to the `Build.cs` and removes the temporary copy. The class keeps its name and module, so Blueprints made from the compiled class keep working.

The graph remembers the converted files: compiling it again updates them while they are unchanged (UStudio stores a checksum in the first comment line). If you edited them, UStudio asks before replacing them. Delete the graph, or the comment line, to keep the code as your own.

Custom nodes



A custom node made of sub-nodes: Inputs, Print String, a multiply, Outputs.

Custom nodes are `.unode` files, in the project's `.ustudio/nodes` (shared with the team) or in `~/.ustudio/nodes` (all your projects). Create one with **UStudio: New Custom Node**, **New** in the Custom Nodes list, or select nodes › right-click › **Collapse to Custom Node**. It opens in the same editor:

- **Sub-nodes:** wire nodes between the **Inputs** and **Outputs** nodes. Where the node is used, its sub-nodes are expanded in place, like a Blueprint macro.
- **Code:** C++ and / or Python statements (or one expression for a pure node). `{Input}` reads an input, `{$Output} = ...` sets an output, `{Input!}` forces a typed value (a `FString`, or a typed null for overloaded functions). Includes, Build.cs modules and Python imports can be listed.
- **Both:** a language with code uses its code; the other uses the sub-nodes.
- **Look:** title, category (`My Nodes|Combat`), description, keywords, header color, icon, an image (up to 256 KB, saved in the file) or a compact symbol.

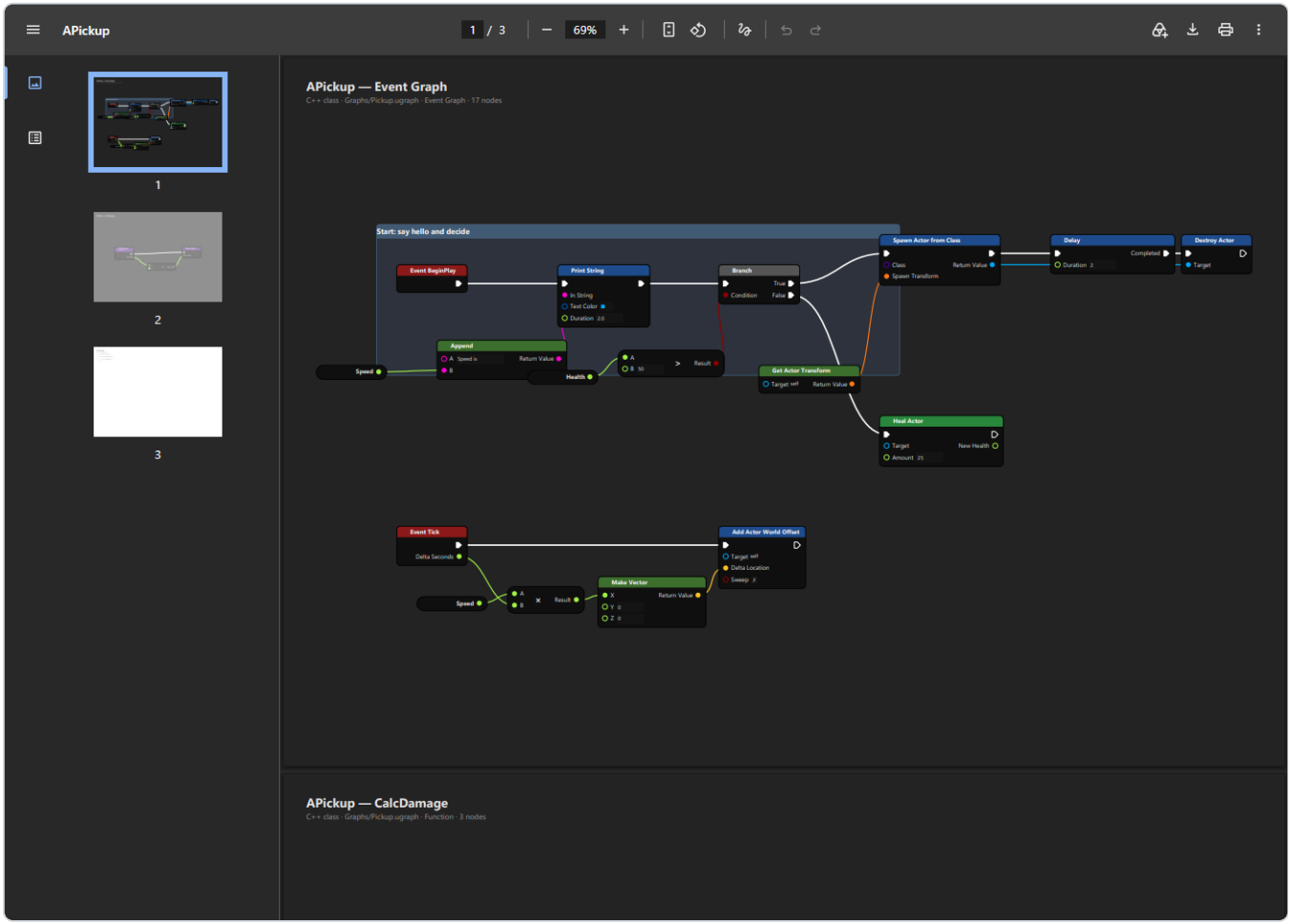
Double-click a custom node in a graph (or `F12`) to open it. Saved changes reach every open graph at once.

Navigation

Action	How
Back / Forward	Toolbar arrows, `Alt+←` / `Alt+→`, the mouse's back / forward buttons.
Find	`Ctrl+F` : nodes (titles, values, notes, code), variables, functions, components and dispatchers of every graph. `Enter` goes to the next result.
Go to Definition	`F12` or right-click a node: a function call opens its graph, a variable or dispatcher its details, a custom node its file, a C++ call the engine or project header at the function.
Show in Code	Right-click a node: its lines in the compiled or converted file, else in the Code tab.

From the code	Generated files have an **Open Node Graph** CodeLens and a **Show in Node Graph** one above each event and function; right-click › **Show in Node Graph** finds the node of any line.

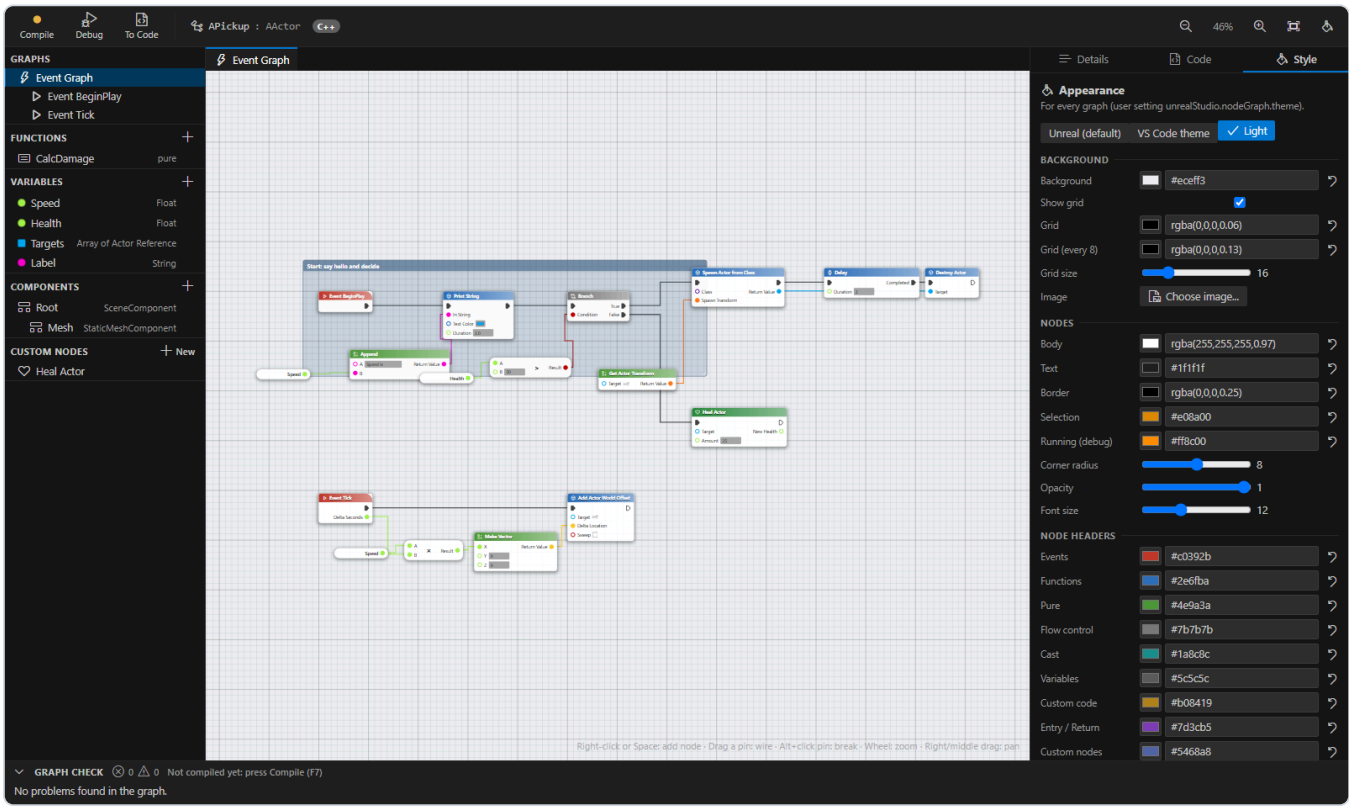
Export to PDF PRO



The exported PDF: one vector page per graph, bookmarks, then the generated code.

The PDF button of the toolbar, **UStudio: Export Node Graph to PDF** or right-click a `.ugraph` / `.unode`. Each graph gets an A3 landscape page drawn as on screen (nodes, wires, pins, values and comments in vector, text selectable), followed by the generated code.

Appearance



The Style tab with the Light preset and right-angled wires.

The **Style** tab changes the look of every graph (user setting `unrealStudio.nodeGraph.theme`). Unreal is the default; VS Code (follows your color theme) and Light are presets. Everything can be changed: background color, grid colors and size, a background image (cover, tile or center, with its opacity), node body, text and border colors, corner radius, opacity and font size, the selection and debug colors, header colors for each kind of node, pin colors for each type, and the wires (curved, straight or right-angled, thickness, exec color, by pin type or one color). **Reset to Unreal look** brings the defaults back.

AI agents PRO

The MCP server has the whole feature: `nodegraph_catalog` lists the nodes with their pins for a language and parent class, `nodegraph_create` and `nodegraph_edit` build graphs with operations (add variables, functions, components, dispatchers and nodes, wire pins by name; new nodes are laid out automatically), `nodegraph_compile` returns the compiler messages with their nodes, `nodegraph_get_execution` reads which nodes ran, and `nodegraph_convert_to_code`, `nodegraph_export_pdf`, `custom_node_create`, `nodegraph_run`, `nodegraph_read`, `nodegraph_generate_code`, `nodegraph_set_theme` complete it.

Good to know

Situation	What to do
The compiled class doesn't appear in Unreal Editor	Live Coding adds new classes only with reinstancing on (Editor Preferences › Live Coding › Enable Reinstancing). Otherwise close Unreal Editor and compile: UStudio builds the project.
Unreal Editor is open without the Live Bridge plugin	UStudio writes the class files; press `Ctrl+Alt+F11` in Unreal Editor (Live Coding), or let UStudio install the plugin from Compiler Results.
A Python class disappears after restarting Unreal Editor	Python classes live while the editor runs: convert the graph to code and import the module from `Content/Python/init_unreal.py`.
Delegates, timers or async nodes in a Python graph	They are generated for C++ only (marked in red in Python graphs).

A value used in another event, or after a Delay	Works as in Blueprints: UStudio keeps it in a member of the class.

The generated C++ of every node kind (components, replication, RPCs, delays, casts, loops, custom nodes, the four kinds of delegate, component events, async tasks, timers, async loads, Do Once / Gate / FlipFlop / Do N and debug builds) was compiled with Unreal Engine 5.7.

Converting Blueprints to C++ PRO EXPERIMENTAL

New in 1.17.0. Turn a Blueprint of an actor, component or object into a real C++ class, and make every asset that used the Blueprint use the class instead: actors placed in levels, child Blueprints, variables, Cast and Spawn nodes, class defaults and soft references.

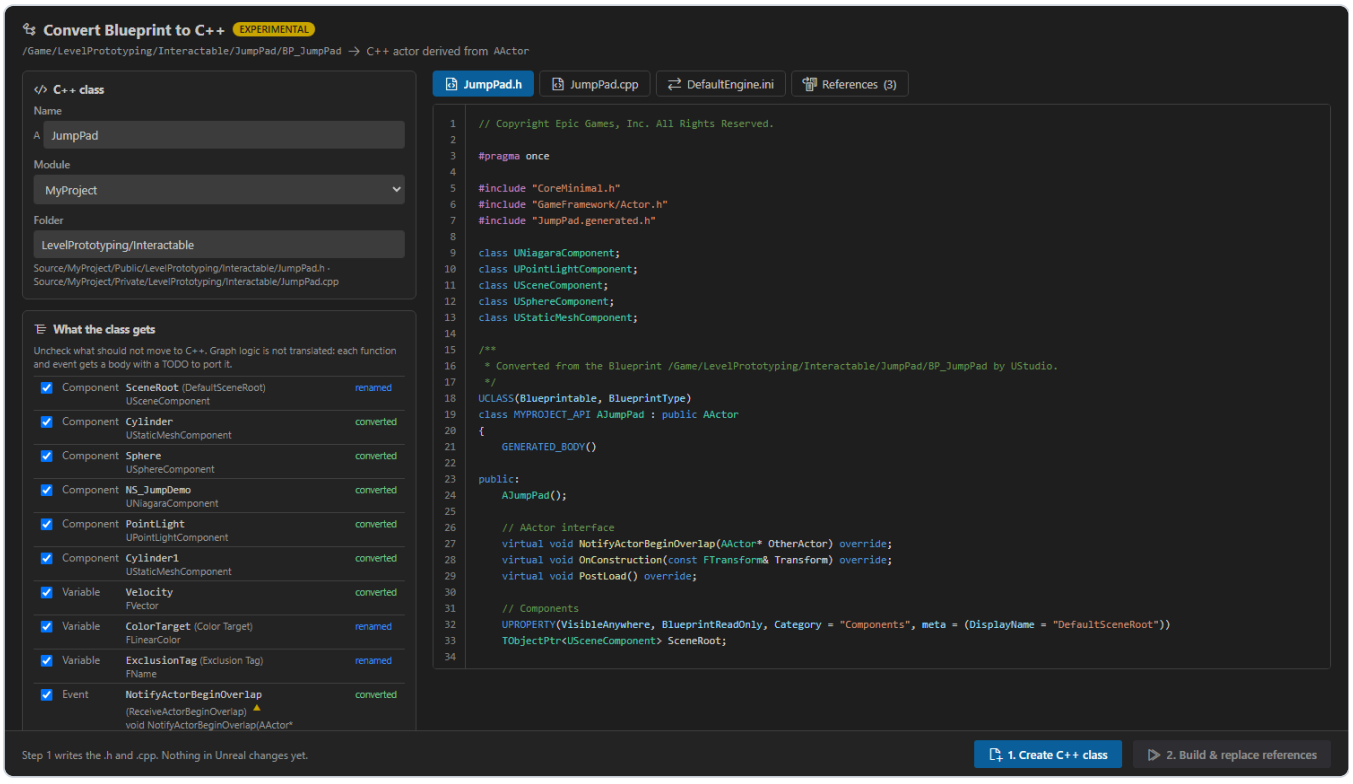
> **Experimental.** The conversion rewrites assets of your project. Commit, shelve or back up the project before step 2. UStudio keeps a copy of the converted Blueprint in `Saved/UStudio/BlueprintBackups`, and source control lets you revert the re-saved assets.

Which Blueprints can be converted

Converted	Not converted
Blueprint classes of actors (Actor, Pawn, Character, GameMode, PlayerController, HUD...), actor components and objects, whose parent is a C++ class (engine or project).	Widget Blueprints, Animation Blueprints, levels (Level Blueprints), materials, data tables, Blueprint Interfaces, Function / Macro Libraries, Blueprint structs and enums. A Blueprint whose parent is another Blueprint: convert the parent first.

Step 1: create the C++ class

Right-click a Blueprint `.uasset` in the Explorer, the Solution Explorer or Favorites › **Convert to C++ Class (Experimental)...**, click the button in the title bar of its C++ view, or run **UStudio: Convert Blueprint to C++ Class** from the Command Palette.



The wizard on BP_JumpPad (UE 5.7 sample): what the class gets, and the generated header.

- **Name, Module, Folder.** The name defaults to the Blueprint's without its prefix (`BP_Enemy` → `AEnemy`), or with `Converted` when that class already exists. The folder mirrors the Blueprint's folder under `Content` ; modules with `Public` / `Private` folders get the header in `Public` .

- **What the class gets.** Every component, variable, dispatcher, function and event, with its C++ name and type and a status: converted, renamed (a name with spaces became an identifier; a redirect keeps references working), partial (for example a variable typed as another Blueprint uses that Blueprint's C++ parent) or not converted (with the reason). Uncheck what should stay out. A yellow marker flags functions and events with Blueprint logic.
- **Preview.** The `.h`, the `.cpp`, the redirects for `DefaultEngine.ini` and the assets that reference the Blueprint.
- **Notes.** What could not be carried over (a private engine property without setter, a default that would be a circular dependency...), and the modules that will be added to the `Build.cs`.

1. Create C++ class writes the two files and adds the modules the code uses to the module's `Build.cs`. Nothing changes in Unreal yet: review the code, build it, adjust it.

What becomes what

In the Blueprint	In the C++ class
Components and their hierarchy	`CreateDefaultSubobject` with the same names, `SetupAttachment` (with sockets), root component, and their settings: meshes, materials, anim class, collision profile, physics, transforms, Niagara systems, child actor classes, lights, sounds...
Inherited components the Blueprint changed	Set in the constructor through their accessor: `GetCharacterMovement()->MaxWalkSpeed`, `GetMesh()`, `GetCapsuleComponent()`, or `GetDefaultSubobjectByName`.
Variables	`UPROPERTY` with the same specifiers (`EditAnywhere`, `BlueprintReadWrite`, `Replicated` / `ReplicatedUsing`, `SaveGame`, `ExposeOnSpawn`, clamps...), category, tooltip and default value. Blueprint floats are `float`; assets and classes load through `ConstructorHelpers`.
Class Defaults (inherited settings)	Constructor statements: `bReplicates`, `AutoPossessPlayer`, `PrimaryActorTick`... Private engine members go through their setter.
Event dispatchers	`DECLARE_DYNAMIC_MULTICAST_DELEGATE_*` inside the class and a `BlueprintAssignable` property.
Functions, custom events	`UFUNCTION(BlueprintCallable / BlueprintPure)` with a body to port. Server / Client / Multicast events become RPCs with `_Implementation`.
Event BeginPlay, Tick, EndPlay, Hit, Overlap, Construction Script...	Overrides of the native functions (`BeginPlay`, `Tick`, `NotifyHit`, `OnConstruction`...) that call `Super`.
Interface events, BlueprintNativeEvents	`_Implementation` overrides (C++ interfaces only).
Component events (On Component Begin Overlap...)	A `UFUNCTION()` handler bound with `AddDynamic` in `BeginPlay`.
Replicated variables	`GetLifetimeReplicatedProps` with `DOREPLIFETIME`.

Graph logic is **not** translated: each function and event body has a `TODO (UStudio)` comment where its Blueprint logic goes. Timelines, Blueprint structs and enums and Blueprint interfaces are listed as not converted.

Example

A Blueprint actor with a root, a mesh, a sphere trigger attached to the mesh and a few variables: one named `Max Speed`, one typed as another Blueprint (`HolderRef`, now its C++ parent `AActor`) and a class of itself (`SpawnClass`):

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Thing.generated.h"

class USceneComponent;
class USphereComponent;
```

```h
class UStaticMeshComponent;

/**
 * Converted from the Blueprint /Game/Conv/BP_Thing by UStudio.
 */
UCLASS(Blueprintable, BlueprintType)
class CONVTEST_API AThing : public AActor
{
	GENERATED_BODY()

public:
	AThing();

	// AActor interface
	virtual void PostLoad() override;

	// Components
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components", meta = (DisplayName = "DefaultSceneRoot"))
	TObjectPtr<USceneComponent> SceneRoot;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
	TObjectPtr<UStaticMeshComponent> Mesh;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Components")
	TObjectPtr<USphereComponent> Trigger;

	// Variables
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Default")
	float Health = 50.0f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Default")
	bool bAlive = true;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Default", meta = (MultiLine = true))
	FString Label = TEXT("Hello");

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Default", meta = (DisplayName = "Max Speed"))
	int32 MaxSpeed = 3;

	UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category = "Default")
	TObjectPtr<AActor> HolderRef;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Default")
	TSubclassOf<AThing> SpawnClass;
};
```

```cpp
#include "Thing.h"
#include "Components/SceneComponent.h"
#include "Components/SphereComponent.h"
#include "Components/StaticMeshComponent.h"
#include "Engine/StaticMesh.h"
#include "UObject/ConstructorHelpers.h"

AThing::AThing()
{
	SceneRoot = CreateDefaultSubobject<USceneComponent>(TEXT("DefaultSceneRoot"));
	SetRootComponent(SceneRoot);

	Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
	Mesh->SetupAttachment(SceneRoot);
	static ConstructorHelpers::FObjectFinder<UStaticMesh> MeshStaticMesh(TEXT("/Engine/BasicShapes/Cube.Cube"));
	Mesh->SetStaticMesh(MeshStaticMesh.Object);
	Mesh->SetRelativeLocation(FVector(0.0, 0.0, 50.0));

	Trigger = CreateDefaultSubobject<USphereComponent>(TEXT("Trigger"));
	Trigger->SetupAttachment(Mesh);
	Trigger->InitSphereRadius(120.0f);
}

void AThing::PostLoad()
{
	Super::PostLoad();

	// Converted from BP_Thing by UStudio: actors saved with the Blueprint mark these components as
```

```
        // made by its construction script, which would destroy them on load. They are this class's own components now.
        // This block can be removed once every level that places this class has been saved again.
        for (UActorComponent* Component : GetComponents())
        {
                if (Component && Component->CreationMethod == EComponentCreationMethod::SimpleConstructionScript && Component-
>HasAnyFlags(RF_DefaultSubObject))
                {
                        Component->CreationMethod = EComponentCreationMethod::Native;
                        BlueprintCreatedComponents.Remove(Component);
                }
        }
}
```

The `PostLoad` block lets actors saved with the Blueprint keep their components: levels saved before the conversion mark them as made by the Blueprint's construction script, which would destroy them on load. Once every level that places the class has been saved again (step 2 does it), the block can be removed.

Step 2: replace the Blueprint

Close Unreal Editor, then click **2. Build & replace references** (or run **UStudio: Finish Blueprint to C++ Conversion** later). After a confirmation, UStudio:

1. **Builds** the project (Development Editor). If the build fails, nothing else happens: fix the errors and run the step again.
2. **Moves the Blueprints that derive from it** onto the new class with a headless Unreal (no redirect exists yet): children are reparented, deeper ones recompiled, and what they changed on inherited components (a different mesh, scale, collision profile...) and their own components attached to them are set again.
3. **Adds the redirects** to `Config/DefaultEngine.ini`: the class (with `OverrideClassName`, which lets a Blueprint class become a C++ class), its default object, and the renamed variables, functions and parameters.
4. **Re-saves every asset that referenced the Blueprint** with a second headless Unreal, so they point at the C++ class themselves.
5. **Deletes the Blueprint**, after copying it to `Saved/UStudio/BlueprintBackups/<date>`.

```
[CoreRedirects]
; BP_Thing was converted to C++ by UStudio: references to the Blueprint load the C++ class
+ClassRedirects=
(OldName="/Game/Conv/BP_Thing.BP_Thing_C",NewName="/Script/ConvTest.Thing",OverrideClassName="/Script/CoreUObject.Class")
+ObjectRedirects=(OldName="/Game/Conv/BP_Thing.Default__BP_Thing_C",NewName="/Script/ConvTest.Default__Thing")
+PropertyRedirects=(OldName="Thing.DefaultSceneRoot",NewName="SceneRoot")
+PropertyRedirects=(OldName="Thing.Max Speed",NewName="MaxSpeed")
```

Keep the redirects: they also cover assets that were not re-saved (another branch, a teammate's unsaved work). Under Perforce, the assets to re-save and `DefaultEngine.ini` are checked out and the Blueprint is marked for delete. If something fails before the references are updated, UStudio removes the redirects again and the Blueprint stays as it was.

Child Blueprints and circular references

- A child Blueprint keeps working: it now derives from the C++ class, keeps its variables' values, the changes it made to inherited components and its own components.
- Circular references (`BP_A` has a variable of type `BP_B` and `BP_B` one of type `BP_A`) are fine: in C++ a variable typed as another Blueprint uses its C++ parent, so the class never depends on a Blueprint.
- A default value that points to a Blueprint deriving from the converted one (`SpawnClass = BP_EnemyVariant`) is not set from the constructor: loading it there would need the class being built. Set it in a Blueprint child, or load it at runtime with a `TSoftClassPtr`.

After the conversion

- Port the logic into the `TODO`s. Other Blueprints' calls, variable reads, casts and bindings now go to the C++ members.
- Variables that were typed as another Blueprint are typed as its C++ parent: cast them where you need that Blueprint's members, or retype them once that Blueprint is converted too.

- If the class should keep editable assets (meshes, sounds) outside C++, create a Blueprint child of the new class and set them there.

Troubleshooting

Message	What to do
Close Unreal Editor first	Redirects are read when Unreal starts and an open editor keeps the Blueprint loaded. Close it and run step 2 again.
The build failed	Fix the errors in the Problems panel (UStudio's error fixer helps), then **UStudio: Finish Blueprint to C++ Conversion**.
Unreal did not find the class after the build	The class did not compile into the editor (check its module is in the `.uproject` / `.uplugin` and loads). The redirects were removed.
Not re-saved: asset (could not be saved)	A read-only file or one checked out by someone else. It still loads through the redirects; re-save it later in Unreal Editor.
Changes not carried over	Listed in the UStudio output: set them on the C++ class or the child Blueprint by hand.

To undo: revert the re-saved assets, `DefaultEngine.ini` and the new source files with your source control (or restore the copy from `Saved/UStudio/BlueprintBackups` and remove the redirect lines UStudio added under `[CoreRedirects]`).

Tested on Unreal Engine 5.7 with about 1,000 Blueprints from 7 projects (every generated class compiles) and full conversions of placed actors, child Blueprints and referencing assets. The code is generated against your engine's headers, so it follows the APIs of your Unreal 5.x version.

Getting started

1. Open the folder that contains your `.uproject` (**File › Open Folder...**).
2. Click the **U** icon in the Activity Bar. UStudio detects the project and the engine version, and sets up IntelliSense.
3. If `compile_commands.json` hasn't been generated yet, click **Fix IntelliSense** in the UStudio sidebar, or run **Unreal: Fix #include & Intellisense** from the Command Palette (`Ctrl+Shift+P`).

User guide. Every feature, step by step, in the **UStudio Pro 1.22.0 User Guide (PDF)**.

Guided tour. The first time, the dashboard shows a **New to UStudio?** card that opens a step-by-step tour: IntelliSense setup, search and navigation, build and launch, new C++ classes and Content assets, error fixes, file history, Python scripts and config files. Each step has a button that runs it and is ticked off once you've done it. Reopen the tour any time with the graduation-cap button in the dashboard title bar, or with **UStudio: Getting Started Guide**.

Code intelligence

Unreal-aware C++ editing on top of clangd: navigation, references and fixes that know about reflection, RPCs and Blueprints.

![C++][chip-cpp] ![clangd][chip-clangd]

Automatic IntelliSense setup

- Generates and repairs `.clangd`, `compile_commands.json` and `.vscode/c_cpp_properties.json`.
- clangd parses each file with the precompiled header its module is built with, like UnrealBuildTool. Code that compiles thanks to the PCH (an automation test without `#include "Misc/AutomationTest.h"`, `#if ENABLE_DRAW_DEBUG`...) shows no false errors, even in a module that hasn't been compiled yet. Modules and definitions you add to a Build.cs reach clangd when you save it, without waiting for a build: their headers are found right away. Headers you open on their own (including engine headers via `F12`) are parsed without the PCH, so nothing is defined twice.
- Edits `.vscode/settings.json`, `launch.json` and `c_cpp_properties.json` in place: your comments, your own configurations and your `clangd.arguments` are kept. A file with a syntax error is reported, never overwritten.

Smart F12 navigation PRO

- **From `.h` to `.cpp`:** `F12` on any method (including `Server_Throw`) jumps straight to its implementation (`Server_Throw_Implementation`).
- **From `.cpp` to `.h`:** `F12` on `ClassName::Method_Implementation` goes back to the declaration in the `.h`, not into `.generated.h`.
- **On RPC call sites:** `F12` on `Server_Throw(...)` opens `Server_Throw_Implementation`.

Instant indexed references PRO

Find every C++ and Blueprint reference to a class, method or property in milliseconds (`Alt+Shift+F12`), backed by a persistent inverted index. Each search stays in the **Symbol References** view as its own entry (newest on top, older ones collapsed), so you can go back to it, search again (↻) or close it (✕ or `Delete`). The last 20 searches are kept between sessions.

"used by N" above every function

Like Visual Studio's reference count or Rider's usages: every function and `UPROPERTY`, in headers and `.cpp` files, shows **used by N**. It counts the C++ calls in the class and its subclasses and, in Pro, the Blueprints that use it.

- Functions with the same name in different classes are counted separately. Each call is attributed by its qualifier (`Super::`, `AWeapon::`), the receiver's type (`Weapon->`, `Cast<AWeapon>(X)->`, `Weapons[i]->`) or the enclosing class.
- Blueprint usages include every Blueprint that derives from the class, even through other Blueprints or C++ subclasses: one that calls the function on itself, overrides the event or changes the property's default counts too.
- `_Implementation` / `_Validate` bodies count the calls to the declared function.
- Click it to open a peek view with every usage, Blueprints included (their C++ view opens inside the same peek).

Real-time reflection sync PRO

No more waiting for UnrealHeaderTool to clear red squiggles in the `.cpp`. When you declare a `UFUNCTION` with `Server`, `Client`, `NetMulticast`, `BlueprintNativeEvent` or `WithValidation`, UStudio synthesizes the matching `_Implementation`, `_Validate` and `Execute_` declarations in real time.

Implementation generator PRO

Press `Ctrl+.` on a method declaration in the `.h` to generate its body in the `.cpp`.

- Works on one method or on several selected methods at once.
- Creates the matching `_Implementation` and `_Validate` bodies for RPCs and events.
- Fills in default return values and `Super::Method()` calls for overrides. An override of an interface function never calls `Super::` (the parent class doesn't have it): it gets a plain body, or `IMyInterface::Method()` when the interface already implements it.

Ghost implementations PRO

Open a `.cpp` and every function or event its header declares without an implementation appears where its body belongs, as a ghost of its code with an **Implement** button:

```
void AMyActor::Tick(float DeltaTime) { Super::Tick(DeltaTime); }  |  Implement
```

- **Implement** writes it there and puts the cursor in the body. **Implement all** writes every ghost of that spot.
- Each ghost goes after the function declared before it in the `.h`, so the `.cpp` keeps the header's order.
- `BlueprintNativeEvent` and RPCs appear as `_Implementation`, with their `_Validate` for `WithValidation`. `BlueprintImplementableEvent`, pure virtuals, inline bodies and templates never appear, nor functions implemented in another `.cpp`.
- Declarations under `#if WITH_EDITOR` keep their `#if`.
- When the `}` of the last function is the file's last line (no newline at the end), the ghost appears after that `}`, on the same line. Hover it for the whole code and **Implement**.
- Declare a function in the `.h` and its ghost appears in the `.cpp` as you type, before you save.
- **UStudio: Implement Missing Functions from the Header...** (command palette, or right-click in the `.cpp`) lets you choose among all of them.
- The Free edition shows the ghosts; writing them is Pro. Turn them off with `unrealStudio.codeLens.ghostImplementations`.

Implement interfaces PRO

Right-click an interface in the class declaration (`class AMyDoor : public AActor, public IInteractable`) and choose **Implement Interface Functions...** to pick the functions you want, or **Implement All Interface Functions**. Also from `Ctrl+.` on the interface name.

- Declares them in the class, inside a `//~ Begin IInteractable Interface` block: `Interact_Implementation(...)` for a `BlueprintNativeEvent`, `virtual ... override;` for the pure and optional virtuals. Functions under `#if WITH_EDITOR` in the interface keep their `#if`.
- Creates their bodies in the `.cpp`.
- Works with project interfaces and engine or plugin ones (`IAbilitySystemInterface`, `IGameplayTagAssetInterface`, `IGenericTeamAgentInterface`...), including the interfaces they derive from.
- While you type a declaration in that class (`virtual void Inter…`), IntelliSense suggests the interface functions it doesn't declare yet and completes the whole signature.

Real-time Unreal inspections PRO

Mistakes the C++ compiler accepts but Unreal punishes at build, link or run time are flagged as you type, each with a quick fix (`Ctrl+.`):

Rule	Problem	Quick fix
UE001	UObject pointer member without `UPROPERTY()` (the garbage collector can destroy it under you)	Add `UPROPERTY()` and use `TObjectPtr<>`
UE002	`UCLASS` / `USTRUCT` without `GENERATED_BODY()`	Add `GENERATED_BODY()`
UE003	`AddDynamic` / `BindUFunction` to a function that isn't a `UFUNCTION` (fails at run time)	Mark it `UFUNCTION()` in the header

Rule	Problem	Quick fix
UE004	Override of `BeginPlay`, `Tick`, `EndPlay`, `GetLifetimeReplicatedProps` … that never calls `Super::`	Insert the `Super::` call (last for teardown functions)
UE005	`Replicated` property missing from `GetLifetimeReplicatedProps`	Add `DOREPLIFETIME`, the function, its declaration and `Net/UnrealNetwork.h`
UE006	RPC / `BlueprintNativeEvent` without `_Implementation` (or `_Validate`)	Create the stub in the `.cpp`
UE007	`CreateDefaultSubobject` / `ConstructorHelpers` outside the constructor	— (explains the alternative)
UE008	UObject created with `new`	`NewObject<T>(this)` / `SpawnActor<T>()`
UE009	`#include` or reflected type from a module the `Build.cs` doesn't reference or link (C1083 / LNK2019 in the real build), or whose plugin isn't enabled	Add the module (editor-only ones inside `if (Target.bBuildEditor)`) and enable the plugin
UE010	Event / RPC parameters UnrealHeaderTool rejects or rewrites: `FString` / `FText` / `TArray` / `TMap` / `TSet` by value in `BlueprintNativeEvent`, `BlueprintImplementableEvent`, `Server`, `Client`, `NetMulticast`; RPC without `Reliable`, with return value or out parameters; `_Implementation` with other parameters; `BlueprintImplementableEvent` or RPC defined under its own name	`const T&` in the header and the `.cpp`, add `Reliable`, rename to `_Implementation`
UE011	`AddDynamic` handler whose parameters differ from the delegate (`OnComponentBeginOverlap`, `OnActorHit`, `OnTakeAnyDamage`, `UButton::OnClicked`, project delegates…)	Rewrite the handler's signature in the header and the `.cpp`
UE012	Declarations UnrealHeaderTool rejects: private `BlueprintReadWrite`, `BlueprintPure` without result, UObject member by value, interface pointer, missing / renamed / misplaced `.generated.h`	`AllowPrivateAccess`, `BlueprintCallable`, `TObjectPtr<>`, `TScriptInterface<>`, fix the include

Silence one line with `// ustudio-ignore UE001`, or turn a rule off with `unrealStudio.inspections.disabledRules`. **UStudio: Run Unreal Inspections on the Project** checks every file.

Include, module and linker quick fixes

In every edition, compiler errors such as *unknown type name 'UCameraComponent'* get a quick fix that adds the right `#include` **and** the missing `Build.cs` dependency, and enables the plugin in the `.uproject` / `.uplugin` when the module lives in one. In headers it can forward declare instead. **LNK2019 / LNK2001** unresolved externals are placed in the module's `Build.cs`, with a fix that adds the module that defines the symbol (`UUserWidget` → `UMG`).

Predictive autocomplete

Inline ghost-text suggestions that follow the Unreal Engine C++ coding standard, **filled with the real variables, members, enums, timers and log category of your code**. Accept with `Tab`, then `Tab` through the slots.

Group	Suggestions
Control flow	`if (condition)` with Allman braces, `if (IsValid(Actor))`, `if (HasAuthority())`, `if (IsLocallyControlled())`, `else`, `else if`, indexed `for`, range-based `for (const auto& Item : Items)`, safe reverse loop, `while (bIsRunning)`, `do { … } while (bCondition);`, `switch (State)` with cases
Multithreading	`AsyncTask(ENamedThreads::GameThread, …)`, `AsyncTask(ENamedThreads::AnyBackgroundThreadNormalTask, …)`, `ParallelFor(Items.Num(), …)`, `FScopeLock ScopeLock(&CriticalSection);`

Group	Suggestions
Diagnostics	`UE_LOG(LogTemp, Warning, …)`, `UE_LOG(LogTemp, Error, …)`, `check(…)`, `checkf(…)`, `ensure(…)`, `ensureMsgf(…)`
Reflection macros	`UPROPERTY(EditAnywhere, BlueprintReadWrite, …)`, `UPROPERTY(VisibleAnywhere, BlueprintReadOnly, …)`, `UPROPERTY(ReplicatedUsing = …)`, `UFUNCTION(BlueprintCallable, …)`, `UFUNCTION(BlueprintPure, …)`, `UFUNCTION(Server, Reliable, WithValidation)`, `UFUNCTION(Client, Reliable)`, `USTRUCT(BlueprintType)`, `UENUM(BlueprintType)`
Objects	`Cast<T>`, `CreateDefaultSubobject<T>`, `NewObject<T>`, `FindComponentByClass<T>`, `GetWorld()->GetTimerManager().SetTimer(…)`

- The same snippets are in the `Ctrl+Space` list, with tab stops.
- In C / C++ files only `Tab` accepts a suggestion: `Enter` inserts a new line and typing `(` or `;` never inserts it, so typing `if` gives you `if`, not a template (`unrealStudio.suggestions.acceptOnlyWithTab`).

Format Document (Unreal style)

Right-click a `.h` / `.cpp` › **Format Document (Unreal)** and choose braces on a new line (Allman, the Unreal standard), braces on the same line (`if (bReady) {`), or your project's `.clang-format`. It runs clang-format with Unreal conventions:

- tabs and `Type* Name`;
- includes keep their order;
- reflection macros stay above their declarations;
- long lines are left as they are.

With a selection, only those lines are formatted. clang-format comes with Visual Studio's C++ Clang tools and with LLVM. Set `unrealStudio.format.braceStyle` to stop being asked.

Build and debug

Build, launch and debug Unreal Editor from VS Code, with errors you can click and fixes you can preview.

One-click build and launch

Build, Rebuild and Launch Unreal Editor with one click from the status bar or the dashboard.

C++ debugging

Debug gameplay code and engine systems with breakpoints and no manual configuration.

- **One-click debug launch: Debug Unreal** in the status bar, or **Launch Unreal Editor (Debug Mode)** in the dashboard, starts `UnrealEditor.exe` with the debugger attached (or attaches if the editor is already running).
- **Readable Unreal types:** `Unreal.natvis` is linked automatically, so `FString`, `FName`, `TArray`, `TMap`, `TSet`, `FVector` and `UObject*` show their values instead of raw memory.
- **Clear status:** while debugging, the status bar turns amber with the PID and the session controls.
- **Shortcuts** (Visual Studio / Rider keymap): `F5` start or attach, `Shift+F5` stop and detach (Unreal Editor keeps running), `F9` toggle breakpoint, `Ctrl+Shift+F9` clear all breakpoints.

Build errors in the Problems panel

Compiler (MSVC / clang), UnrealHeaderTool, `*.Build.cs` and linker errors from every build appear in the **Problems** panel with clickable file and line, with MSVC `note:` lines attached to their error. If Live Coding is active in the editor, UStudio offers to compile with Live Coding instead of showing a cryptic UBT failure. Linker errors land in the module's `Build.cs`, with a fix that adds the missing module.

Error fix suggestions with multi-file preview

Open the **Build Errors & Fixes** panel (next to **Unreal Log**), or right-click a compilation error there or in **Unreal Log**.

- **Suggest Fix (Preview)...** is highlighted when UStudio has an automatic fix for that error, and greyed out when it doesn't.
- The preview shows the diagnosis, every file that will change (`*.h`, `*.cpp`, `*.Build.cs`, `*.uproject`) with a color-coded unified diff, and a button to compare in VS Code's diff editor. Nothing is written until you press **Apply Fix**.

Errors it fixes:

- **Missing `#include`, `Build.cs` module or `.uproject` plugin** (`EnhancedInput`, `Niagara`, `UMG`, `AIModule`, `GameplayTags`, `GameplayAbilities`, `Json`, `HTTP` ...).
- **Circular header dependencies:** the `#include` in the header becomes a forward declaration (`class A…;` / `struct F…;`) and moves to the `.cpp`.
- **Missing semicolons** (`C2143`, `C2144`) after a declaration or a class / struct closing brace.
- **Misdeclared types:** `C4099` `class` vs `struct`, and missing `A` / `U` / `F` prefixes (`Vector` → `FVector`, `Rotator` → `FRotator`, `Actor` → `AActor`, `Character` → `ACharacter`, `UserWidget` → `UUserWidget`).
- **Include paths not found** (`C1083`) and **unresolved externals** (`LNK2019` / `LNK2001`): missing `Build.cs` module, missing `<MODULE>_API` export macro, or a header method without a body in the `.cpp`.
- **UHT and replication errors:** `.generated.h` include order, missing `GENERATED_BODY()`, missing `_Implementation` for RPCs / `BlueprintNativeEvent`, missing `GetLifetimeReplicatedProps` and `Net/UnrealNetwork.h`.

Automation tests in the Testing view PRO

`IMPLEMENT_SIMPLE/COMPLEX_AUTOMATION_TEST`, Automation Specs (`Describe` / `It`, with their real names) and CQTest `TEST_CLASS` / `TEST_METHOD` appear in VS Code's Testing view, with ▶ buttons in the gutter. UStudio builds the editor target if needed and runs them headless with `UnrealEditor-Cmd` (or under the debugger with **Debug Test**); failures show their

message at the failing line. Specs and complex tests expand into their real cases after the first run.

Live Unreal Editor

UStudio talks to the Unreal Editor that is running your project: open classes from the editor, compile into it, and watch your C++ run.

![Unreal Editor][chip-ue]

Live Bridge plugin

Double-click a C++ class in Unreal Editor and it opens in your VS Code (or Cursor) window.

- Plugin v4 lets VS Code / Cursor trigger Live Coding in the editor; v5 adds the `UStudioTrace` runtime module behind Live Execution.
- UStudio **asks before** adding the plugin to `Plugins/UnrealStudioBridge` (set `unrealStudio.bridge.pluginInstall` to `always` or `never` to skip the question) and never modifies your `.uproject` or shared `Config/*.ini` files.
- The plugin only takes over source code access on machines where UStudio is used for the project, so teammates who get it through source control keep their own IDE.
- The local bridge only accepts requests that carry a per-session token, and rejects browser (cross-origin) requests.

Live Coding

`Ctrl+Alt+F11` (or the **Live Coding** status bar button that appears while Unreal Editor runs) saves your files and compiles them into the running editor, like the editor's own shortcut. The result appears in the status bar; on failure the compiler errors go to the Problems panel. When a normal build is refused because Live Coding is active, the error offers **Compile with Live Coding**. Requires Live Bridge plugin v4.

Live Execution PRO

See your C++ run, like the execution highlight of Blueprint nodes. While Unreal Editor runs (in Play in Editor or not), the functions that execute light up:

- **Running now:** the signature glows amber and pulses, with its name in bold and `⚡ running · 60/s · 1,204 calls` beside it. Header declarations glow too, so a `UFUNCTION(BlueprintNativeEvent)` lights up when its `_Implementation` runs.
- **Just ran:** the glow fades out over 3 seconds (`⚡ 0.8 s ago`).
- **Ran earlier:** a faint bar and `✓ 12 calls · 34 s ago` stay until the next Play or **Clear Marks**. The overview ruler shows them all at a glance.

How to trace:

- Right-click a function › **Trace This Function Live** adds `USTUDIO_TRACE();` as its first line (run it again to remove it). **Trace All Functions in This File Live** and **Stop Tracing This File** work on a whole file.
- UStudio adds `#include "UStudioTrace.h"` and the `UStudioTrace` module dependency to the `Build.cs` for you. After that first build, Live Coding is enough to trace more functions.
- The **N running** status bar item lists the traced functions, running ones first, updating live; `Enter` opens one.
- `USTUDIO_TRACE()` costs a few atomic operations per call and compiles to nothing in Test and Shipping builds. Requires Live Bridge plugin v5. Settings: `unrealStudio.liveExecution.*` (color, fade time, reset on Play, call counts).

Unreal Log panel

An **Unreal Log** tab next to the terminal follows the Unreal Editor Output Log (`Saved/Logs`), including Play in Editor sessions.

- Filter by Errors / Warnings / Messages / Verbose and by category, search with text or `/regex/` , toggle timestamps, auto-scroll.
- Source locations (`MyActor.cpp(42)` , `ensure` reports `[File:...] [Line: 42]`) and asset paths (`/Game/Blueprints/BP_Player`) are clickable.

- The tab shows a badge with the number of new errors. **Choose Log File** switches to another session's log (a `-game` or server run).

Live editor status

When your project is open in Unreal Editor, the dashboard, Solution Explorer and status bar show it:

- The project card gets an emerald / cyan accent bar and a live banner (`EDITOR RUNNING · PID · Bridge Live`), amber during Play in Editor and violet while debugging.
- **Launch Editor**, **Live Coding**, **Unreal Log** and **Python Studio** light up when they're ready to work with the running editor.

Blueprints and assets

Read Blueprints as C++ and see how your C++ is used from Content, without opening Unreal Editor.

Blueprints as C++ classes PRO

Open any `.uasset` (Blueprint, Widget or Anim Blueprint, Interface, Function Library, User Defined Struct / Enum or data asset) as a read-only C++ header, without Unreal Editor:

- `UCLASS` / `UPROPERTY` / `UFUNCTION` with the real specifiers (`EditAnywhere` , `BlueprintReadWrite` , `Replicated` , `Category` , `ExposeOnSpawn` ...), tooltips and default values.
- Components, designer widgets (`BindWidget`), event dispatchers, custom events, RPCs, overridden events (BeginPlay, Construction Script...) and bound component events.
- `F12` on a type opens its engine or project header, or the other Blueprint's C++ view. The Outline lists every member, and the view refreshes when Unreal saves the asset.
- Double-click a `.uasset` in VS Code, click it in the Solution Explorer, or right-click › **View as C++ Class**.

Convert Blueprints to C++ PRO EXPERIMENTAL

Turn a Blueprint of an actor, component or object into a C++ class, and make everything that used the Blueprint use the class: actors placed in levels, child Blueprints, variables, Cast and Spawn nodes, class defaults. Right-click a `.uasset` › **Convert to C++ Class (Experimental)...**, or use the button in the title bar of its C++ view.

- **Step 1, the class.** Components with their hierarchy, attachments and settings (meshes, materials, collision, transforms), variables with their defaults and specifiers, event dispatchers, functions, custom events, RPCs, replication (`GetLifetimeReplicatedProps`), overridden events (`BeginPlay` , `Tick` , `OnConstruction` ...), `BlueprintNativeEvent` implementations and component events bound in `BeginPlay` . Uncheck what should stay out. The code is read against your engine's headers, so it uses the APIs of your Unreal version, and the modules it needs are added to the `Build.cs` .
- **Step 2, the references** (Unreal Editor closed): UStudio builds the project, moves child Blueprints onto the new class (keeping what they changed on inherited components), adds `[CoreRedirects]` to `Config/DefaultEngine.ini` , has a headless Unreal re-save every asset that referenced the Blueprint, and deletes it. A copy is kept in `Saved/UStudio/BlueprintBackups` ; under Perforce the assets are checked out and the Blueprint is marked for delete.
- **Graph logic is not translated.** Each function and event gets a body with a `TODO` to port it; the wizard warns when the Blueprint has logic. Variables typed as another Blueprint use its C++ parent, and Timelines, Blueprint structs / enums and Blueprint interfaces are listed as not converted.
- Widgets, Anim Blueprints, levels, materials and data tables are not converted. Commit or shelve your work before step 2.

Blueprint and asset CodeLens PRO

Above your `UCLASS` and `USTRUCT` declarations, see how many Blueprints (`.uasset` and `.umap`) inherit from or reference them; `UFUNCTION` s and `UPROPERTY` s include their Blueprint usages in **used by N**. Click to inspect the connected Blueprints or copy their `/Game/...` reference paths.

- **N derived Blueprints** above each `UCLASS` lists the Blueprints that have it as parent class: directly, through other Blueprints (`BP_Rifle` ← `BP_WeaponBase` ← `AWeapon`) or through C++ subclasses. Click it to pick one: `Enter` opens its C++ view, the buttons open it in Unreal Editor or reveal the `.uasset` . Also in the editor's right-click menu: **Show Blueprints Deriving from This Class**.
- **Open Asset in Unreal Editor** (title bar of a Blueprint's C++ view, or right-click a `.uasset` in the Explorer) opens the asset's editor in the running Unreal Editor. It needs the Python plugin with Remote Execution enabled, or the Live Bridge plugin.

Class Lifecycle diagram PRO

Class Lifecycle (editor title bar, context menu or dashboard) shows the life of the class at the cursor as a tree: where it appears when the game starts, who creates it, every engine callback in order with what your code does in it (creates, binds, timers, input), events, RPCs and replication. It also warns about missing `Super::` calls, unimplemented RPCs and replicated properties without `DOREPLIFETIME`. Works for C++ classes and Blueprints.

Node Graphs

Build C++ and Python classes with Blueprint-style nodes, compile them into the running Unreal Editor, and turn them into regular code when they're done. A graph is a `.ugraph` file of your project, so Git and Perforce work as usual.

Visual class editor

Right-click any folder (Explorer, Solution Explorer, Favorites) › **New Node Graph (C++ / Python)...**, or **Node Graph** in the UStudio panel: a wizard with cards for C++, Python or a custom node, the module and code folder taken from the folder you clicked, and the starting events. Pick C++ or Python, the parent class (Actor, Pawn, Character, components, User Widget, Player / AI Controller, Game Mode, Game Instance, subsystems, Anim Instance, Function Library, Data Asset or any project class; for Python an editor tool, Object, Actor, Component or Function Library) and a name.

- **A canvas like the Blueprint editor.** Right-click or `Space` opens the searchable node list; drop a wire on empty space to see only the nodes that fit, connected for you. Drag pins to wire, `Alt+click` breaks a wire, double-click one for a reroute. Comments (`C`), copy / paste / duplicate (`Ctrl+D`), box select, zoom with the wheel, pan with the right or middle button.
- **Arrange like in Unreal.** `Q` straightens wires, `Shift+W` / `S` / `A` / `D` align top / bottom / left / right, `Alt+Shift+W` / `S` align middle / center, `L` arranges the selection or the whole graph, arrow keys nudge. Pins only connect when their types fit, as in Blueprints (a parent class needs a Cast).
- **My Class panel.** Variables (any type, arrays, defaults, Instance Editable, Read Only, Expose on Spawn, Replicated), functions (inputs, outputs, pure, static), components with their hierarchy and properties (mesh, collision, size, camera...), and event dispatchers. Drag them to the canvas: Get / Set, call, Bind...
- **Graphs use each other, like Blueprints.** Another graph's class works as a type, in Cast To and as a parent. Drag a wire from its pin to call its functions and custom events, read and write its variables, or bind to its dispatchers. The project's C++ classes and plugins work the same way (their public `UFUNCTION`s / `UPROPERTY`s, and library functions). Compiling a graph also compiles the graphs it uses.
- **Class Defaults**, like in Blueprints: variable defaults and the inherited properties: replication, tick, auto possess, Character Movement, the Game Mode's classes...
- **Inherited components.** Components shows what the parent class already has (a Character's capsule, mesh and movement), with its hierarchy. Attach your components to them, use their events and set their properties; the parent's root stays the root.
- **Threads.** Run on Game Thread, Run on Background Thread, Run on Thread Pool and Parallel For, with a warning when a node that needs the game thread runs on another one.
- **Copy and paste as JSON** through the system clipboard: paste nodes into another graph, window or chat.
- **Engine classes and plugins, like Blueprint.** Drag a wire from any reference (Character Movement, an actor, a component) for its functions, properties and events, inherited ones included (Get Velocity, Is Falling, Bind Event to On Component Hit...). Async actions from C++ plugins (e.g. SteamCorePro's Create Session) are latent nodes with On Success / On Failure, and structs, enums and maps are pin types with Make / Break nodes.
- **Asset and class pickers** like Unreal's, for the project's, plugins' and engine's assets and classes (C++ and Blueprint), with Use Selected / Browse to Asset in the Content Browser.
- **Timelines** with a curve editor, Float Curve variables, and a color picker.
- **Game framework, subsystems and spawning.** Get Game Instance / Game Mode / Game State / Player State / HUD, and the Game Instance, World, Local Player and Engine subsystems with the class you pick. Spawn Actor (typed), sounds, Niagara and Cascade effects, decals, widgets, objects and components, with assets picked by path (`/Game/...`).
- **Details, like Unreal.** Select a component to subscribe to its events with the **+** buttons (On Component Begin Overlap, End Overlap, Hit). Custom events can replicate (Run on Server, owning Client, Multicast, reliable). Every node can have its own color, icon and note.
- **Code tab.** The `.h` / `.cpp` or `.py` as you edit, with the selected node's lines highlighted. Click a line to find its node.
- **160+ nodes.** The events of each parent class, flow control (Branch, Sequence, For / For Each / While, Switch, Do Once, Do N, Gate, FlipFlop), Cast, math, vectors, rotators, strings and arrays, actor and component functions, Delay and Retriggerable Delay, timers, **Async Task** (a background thread, then back on the game thread), **Async Load Asset**,

event dispatchers with every kind of delegate (dynamic multicast, multicast, dynamic and single: Call, Bind, Unbind, Unbind All, Is Bound), Unreal Editor scripting for Python (selection, assets, actors, dialogs), and **Custom Code** / **Custom Expression** nodes for any C++ or Python.

Compile into Unreal Editor PRO

Compile (`F7`) checks the graph and builds a temporary copy of the class into the running editor, so you can use it right away without converting anything:

- **C++:** Live Coding into the open Unreal Editor, or a build when it is closed. The copy lives in `Source/<Module>/Private/UStudioGraphs` .
- **Python:** loaded live through Remote Execution. **Run** runs an editor tool's Run Script event.
- **Compiler Results** lists the graph checks and the compiler's errors and warnings, each on the node that produced the line, like Blueprint compile errors.

Live debugging PRO

Turn on **Debug** (it compiles the class for debugging when needed): while the game (Play in Editor) or the tool runs, the nodes that run light up, the exec wires flow and each node counts its runs, like the Blueprint debugger. Pick which object to watch in the instance list, as in Blueprint's debug object list. Put breakpoints on nodes with `F9` : in Play in Editor the game pauses at the node, like a Blueprint breakpoint, with Continue (`F5`), Step (`F10`) and Stop, no debugger needed. C++ uses the Live Bridge plugin (v7) and its `USTUDIO_TRACE` ; UStudio sets it up the first time.

Convert to Code PRO

To Code writes the final `.h` / `.cpp` into the module (or the `.py` into `Content/Python`) as regular project code and removes the temporary copy. The class keeps its name, so Blueprints made from the compiled class keep working. Compiling the graph again updates that code while nobody edited it; files edited by hand are never replaced without asking.

Custom nodes

Make your own nodes (`.unode`), saved in the project (`.ustudio/nodes` , shared through source control) or just for you (`~/.ustudio/nodes`):

- **With sub-nodes**, like Blueprint macros: a graph between an Inputs and an Outputs node.
- **With code**, for C++ and / or Python: `{Input}` reads an input, `{$Output} = ...` sets an output.
- **Or both:** a language with code uses it, the other uses the sub-nodes.
- Select nodes › right-click › **Collapse to Custom Node** makes one from a selection. Give it a category, a color, an icon or an image.

Navigation and export

Back / Forward (`Alt+←` / `Alt+→` , mouse buttons), **Find** (`Ctrl+F`) across every graph, **Go to Definition** (`F12` : a function's graph, a variable, a custom node, or the engine header of a C++ call), **Show in Code**, and from the generated code back to the graph (**Show in Node Graph**, CodeLens). **Export to PDF** PRO writes every graph as drawn, in vector with selectable text, plus the generated code.

Appearance

The **Style** tab, Unreal by default: VS Code and Light presets, background color, grid and image, node body, borders, corners and font size, header colors per kind of node, pin colors per type, and wires (curved, straight or right-angled, thickness and colors). One wire can have its own color (right-click it).

Python classes that derive from Unreal classes exist while Unreal Editor runs (import them from `init_unreal.py` to have them at startup). Event dispatchers, delegates, timers and the async nodes are generated for C++.

Wizards and templates

Create classes, gameplay systems, Content assets, modules and plugins with a preview of every file before it's written.

New C++ class wizard PRO

New Unreal C++ Class (.h + .cpp) from the Solution Explorer's right-click menu, or **Unreal: Create New C++ Class** from the Command Palette: pick the parent class (engine classes and your project's own), the module and the folder, and see where the `.h` and `.cpp` will go before they're created.

Modular gameplay templates PRO

Component-driven C++ classes ready to drop into any UE 5.0 – 5.8 project, from the class wizard or **UStudio Pro: Create Modular Gameplay Template**:

- **Modular Character** and **Ability Component**;
- **Projectile**, **AoE**, **Dash** and **Melee Combo** abilities;
- **Attribute Component** (Health / Stamina / Mana / Armor with delegates);
- **Status Effect Component** (DoT, HoT, Slow, buffs);
- **Interactable System** (`UINTERFACE` + `UInteractionComponent` + `AInteractableActor`).

Every preview shows the code with line numbers and VS Code's colors (Dark+ / Light+).

Your own templates PRO

Turn the classes and scripts you write over and over into templates, in C++ and Python:

- **New Template** in the class wizard's Templates tab or in Python Studio, **Save as Template** on any preview, or right-click a `.h` / `.cpp` / `.py` file > **Save as Template**. The class name, API macro and includes become placeholders on their own.
- Placeholders: `{{ClassName}}` , `{{BaseName}}` , `{{API}}` , `{{Module}}` , `{{Header}}` , `{{Project}}` , `{{Copyright}}` , `{{Year}}` , `{{Date}}` (C++) and `{{FileName}}` , `{{Project}}` , `{{Year}}` , `{{Date}}` (Python), edited with syntax highlighting.
- **My templates** live in `~/.ustudio/templates` and work in every project, in VS Code and Cursor. **Project templates** live in the project's `.ustudio/templates` : commit them and the whole team gets them.

Content asset wizard PRO

Create real `.uasset` files in `Content/` with your own parent classes and default property values. Unreal Editor creates them: UStudio runs the generated Python script in the editor that has the project open (Remote Execution or the Live Bridge), or headless with `UnrealEditor-Cmd` when it's closed (about a minute while Unreal starts). Blueprint variables need UE 5.6 or newer.

- **Core framework:** `GameInstance` (`GI_`) and `GameMode` (`GM_`), with one-click generation of its `PlayerController` , `Pawn` / `Character` , `PlayerState` , `GameState` and `HUD` .
- **UMG widgets (`WBP_`):** blank widgets, reusable pieces (`WBP_ModularButton` , `WBP_StatBar` , `WBP_AbilitySlot`) or full layouts (`WBP_MainMenu` , `WBP_GameplayHUD` , `WBP_PauseMenu` , `WBP_SettingsMenu` , `WBP_InventoryGrid` , `WBP_InteractionPrompt`), with a live, interactive 16:9 preview in the wizard. The layout is built on the widget's root CanvasPanel as previewed.
- **Gameplay and data:** `InputAction` (`IA_`), `InputMappingContext` (`IMC_`), Blueprint `Actor` , `ActorComponent` , `UObject` , `DataAsset` and `DataTable` .

New module and plugin wizards PRO

Create C++ modules (in the project or inside a plugin) and plugins (Blank, Blueprint Function Library, Runtime + Editor, Editor tool with a Tools menu entry, Content only) from the Solution Explorer or the dashboard. The wizard previews every file it creates and shows a diff of the `.uproject` / `.uplugin` / `*.Target.cs` changes before you confirm, keeps their formatting, and can refresh IntelliSense and build right after.

Python for Unreal Editor

Write, run and debug `import unreal` scripts for UE 5.0 – 5.8.

![Python][chip-python]

Python Scripting Studio PRO

- **Full `import unreal` IntelliSense:** configures `Intermediate/PythonStub/unreal.py` and generates a project-aware stub that includes your own C++ `UCLASS`, `UPROPERTY` and `UFUNCTION` symbols exposed to Python, plus inline completions, hover documentation and ▶ **Run in Unreal Editor** CodeLenses.
- **Three ways to run:** live in the open editor through Unreal's Remote Execution protocol (UDP/TCP, `239.0.0.1:6766`) or the UStudio Live Bridge (`POST /python/exec`), or headless with `UnrealEditor-Cmd.exe`.

29 searchable templates for the work studios automate most:

Area	Templates
Sequencer and cinematics	Sequence from selected actors; camera shot (CineCamera, Camera Cut, dolly, lens and focus keys); master sequence with SEQ/SH shots from a shot list or CSV; animations on characters; timing offset and frame rate change; sequence QA audit; FBX export; audio and fades
Rendering	Movie Render Queue batch render (EXR / PNG / JPG / ProRes); asset turntables
Animation	Root motion, root lock and footstep notifies; batch IK retargeting
Import and pipeline	Batch import from disk; texture settings by suffix; Static Mesh Nanite / LOD / collision pass
Assets, levels and data	Naming prefixes; unused assets; content budget report; Material Instances; Blueprint defaults; DataTable ⇄ CSV / JSON; blockout mesh swap; CSV / JSON actor layout; World Outliner cleanup; editor menus; non-blocking batch processing

Python debugging PRO

Debug the scripts that run inside Unreal Editor the way you debug C++: breakpoints, stepping, call stack, variables, watches, and a Debug Console where you can call the `unreal` API while paused.

- **Debug in Unreal Editor:** the debug button in a `.py` file's title bar, the CodeLens or the right-click menu runs the script in the editor and stops at your breakpoints.
- **Stays attached:** breakpoints also hit when Unreal runs Python by itself (menu entries, `py` commands, startup scripts). **Attach Python Debugger to Unreal Editor** attaches without running anything; `Shift+F5` detaches and the editor keeps running.
- **Ready in `launch.json`: Unreal Python - Debug Current Script (UStudio)** and **Unreal Python - Attach to Editor (UStudio)** work with `F5`.
- Uses Python Remote Execution and Microsoft's **Python Debugger** extension (debugpy); UStudio offers to set up both. The editor is frozen while a script is paused at a breakpoint.

Source control and history

Know who changed what, keep every version of your files, and go back to any point in time.

![Git][chip-git] ![GitHub][chip-github] ![Perforce][chip-perforce]

Source control status

- Detects whether the project uses **Git** (GitHub, GitLab, Azure DevOps, Bitbucket) or **Perforce**.
- **Status bar:** current branch, incoming (↓) and unpushed (↑) commits, modified and untracked files, and merge conflicts.
- **Background fetch:** tells you when teammates push to your branch, so you don't build against stale code.
- **Perforce:** `p4 edit` (check out), `p4 sync` (Get Latest), `p4 revert -a` (revert unchanged) and a badge on every file for its Perforce state.

Perforce file badges

Like P4V, every file and folder in the Explorer, the Solution Explorer and the editor tabs shows its Perforce state. Hover a file for the details: who has it checked out and in which changelist, its `#have` and `#head` revisions, its file type.

Badge	State
`✓`	Checked out by you
`+` `⤫` `➔` `⇔`	Marked for add, delete, moved here, opened for integrate
`!`	Needs resolve (`✓!`)
`↓`	Newer revision in the depot, deleted in the depot, or not synced. `✓↓` : checked out and out of date
👥	Checked out by someone else. `✓👥` : by you and someone else
🔒	Locked: an exclusive (`+l`) file checked out by someone else, or locked with `p4 lock`
✎	Modified or made writable without checking it out
⏸	Shelved in one of your changelists
`?`	Not in the depot (a local file that hasn't been added)
∅	Not mapped: outside the workspace view
dimmed	Ignored by `P4IGNORE`
•	Folder with checked-out, conflicting, modified or out-of-date files inside

UStudio asks the server about each folder when its files appear on screen, and again when you come back to VS Code and every few minutes. Set `unrealStudio.perforce.fileDecorations` to `changes` to mark only your own changes, out-of-date and locally modified files, or `off`. The colors are `ustudio.perforce.*` in `workbench.colorCustomizations`. The legend is in the source control menu (click the P4 status bar item).

Perforce changelists PRO

A P4V-style panel: right-click a file › **Check Out / Shelve / Submit (Perforce)...**. Pick a new, default or pending changelist and write its description, then check out (optionally with a lock), shelve (optionally reverting afterwards), submit or revert the checked files. Each file shows whether it is read-only, new, in another changelist, exclusively checked out by a teammate or behind the depot, with a diff against your `#have` revision.

Read-only file protection

- Read-only files show a lock badge in the Solution Explorer and in VS Code's Explorer. In a Perforce workspace the Perforce badges show the file's state instead, and 🔒 means a Perforce lock.
- Editing one prompts right away with **Check Out (Perforce)**, **Check Out to Changelist...** or **Make Writable**, so you don't lose the edit at save time.
- A status bar item shows the current file's read-only state, with one-click actions.

Code authors

Who created and last edited the file and each class, struct, enum, property, function, event and delegate, shown as CodeLens, plus the author of the current line. Uses `git blame` (including unsaved and uncommitted changes) or Perforce `p4 annotate`. Turn it on or off with the **Authors** switch in the dashboard.

Local file history

UStudio tracks your edits automatically: **10 versions** per file by default (`1` to `100` with `unrealStudio.fileHistory.maxVersions`), stored as unified `.diff` snapshots in `<temp>/ustudio-file-history/`. The **File Version History** panel shows any revision as a unified or side-by-side diff and restores it with one click.

Change Timeline

Every change to every file of the project, day by day, from three sources you can show or hide: **local saves** (file history), **Git commits** and **Perforce changelists**. Open it from the dashboard (*Diagnostics & Tools*) or with **UStudio: Change Timeline**. Everything in it is free except **Apply**.

- **Day bar:** one column per day, as tall as the number of changed lines. Drag it, click a day, or use the date picker or the ‹ › buttons to jump between days with changes.
- **Step slider:** like an undo history, one notch per save, commit or changelist of the day. Drag it, click a change in the list, or use `←` / `→` (`Shift+←` / `Shift+→` jumps a day). Every file of that change shows its diff at once, with line numbers; open any file side by side or at the changed line.
- **Files tab:** every file changed that day. Click one to follow only that file on both bars.
- **Search** like the Solution Explorer: best matches first (exact name, prefix, start of a word, CamelCase initials, or a path with `/`), with Match Case, Whole Word and Regex. The day bar marks the days with matches; `Enter` jumps to the best one.
- **Export day...:** the day's changes as a unified `.diff`, a Markdown report or a self-contained HTML report.
- **Apply...** PRO puts files back the way they were at the slider's position: only the files of that change, or every file changed since that moment. One confirmation lists what will be restored, created or deleted. Perforce files that aren't checked out are flagged and checked out (`p4 edit`) only if you confirm; files locked by someone else are skipped, and you're told when you don't have the latest revision. `Ctrl+Z` undoes it.

Navigation and workspace

Find any file, member or symbol fast, and arrange VS Code the way you're used to.

Solution Explorer

A Rider-style project tree for `Source/`, `Plugins/`, `Config/` and `Content/`, with right-click menus to create Unreal C++ classes, files and folders (mirrored in `Public/` and `Private/`), rename and delete.

- **Follows the active file:** switching tabs reveals and highlights the file, expanding its folders. The **Reveal Active File** button in the title bar does it on demand.
- **Sorts like Windows Explorer: Sort By** (title bar, or right-click a folder) sorts by name (natural order: `Level2` before `Level10`), type, date modified or size, ascending or descending. Folders can go first or mix with files, and `Source`, `Plugins`, `Config` and `Content` can stay on top. Sorting by date or size shows it next to each item.
- **Searches file names and contents:** type at the top of the tree (`Ctrl+;` from anywhere, `Ctrl+F` inside it) and it shows only the results, with their folders:
  - **files by name**, best match first and preselected: `weapon` finds `Weapon.h` before `BP_Weapon.uasset`, `wc` finds `WeaponComponent.h`;
  - **lines that contain the text**, under each file, with the match highlighted.

  `Enter` opens the file or jumps to the line, and ↑ / ↓ move through the results. Match Case, Whole Word and Regex are supported. Content search uses the ripgrep that ships with VS Code / Cursor, so results arrive while you type. `Intermediate`, `Binaries`, `Saved`, `DerivedDataCache` and `.uasset` contents are skipped.
- `F2` renames and `Delete` deletes the selected file.

Favorites

The files and folders you use most, one click away in their own **Favorites** view.

- **Add or remove:** click the ☆ that appears when you hover a row in the Solution Explorer (favorites keep a yellow ★; click it to remove), or right-click › **Add to Favorites** / **Remove from Favorites**. The same entries are in VS Code's Explorer and in the editor tab's right-click menu, so engine headers can be favorites too. The ☆+ button in the view adds the active file.
- **The view:** folders first, then files, each with its location (`Source/MyGame/Public`, `Engine/Source/Runtime/...`). Favorite folders expand like in the Solution Explorer. Remove a favorite with its ✕ (on hover), middle-click or `Delete`; the files stay on disk.
- **Saved per project:** favorites follow files and folders renamed, moved or deleted from UStudio or VS Code. A favorite deleted outside VS Code stays in the list as **not found** until you remove it or it comes back (another branch, a Perforce sync).

Members view

Lists the variables, events and functions of the active class. The `.h` and its `.cpp` are shown together, one line each with its declaration (`Fire(int32 Count) : void`, `Ammo : int32`) and short tags (`BP`, `RPC Server`, `replicated`, `private`…).

- **Search** with `Alt+M` from a C++ editor: filter by name or initials (`ga` → `GetAmmo`), then press `Enter` to jump.
- Filter variables / events / functions, and sort in declaration order or A–Z.
- Click an entry to open its declaration in the `.h`, or its implementation when you're in the `.cpp`; ⇄ switches between the two.
- The member under the cursor stays highlighted.

Panel layout and presets

Every UStudio view (Dashboard, Solution Explorer, Favorites, Members, Symbol References, C++ Classes & Blueprints) can open as an **editor tab** (drag it anywhere, split it, maximize it or move it into a floating window) or move to the panel or the secondary side bar. **Reset Panel Layout** (view menu `...` or dashboard) restores the defaults.

Layout presets (dashboard › *Diagnostics & Tools*, view menu `...`, or **UStudio: Apply Layout Preset**) arrange the side bars, the panel and the UStudio views in one click:

Preset	Layout
VS Code	UStudio on the left; Build Errors and Unreal Log in the bottom panel
Visual Studio	Solution Explorer and Members on the right; Error List, Output and Find References under the editor
Rider	Explorer and Members (Structure) on the left; Find Usages, Build and Log in a full-width bottom panel
Widescreen	Three columns: Solution Explorer, the code, then Members, References and Classes in the secondary side bar
Header | Source	The active class's `.h` and `.cpp` side by side
Focus	Only the code
Unreal Live	The Unreal Log beside the code while the editor runs or you Play in Editor

Bind a preset to a key (ids: `vscode`, `visualStudio`, `rider`, `widescreen`, `headerSource`, `focus`, `live`):

```
{ "key": "ctrl+alt+1", "command": "unrealStudio.applyLayoutPreset", "args": "rider" }
```

Export to PDF PRO

Right-click a folder or a file in the Explorer, the Solution Explorer, Favorites or an editor tab › **Export to PDF**. Select several to put them in one PDF.

- **The structure first:** the folder tree with each file's size and the page where it starts. Click a row to jump there; the PDF's bookmarks follow the same folders.
- **Then every text file:** its path and name in a header, its line count and size, and its contents with line numbers and syntax colors (C++, C#, Python, JSON, `.ini`). Long lines wrap with an indent. Unsaved changes are exported as they appear in the editor.
- **Structure only** leaves the contents out.
- **Any language:** the PDF embeds Consolas and Segoe UI (Menlo and Arial on macOS, DejaVu on Linux), plus Chinese, Japanese, Korean and symbol fonts for the characters they lack. The text can be searched and copied. Files are read as UTF-8, UTF-16 or Windows-1252.
- **Left out:** `Binaries`, `Intermediate`, `Saved`, `DerivedDataCache`, `.git`, `.vs` and what `files.exclude` hides (`unrealStudio.exportPdf.exclude`), unless you export that folder itself. Binary files such as `.uasset`, and files over 1 MB (`unrealStudio.exportPdf.maxFileSizeKB`), are listed without their contents.
- Page size (A4 or Letter), font size, line numbers and colors: `unrealStudio.exportPdf.*`.

Visual Studio and Rider shortcuts

UStudio keeps VS Code's shortcuts by default. On first use it offers to switch Unreal projects to the Visual Studio / Rider keymap; change it any time with the `unrealStudio.keybindings.visualStudioStyle` setting.

Shortcut	Action	Replaces in VS Code
`Ctrl+D`	Duplicate the line down (the caret moves to the copy), or duplicate the selections inline	Add Selection to Next Find Match

Shortcut	Action	Replaces in VS Code
`Ctrl+R, Ctrl+R` (or `Ctrl+R R`)	Rename a symbol across headers and sources, or a file in the Solution Explorer	Open Recent
`F5` / `Shift+F5`	Attach to Unreal Editor / stop	—
`F9` / `Ctrl+Shift+F9`	Toggle breakpoint / clear all breakpoints	—
`Alt+Shift+F12`	Fast indexed references (Pro)	—

`F2` and right-click › **Unreal: Rename Symbol** always work, whichever keymap you use.

Build.cs and Target.cs

- Inside `PublicDependencyModuleNames`, `PrivateDependencyModuleNames`, `DynamicallyLoadedModuleNames`, `ExtraModuleNames`..., typing `"` suggests every engine, plugin, project and ThirdParty module, and shows where each one lives. Modules already listed are left out. If the module's plugin isn't enabled yet, picking it also enables the plugin in the `.uproject` (or lists it in the `.uplugin`).
- `ModuleRules` and `TargetRules` members (`PCHUsage`, `bUseUnity`, `OptimizeCode`, `DefaultBuildSettings`...) complete with a snippet, and enum values complete after `PCHUsageMode.`, `CodeOptimization.`, `TargetType.`... Names and docs come from your engine's UnrealBuildTool sources.
- Hover shows the docs. `Ctrl+Click` on a module name opens its `Build.cs`.

Config files: .uproject, .uplugin and .ini

- JSON Schemas and completions for `.uproject` and `.uplugin` descriptors: modules, loading phases, target platforms, and every installed engine, Fab and project plugin with its description. Hover describes a plugin; `Ctrl+Click` opens its `.uplugin`.
- For `Config/*.ini` (`DefaultEngine.ini`, `DefaultGame.ini`, `DefaultInput.ini`, `DefaultEditor.ini`): a dedicated grammar (`unreal-ini`), semantic coloring, hover documentation, smart completions and `Ctrl+Click` Go to Definition on class paths (`/Script/Module.ClassName`).

AI agents

Give AI agents what only UStudio knows about your Unreal project, through a local Model Context Protocol server.

![Model Context Protocol][chip-mcp] ![GitHub Copilot][chip-copilot] ![Cursor][chip-cursor] ![Claude Code][chip-claude] ![Codex][chip-codex]

MCP server PRO

UStudio runs the server while the project is open and registers it automatically in VS Code (Copilot agent mode) and Cursor. For Claude Code, Codex, Windsurf and other clients, **UStudio: Connect AI Agents** copies the configuration. For Codex, paste the copied block into `~/.codex/config.toml`; the Codex CLI, IDE extension and ChatGPT app share it.

Tools	What the agent gets
`build`, `live_coding_compile`	Compiles and gets the structured compiler / linker errors back, so it can fix them and try again
`get_problems`, `inspect_file`	The Problems panel and UStudio's Unreal inspections
`find_references`, `find_blueprints_using`, `read_blueprint_as_cpp`, `get_class_lifecycle`	C++ and Blueprint usages, Blueprints read as C++, and how a class is created and driven
`find_header_for_type`, `add_module_dependency`	The right `#include`, and the `Build.cs` / plugin entries a type needs
`get_unreal_log`, `list_tests`, `run_tests`	The Unreal Editor log, and Automation tests with their failures
`list_gameplay_templates`, `create_gameplay_template`	Modular C++ gameplay templates (Characters, Projectile / AoE / Dash / Melee abilities, Attributes, Status Effects, Interactables)
`list_content_asset_presets`, `inspect_cpp_class_properties`, `create_content_asset`	`.uasset` files in `Content/` (GameInstance, GameMode and sub-assets, UMG widgets, Enhanced Input, Actors, Components, DataAssets, DataTables)
`list_unreal_python_templates`, `create_unreal_python_script`, `run_unreal_python_script`, `setup_unreal_python_intellisense`	Unreal Editor Python scripts, generated and run (UE 5.0 – 5.8)
`nodegraph_catalog`, `nodegraph_create`, `nodegraph_edit`, `nodegraph_read`, `nodegraph_generate_code`, `nodegraph_compile`, `nodegraph_run`, `nodegraph_get_execution`, `nodegraph_convert_to_code`, `nodegraph_export_pdf`, `custom_node_create`, `nodegraph_set_theme`	Node Graphs: build C++ / Python classes and custom nodes from nodes (wired by pin name), use other graphs' and C++ classes' members, set Class Defaults and component properties, compile them into Unreal Editor with the errors on their nodes, read which nodes ran, convert them to code

The server only listens on `127.0.0.1`, requires a per-machine token and rejects browser requests.

Cursor support

UStudio runs in Cursor as well as VS Code. Install it from Cursor's extension marketplace (Open VSX) or with `cursor --install-extension ustudio-<version>.vsix`.

- **IntelliSense:** clangd, as in VS Code. Cursor's own C/C++ extension also uses clangd, so the two work together, and UStudio finds the clangd that Cursor downloaded.
- **Debugging:** Microsoft's C++ debugger only runs in VS Code, so in Cursor UStudio sets up **CodeLLDB** (`vadimcn.vscode-lldb`) and loads Unreal's LLDB data formatters (`FString`, `FName`, `TArray`, `TMap`…). If no debugger is installed, UStudio offers to install it. Its launch and attach configurations have "(UStudio, LLDB)" in their names.
- **Live Bridge:** double-clicking a class in Unreal Editor opens it in the editor that is running UStudio for the project (VS Code or Cursor). This needs plugin v3; UStudio offers to update older copies.
- **Predictive autocomplete:** off by default in Cursor so it doesn't compete with Cursor Tab. Turn it on with `unrealStudio.predictiveCode.enabled`.

Free vs Pro

Every install starts with a **14-day Pro trial** with every feature. When the trial ends, UStudio keeps working in Free mode, forever.

Feature	Free	Pro
Code intelligence		
Automatic IntelliSense and `clangd` setup	✅	✅
Predictive autocomplete (ghost text, `Tab`)	✅	✅
"used by N" above functions: C++ usages	✅	✅
"used by N" above functions: Blueprint usages	—	✅
`#include`, `Build.cs` dependency and LNK2019 quick fixes	✅	✅
Format Document, Unreal style (braces on a new line or the same line)	✅	✅
Symbol and file renaming (`Ctrl+R, Ctrl+R`, right-click)	✅	✅
Duplicate line or selection (`Ctrl+D`)	✅	✅
Real-time `.generated.h` sync (RPCs and macros)	—	✅
`Ctrl+.` implementation generator	—	✅
Implement interfaces (right-click, `Ctrl+.`) and interface IntelliSense	—	✅
Smart `F12` navigation (`.h` ↔ `.cpp` and RPCs)	—	✅
Fast indexed Find References (`Alt+Shift+F12`) with search history	—	✅
Real-time Unreal inspections with quick fixes	—	✅
Build and debug		
Build / Rebuild / Launch Editor toolbar	✅	✅
One-click launch in Debug mode	✅	✅
Build errors in the Problems panel	✅	✅
Error fix suggestions with multi-file preview	✅	✅
Automation tests in the Testing view	—	✅
Live Unreal Editor		
Unreal Editor Live Bridge (UE 5.0 – 5.8)	✅	✅
Live Coding from VS Code (`Ctrl+Alt+F11`)	✅	✅
Unreal Log panel and Build Errors panel	✅	✅
Live editor status in the dashboard and Solution Explorer	✅	✅
Live Execution: C++ functions light up while Unreal Editor runs them	—	✅
Blueprints and assets		
Blueprints / `.uasset` as C++ classes	Summary	✅
Convert Blueprints to C++ classes and replace their references (experimental)	—	✅
Blueprint / `.uasset` CodeLens	—	✅
Derived Blueprints of each C++ class (direct and indirect)	—	✅

Feature	Free	Pro
Class Lifecycle diagram	—	✅
Node Graphs		
Node Graph editor for C++ and Python classes, live code preview, custom nodes, appearance	✅	✅
Compile into Unreal Editor, Compiler Results, live debugging, Convert to Code, Export to PDF	—	✅
Wizards and templates		
Visual C++ class wizard	—	✅
Modular gameplay templates (characters, abilities, attributes, status effects, interactables)	—	✅
Your own C++ and Python templates (personal or shared with the team)	—	✅
Content `.uasset` wizard (`GameInstance` , `GameMode` and sub-assets, UMG widgets, input, DataAssets)	—	✅
New Module / New Plugin wizards	—	✅
Python for Unreal Editor		
Python Scripting Studio (IntelliSense, templates and live runner)	—	✅
Python debugger for Unreal Editor scripts	—	✅
Source control and history		
Git and Perforce status, Perforce file badges (P4V-style), read-only badges	✅	✅
Local file history and code authors (Git, GitHub, Perforce)	✅	✅
Change Timeline: every change by date (local, Git, Perforce), diffs, search and export	✅	✅
Change Timeline Apply: put files back the way they were at any point (Perforce-aware)	—	✅
Perforce changelist panel: check out, shelve and submit with a description	—	✅
Navigation and workspace		
Solution Explorer with auto-tracking and custom icons	✅	✅
Solution Explorer search: file names and contents (`Ctrl+;`)	✅	✅
Favorites view (files and folders)	✅	✅
Members view with search (`Alt+M`)	✅	✅
Panels as editor tabs, layout presets and reset	✅	✅
Export folders and files to PDF (structure, paths and contents)	—	✅
`Build.cs` , `Target.cs` , `.uproject` , `.uplugin` and `Config/*.ini` IntelliSense	✅	✅
Two-phase background engine indexer and dashboard	✅	✅
AI agents		
MCP server for AI agents (Copilot, Cursor, Claude Code, Codex)	—	✅

Pricing

- **One-time purchase.** A Pro license includes one year of updates and unlocks up to 3 machines for your own use. **Get UStudio Pro**
- **Yours to keep.** After that year, Pro keeps working on every version released during it.
- **Optional update renewal (US$15)** adds another year of updates. Renewing early is fine: the new year starts when the current one ends.

Activating a license

- Click the license item in the status bar, or run **UStudio: Manage License & Subscription** from the Command Palette, and enter your key.
- **Renewing updates:** choose **Extend Updates by 1 Year** (or **Renew Updates** when your license doesn't cover the installed version), then enter the renewal key from the email with **Enter License or Renewal Key**. A renewal key extends the Pro license activated on that machine; it doesn't unlock Pro on its own.

Requirements

- **Unreal Engine:** 5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 or 5.8.
- **Editor:** VS Code 1.85.0 or newer, or Cursor.
- **Platform:** Windows 10 / 11 (64-bit).

Privacy

- **No telemetry or analytics.** UStudio never sends your code, project files or usage data anywhere.
- **License checks (Pro):** activating, re-validating (about once a week) or deactivating a license contacts the Lemon Squeezy license API (`api.lemonsqueezy.com`) with your license key and an activation name made of your computer name, the editor name (VS Code / Cursor) and the first 8 characters of the editor's anonymous machine ID: the name you see when managing activations in the customer portal. Trial and license state are stored locally by the editor.
- **Local connections only:** the Live Bridge (Unreal Editor ↔ VS Code / Cursor), the Unreal Editor plugin's Live Coding endpoint and the MCP server (Pro) listen on `127.0.0.1` and require a local token.
- **Files written:** bridge discovery files (`.vscode/.unreal-studio-bridge.json`, `Saved/UStudio/editor.json` and `%LOCALAPPDATA%\UnrealStudioWorkflow\bridge.json`), index caches in the project's `.vscode` folder and engine caches in your user profile.

Support

- **User guide:** UStudio Pro 1.22.0 User Guide (PDF).
- **Discord:** discord.gg/4f6VpWpzp5 for help, bug reports, feature requests and release news.
- **License and invoices:** run **UStudio: Manage License & Subscription** › *Manage License & Invoices*, or open the customer portal.

License

Commercial proprietary software. See `LICENSE.txt` for the full End User License Agreement (EULA).

Unreal and Unreal Engine are trademarks of Epic Games, Inc. UStudio is an independent extension and is not affiliated with Epic Games. Other product names and logos belong to their respective owners.

Release notes 1.15.2 – 1.22.0

UStudio 1.22.0 (2026-11-07)

Added

- **Breakpoints pause the game, like Blueprints.** In Play in Editor, a node with a breakpoint (`F9`) pauses the game right there, with no debugger attached.
  - The graph highlights the node, and a bar shows the paused object with **Continue (`F5`)**, **Step (`F10` , to the next node)** and **Stop**. Unreal Editor's own Resume works too.
  - Only the instance picked in the debug list stops. Code on worker threads never pauses.
  - **Breakpoints** on the toolbar (and the `unrealStudio.nodeGraph.pauseOnBreakpoints` setting) turns pausing on and off.
  - When a C++ debugger is attached and really stops, it takes over.
- **The Live Bridge plugin is v7** (breakpoints, Content Browser). Projects ask to update it once; build with Unreal Editor closed.
- **Members of engine classes** from a reference: drag a wire from a Character Movement pin and **Get Velocity**, Is Falling, Get Max Speed, Set Movement Mode, Max Walk Speed, Add Impulse... are there, with everything inherited, like Blueprint.
  - UStudio reads the classes of the engine the project uses and its enabled plugins, in the background once (about 4 s), then from a cache.
  - Engine events of a reference have **Bind Event to / Unbind / Unbind all / Is Bound** (On Component Hit, On Destroyed...). Binding creates the matching custom event.
  - Engine function library functions (Vector Length, Break Hit Result...) appear when you search for them or drag from a pin that fits.
  - Compiling adds the engine modules they need to the Build.cs.
- **Async action nodes, like Blueprint's latent nodes,** from the project's and its plugins' C++ (e.g. SteamCorePro's **Create Session**, Find Sessions, Join Session). They have **On Success / On Failure** outputs with their data. The generated code binds every result and calls `Activate()` .
- **Structs, enums, maps, sets, soft references, interfaces and delegates** as pin and variable types. **Make / Break** nodes for the project's structs, and enum values as dropdowns.
  - In a large project the public functions UStudio can use went from 948 to 1648 (of 1649).
  - Functions marked `BlueprintInternalUseOnly` are hidden, as in Blueprints.
- **Asset and class pickers, like Unreal's.** Asset pins, asset variables, soft references, Class Defaults and component properties get a searchable list of the project's, its plugins' and the engine's assets of the right type. **Use Selected** and **Browse to Asset** work with Unreal Editor's Content Browser.
  - Class pickers list the project's C++ classes, Node Graph classes, engine classes and Blueprints, filtered by the base class.
  - Levels for level references and Open Level.
  - A Blueprint class on Spawn Actor from Class gives an output of its C++ parent class.
  - A class typed by hand that doesn't exist is reported in Graph Check.
- **Timelines:** the Blueprint Timeline node (C++ actors).
  - Inputs: Play, Play from Start, Stop, Reverse, Reverse from End, Set New Time.
  - Outputs: Update, Finished, Direction, and float, vector, color and event tracks.
  - Settings: Length, Autoplay, Loop, Ignore Time Dilation.
  - It's generated as a Timeline Component with its curves. A track can play a curve asset instead.
- **Curve editor:** keys, tangents, Auto / User / Break / Linear / Constant interpolation, presets (ease in / out, bounce...), several channels, zoom and fit.
- **Float Curve variables** with **Evaluate Curve**.
- **Color picker** for color pins, variables and Class Defaults: hue / saturation, value, alpha, RGB, HSV and hex (sRGB and linear).

Fixed

- **Breakpoints never stopped when attaching the debugger to an open Unreal Editor.** The C++ debugger only loads symbols for its first 500 modules, and a project's modules come after those. Node Graphs now pause without a debugger, and offer **Launch with Debugger** instead of attaching.

UStudio 1.21.0 (2026-11-05)

Added

- **Debugging Node Graphs, like Blueprints:**
    - **Debug instance picker.** While debugging, a list next to Debug shows every object of the class that ran ("All instances" or one of them), named by its actor label. Only the chosen one lights up. It works for C++ (including code on worker threads) and Python graphs.
    - **Breakpoints on nodes.** Press `F9` or use the node's right-click menu. A red dot marks the node, and the breakpoint goes on the node's line in the compiled code. When the C++ or Python debugger stops there, the node is highlighted ("Paused at a breakpoint"). Breakpoints are saved per project, not in the `.ugraph`.
    - **Debug compiles by itself:** turning Debug on compiles the class for debugging when needed. After restarting VS Code it picks up the compiled copy without compiling again.
    - **Clear messages** when debugging can't work: the project's Live Bridge plugin is older than v6 (with an **Update Plugin** button), Unreal Editor isn't running, or the class isn't compiled.
- **The Live Bridge plugin is v6:** it counts the runs of each object for the instance picker. Projects with v5 are asked to update it once; build with Unreal Editor closed.
- **Read only while compiling:** during Compile and Convert to Code the graph is greyed out with a "(read only)" banner. You can still look around, zoom and read the code, but you can't change it.
- **Inherited components:**
    - **Components** shows the components the parent class already has, with their hierarchy and an "(inherited)" mark. In a Character that's CapsuleComponent > ArrowComponent, Mesh, plus CharacterMovement. It works for engine parents (Character, Pawn, Default Pawn, Static / Skeletal Mesh Actor, Camera Actor, controllers...), the project's C++ classes (read from their `.h` / `.cpp`) and other graphs.
    - Your components attach to them, and a parent's root is never replaced by accident.
    - You can drag them to the graph (Get Mesh), use their events (On Component Begin Overlap of the capsule) and set their properties in Details. These are written in the constructor through `GetMesh()`, `GetCapsuleComponent()` ...
    - Graph Check reports a component or variable named like an inherited one, and ArrowComponent used in the game (it only exists in the editor).
- **Thread nodes:**
    - **Run on Game Thread** (back to the game thread, skipped if the object was destroyed meanwhile).
    - **Run on Background Thread** (normal or high priority).
    - **Run on Thread Pool** (thread pool, task graph or a dedicated thread).
    - **Parallel For**, **Sleep** and **Is In Game Thread**.

  Bodies copy the values they read. Graph Check warns when a node that needs the game thread (spawning, Print String, moving actors...) runs on a background thread, and when Sleep would freeze the game.
- **Copy and paste nodes as JSON** through the system clipboard: `Ctrl+C` / `Ctrl+X` / `Ctrl+V`, or **Copy as JSON** / **Paste** in the menus. Paste into another graph, another window, a chat or a file. Nodes that use variables the graph doesn't have offer to create them.

- **The wire stays attached to the node list** when you drop it on empty space, until you pick a node. It follows zoom and pan.
- **AI agents (MCP):** `nodegraph_read` lists the inherited components, and the edit operations can attach to and set them.

Fixed

- **Code generated for UE 5.0 - 5.4 did not compile** in some cases:
    - Net Update Frequency, Min Net Update Frequency and Net Cull Distance in Class Defaults used functions that only exist since UE 5.5.
    - Skeletal Mesh Asset used a function that only exists since UE 5.1.
    - Delays, timers and async nodes used a lambda capture that C++17 (UE 5.0 - 5.2) rejects.

  The generated code now picks the right form for the engine version.
- **A local value named like a member variable hid it** in the generated C++, which Unreal treats as an error.

UStudio 1.20.0 (2026-11-05)

Added

- **Node Graphs use each other's classes, like Blueprints:**
    - The classes of every `.ugraph` of the project work as variable and parameter types, in Cast To, as the parent of another graph, and on class pins. They don't need to be converted to code first.
    - Drag a wire from an object pin and its class's functions, custom events, variables and event dispatchers come first in the list. This works for another graph's class or a C++ class of the project.
    - The new nodes have a **Target** pin: call a function, Get / Set a variable, Bind / Unbind / Call another class's dispatcher.
    - Renaming a function or variable in its graph doesn't break the graphs that use it. Deleting it shows an error on the nodes that used it.
    - **F12** on these nodes opens the other graph at the member, or the C++ header.
- **Compiling with dependencies:**
    - Compiling a graph first writes the compiled copy of the graph classes it uses. This includes the ones they use, class defaults (a Game Mode's Default Pawn Class) and class variables.
    - Compiling or converting a class rewrites the compiled code of the graphs that use it, so they follow renames and switch to its final header.
    - Includes are worked out for each file: from Public, from Private within the same module, or relative to the file in modules without Public / Private.
    - A class from another module adds that module to the Build.cs. When that class is only compiled (not converted), Graph Check says so.
- **Your C++ classes and plugins as nodes:**
    - The public `UFUNCTION`s and `UPROPERTY`s of the project's C++ classes (`Source` and `Plugins`) are read from their headers, with reference parameters as outputs and defaults on the pins.
    - Static functions of function libraries are always in the list.
    - A class from a plugin module adds that module to the Build.cs.
    - A `WorldContextObject` left unconnected is self, like in Blueprints.
- **Game framework nodes:** Get Game Instance, Get Game Mode, Get Game State, Get Player State, Get Player Camera Manager, Get Controlled Pawn, Get Controller, Get HUD, Possess, Un Possess.

- **Subsystem nodes:** Get Game Instance Subsystem, Get World Subsystem, Get Local Player Subsystem, Get Engine Subsystem and, in Python, Get Editor Subsystem. Choose the class on the pin (yours or the engine's) and the output has that type.
- **Spawning nodes:**
    - Sound: Play Sound at Location, Play Sound 2D, Spawn Sound at Location / Attached.
    - Effects: Spawn System at Location / Attached (Niagara, which adds Niagara to the Build.cs), Spawn Emitter at Location (Cascade) and Spawn Decal at Location.
    - User interface: Create Widget, Add to Viewport, Remove from Parent.
    - Objects and components: Construct Object from Class and Add Component by Class.
- **Spawn Actor from Class gives the class you pick** (`SpawnActor<ACoin>`), like Blueprints. Class pins have a class picker.
- **Assets on pins and variables.** Sound, Niagara, particle, material, mesh, texture and data asset pins take a path (`/Game/Audio/Hit`). Asset variables of actors and components get their default in the constructor (`ConstructorHelpers`).
- **Class Defaults** (toolbar), like the Blueprint panel:
    - The default value of each variable.
    - The inherited properties of the parent class, set in the constructor when they differ from the engine's value:
        - replication (Replicates, Replicate Movement, Always Relevant, Net Update Frequency, Net Priority, Net Cull Distance, Net Dormancy);
        - tick (Can Ever Tick, Tick Interval, Tick Group);
        - actor (Initial Life Span, Spawn Collision Handling, Auto Receive Input);
        - pawn (Auto Possess Player / AI, AI Controller Class, Use Controller Rotation);
        - Character Movement (Max Walk Speed, Jump Z Velocity, Air Control, Orient Rotation to Movement, Rotation Rate, Can Crouch), and the character's capsule, mesh asset, mesh transform and Anim Class;
        - the Game Mode's classes (Default Pawn, Player Controller, Game State, Player State, HUD, Spectator);
        - the Player Controller's mouse settings;
        - component replication and activation.

  Changed values are highlighted, with an arrow to reset them.
- **Component properties** in the component's Details: Location / Rotation / Scale, visibility, Static Mesh, Skeletal Mesh, Material, Collision Presets, Generate Overlap Events, Simulate Physics, Box Extent, Sphere Radius, Capsule size, text, light, sound, spring arm, camera, projectile and rotating movement settings.
- **AI agents (MCP):**
    - `nodegraph_catalog` with `class` lists a class's members as nodes, and lists the classes of the project's graphs.
    - `nodegraph_edit` adds `memberCall` / `memberGet` / `memberSet` / other classes' dispatchers by class and member name, plus `setClassDefault` and `setComponentProperty` .
    - `nodegraph_read` shows the Class Defaults and component properties.

Every generated construct was built with Unreal Engine 5.7: graphs using a converted class and temporary ones, inheritance from a graph class, a game instance subsystem, Niagara, UMG, the Class Defaults of an actor, a character and a game mode, component properties, and a C++ function library with reference outputs.

UStudio 1.19.1 (2026-11-05)

Added

- **New Node Graph wizard.** Right-click any folder (Explorer, Solution Explorer, Favorites) › **New Node Graph (C++ / Python)...** opens a wizard. It has cards for C++ class, Python class or custom node and shows the parent class as cards with a search (engine and project classes). You type the name and see a preview (`ACoinCollector ·`

`CoinCollector.ugraph`). The module and code folder come from the folder you clicked, and you pick the starting events and where the graph is saved. **UStudio: New Custom Node...** opens the same wizard on the custom node card.

- **Arrange nodes like in Unreal:**
  - `Q` straightens wires.
  - `Shift+W` / `Shift+S` / `Shift+A` / `Shift+D` align top / bottom / left / right.
  - `Alt+Shift+W` / `Alt+Shift+S` align middle / center.
  - `L` arranges the selection, or the whole graph: exec chains left to right, branches below, pure nodes before what reads them, and comments follow their nodes.
  - Arrow keys move the selection one grid step (`Shift`: four).

  These are also in the right-click menu (**Alignment**, with Distribute Horizontally / Vertically) and under **Arrange** on the toolbar.

Fixed

- **Right-click on the graph showed VS Code's Cut / Copy / Paste menu along with the node list.** Now only the node list appears. Text fields keep their usual menu.
- **Pins of types that don't fit could be connected** (an Actor into a Branch's Condition). Pins are now checked like in Blueprints. They connect only when they have the same type, when Blueprint converts between them automatically (numbers, values to string, string / name / text), or when the object is of the pin's class or a child of it. A parent class needs a **Cast**, and the message says so. Wires that don't fit in graphs saved before this fix are reported in Graph Check.

UStudio 1.19.0 (2026-11-05)

Added

- **Node Graphs: C++ and Python classes made with Blueprint-style nodes. UStudio: New Node Graph Class** (or **Node Graph** in the UStudio panel, or right-click a folder in the Explorer) creates a `.ugraph` file: pick C++ or Python, the parent class (Actor, Pawn, Character, components, User Widget, controllers, Game Mode, Game Instance, subsystems, Anim Instance, Function Library, Data Asset or a project class; Python: an editor tool, Object, Actor, Component or Function Library) and a name. The graph is a file of the project: `Ctrl+S`, undo / redo, Git and Perforce work as usual.
  - **Editor.** A canvas like the Blueprint editor: the searchable node list (right-click or `Space`; drop a wire on empty space to see the nodes that fit, wired for you), wires by dragging pins, `Alt+click` to break, reroutes (double-click a wire), comments (`C`), copy / paste / duplicate, box select, zoom and pan. A **My Class** panel with variables, functions, components and event dispatchers (drag them to the canvas), a **Details** panel and a **Code** tab with the `.h` / `.cpp` or `.py` as you edit: the selected node's lines are highlighted and clicking a line finds its node.
  - **Nodes.** The events of each parent class; Branch, Sequence, For / For Each / While, Switch on Int, Do Once, Do N, Gate, FlipFlop; Cast; variables, functions and custom events (with replication: Server, Client, Multicast); math, vectors, rotators, strings and arrays; actor and component functions; Delay, Retriggerable Delay, Set Timer / Clear Timer / Is Timer Active, **Async Task** (a background thread, then back on the game thread) and **Async Load Asset**; Unreal Editor scripting for Python; **Custom Code** and **Custom Expression** for any C++ or Python.
  - **Event dispatchers and delegates.** Dynamic multicast (Blueprint-assignable), multicast, dynamic and single delegates, with Call, Bind, Unbind, Unbind All and Is Bound nodes; **Create a matching event** makes the custom event to bind. Select a component and its events (**On Component Begin Overlap**, End Overlap, Hit) appear in Details with **+** buttons, like in Unreal: the handler has the engine's signature and is bound in `BeginPlay`.
  - **Compile (`F7`, Pro).** Checks the graph and builds a temporary copy of the class into the running Unreal Editor, ready to use without converting it: C++ through Live Coding (or a build when Unreal Editor is closed; the copy goes in `Source/<Module>/Private/UStudioGraphs`), Python loaded live through Remote Execution (**Run** runs an

editor tool). **Compiler Results** shows the graph checks and the compiler's errors and warnings on the nodes that produced them.
    - **Live debugging (Pro).** With **Debug** on, the nodes that run light up while the game or the tool runs, exec wires flow and each node counts its runs, like the Blueprint debugger (C++ through the Live Bridge plugin's `USTUDIO_TRACE`, set up on first use).
    - **Convert to Code (Pro).** Writes the final `.h` / `.cpp` into the module (or the `.py` into `Content/Python`) and removes the temporary copy; the class keeps its name, so Blueprints made from it keep working. Compiling again updates that code while nobody edited it.
    - **Custom nodes.** Your own `.unode` nodes, saved in the project (`.ustudio/nodes`) or for you (`~/.ustudio/nodes`), made of sub-nodes (like Blueprint macros), of C++ / Python code (`{Input}`, `{$Output} = ...`) or both, with a category, color, icon or image. **Collapse to Custom Node** (right-click a selection) makes one from existing nodes. **UStudio: New Custom Node** creates an empty one.
    - **Navigation.** Back / Forward (`Alt+←` / `Alt+→`, mouse buttons), **Find** (`Ctrl+F`) across every graph, **Go to Definition** (`F12`: a function's graph, a variable, a custom node or the engine header of a C++ call), **Show in Code**, and **Show in Node Graph** / CodeLens in the generated code to go back to the node.
    - **Export to PDF (Pro).** Every graph as drawn, in vector with selectable text, then the generated code, with bookmarks. Also **UStudio: Export Node Graph to PDF** and right-click a `.ugraph` / `.unode`.
    - **Appearance.** The **Style** tab: Unreal (default), VS Code and Light presets; background color, grid and image; node body, border, corners, opacity and font size; header colors per kind of node, pin colors per type; wires curved, straight or right-angled, their thickness and colors. A node's own color, icon and note in its Details; a wire's own color from its right-click menu. Saved in `unrealStudio.nodeGraph.theme`.
    - **AI agents (MCP, Pro).** `nodegraph_list`, `nodegraph_catalog`, `nodegraph_read`, `nodegraph_create`, `nodegraph_edit` (variables, functions, components, dispatchers, nodes and wires by pin name, laid out for you), `nodegraph_write`, `nodegraph_generate_code`, `nodegraph_compile`, `nodegraph_run`, `nodegraph_get_execution`, `nodegraph_convert_to_code`, `nodegraph_export_pdf`, `custom_node_create` and `nodegraph_set_theme`.
    - Every generated C++ construct was compiled with Unreal Engine 5.7 (actors with components, replication, RPCs, delays, casts, loops, custom nodes, the four kinds of delegate, component events, async tasks, timers, async loads, Do Once / Gate / FlipFlop / Do N, debug builds).
- **User guide (PDF)** for 1.19.0, with a Node Graphs chapter.

UStudio 1.18.1 (2026-11-05)

Fixed

- **A ghost implementation that goes after the last function of a `.cpp` showed up above its closing `}`** when that `}` was the last line of the file (no newline at the end). VS Code can only draw a CodeLens row above a line, so this ghost now appears after the `}`, on the same line: `} PostEditChangeProperty(…) Implement`. Hover it to see the whole code with **Implement** links, or Ctrl+Click **Implement**. The code is written below the `}`, and the file then ends with a newline, so the next ghosts appear as usual.

UStudio 1.18.0 (2026-11-04)

Added

- **Export to PDF (Pro).** Right-click a folder or a file in the Explorer, the Solution Explorer, Favorites or an editor tab › **Export to PDF**, or run **UStudio Pro: Export to PDF (Structure & Contents)…** and pick them. Select several to put them in one PDF.
    - **Structure:** the folder tree with each file's size and the page where it starts. Rows are links, and the PDF's bookmarks follow the folders.
    - **Contents:** each text file under a header with its path, name, line count and size, with line numbers and syntax colors (C++, C#, Python, JSON, `.ini`). Long lines wrap with an indent; tabs follow `editor.tabSize`. Unsaved changes are exported as they appear in the editor. Choose **Structure only** to leave the contents out.

    - **Fonts:** the PDF embeds the system fonts (Consolas and Segoe UI on Windows, Menlo and Arial on macOS, DejaVu on Linux), with only the characters it uses. Characters they lack (Chinese, Japanese, Korean, emoji, symbols) come from the CJK and symbol fonts installed. The text can be searched and copied. `unrealStudio.exportPdf.fonts: builtin` uses the PDF built-in Courier / Helvetica instead (smaller, Western European only), and `unrealStudio.exportPdf.codeFontFile` picks another font for the code.
    - **Left out:** `Binaries`, `Intermediate`, `Saved`, `DerivedDataCache`, `.git`, `.vs`, `.idea`, `node_modules` and `__pycache__` (`unrealStudio.exportPdf.exclude`), and what `files.exclude` hides, unless you export that folder itself. Binary files (`.uasset`, `.umap`, images, DLLs...) and files over 1 MB (`unrealStudio.exportPdf.maxFileSizeKB`) are listed without contents. Files are read as UTF-8, UTF-16 or Windows-1252.
    - Also `unrealStudio.exportPdf.pageSize` (A4 or Letter), `fontSize`, `lineNumbers` and `syntaxColors`. Exports of more than 30 MB of text ask first.

UStudio 1.17.0 (2026-10-31)

Added

- **Convert Blueprints to C++ (Pro, experimental).** Right-click a Blueprint `.uasset` of an actor, component or object › **Convert to C++ Class (Experimental)...** (Explorer, Solution Explorer, Favorites, the title bar of its C++ view, or **UStudio: Convert Blueprint to C++ Class** in the Command Palette). A wizard shows what the class will contain, lets you leave members out and previews the `.h`, the `.cpp`, the `DefaultEngine.ini` redirects and the assets that reference the Blueprint.
    - **Step 1 writes the C++ class.** Components with their hierarchy, sockets and settings (static and skeletal meshes, materials, anim class, collision profile, physics, transforms, Niagara systems, child actor classes...), inherited components the Blueprint changed (`GetCharacterMovement()->MaxWalkSpeed`, `GetMesh()`...), variables with their defaults, specifiers, categories, tooltips and replication, event dispatchers, functions, custom events, Server / Client / Multicast RPCs, `GetLifetimeReplicatedProps`, overridden events (`BeginPlay`, `Tick`, `EndPlay`, `NotifyHit`, `OnConstruction`...), `BlueprintNativeEvent` and C++ interface implementations, and component events bound in `BeginPlay`. Names with spaces become identifiers (`Max Speed` → `MaxSpeed`) and keep their display name. The code is checked against your engine's headers: protected members are assigned, private ones go through their setter (`SetHidden`, `SetChildActorClass`, `SetIsReplicatedByDefault`...), and APIs match your engine version. The modules it uses are added to the `Build.cs`.
    - **Step 2 replaces the Blueprint** (Unreal Editor closed): UStudio builds the project, moves the Blueprints that derive from it onto the new class, keeping what they changed on inherited components and their components attached to them, adds `[CoreRedirects]` to `Config/DefaultEngine.ini` (class, default object, renamed variables, functions and parameters), has a headless Unreal re-save every asset that referenced the Blueprint, and deletes it. Actors placed in levels keep their position, components and overridden values. A copy of the Blueprint is kept in `Saved/UStudio/BlueprintBackups`. Under Perforce the assets are checked out and the Blueprint is marked for delete. If anything fails before the references are updated, the redirects are removed again. **UStudio: Finish Blueprint to C++ Conversion** resumes a conversion later (after fixing a build error, for example).
    - Circular references between Blueprints are fine: a variable typed as another Blueprint uses its C++ parent. A default that points to a Blueprint deriving from the converted one is not loaded from the constructor (that would be a circular dependency) and is listed in the notes.
    - **Graph logic is not translated**: each function and event gets a body with a `TODO` to port it, and the wizard warns when the Blueprint has logic. Timelines, Blueprint structs and enums and Blueprint interfaces are listed as not converted. Widgets, Anim Blueprints, levels, materials and data tables are not converted, and a Blueprint whose parent is another Blueprint asks you to convert that one first.
    - Tested on UE 5.7 with about 1,000 Blueprints from 7 projects (every generated class compiles) and full conversions of placed, child and referencing assets.
- **User guide (PDF)** for 1.17.0, linked from the README.

UStudio 1.16.0 (2026-10-30)

Added

- **Ghost implementations in the `.cpp`.** Every function or event the header declares without an implementation appears in the `.cpp` where it belongs, as a ghost of its code (`void AMyActor::Tick(float DeltaTime) { Super::Tick(DeltaTime); }`), with an **Implement** button that writes it there and puts the cursor in the body. Writing them is Pro; the Free edition shows the ghosts.
  - Each ghost goes after the function declared before it in the `.h`, else before the next one, else at the end of the file, so the `.cpp` follows the header's order.
  - `BlueprintNativeEvent`, `Server`, `Client` and `NetMulticast` functions appear as `_Implementation`, with their `_Validate` for `WithValidation`.
  - No ghost for `BlueprintImplementableEvent`, pure virtuals (`= 0`, `PURE_VIRTUAL`), templates, `= default` definitions, or functions with a body in the header (inline or after the class) or in another `.cpp` of the project.
  - Declarations under `#if WITH_EDITOR` keep their `#if`, or go inside the `.cpp` 's own `#if WITH_EDITOR` block. Other functions never land inside one.
  - A declaration typed in the `.h` shows up in the `.cpp` beside it as you type, before you save.
  - Several ghosts in the same place share a row: **Implement all**, or one **Implement** button per function. **UStudio: Implement Missing Functions from the Header...** in the command palette, or the right-click menu of the `.cpp`, lets you choose among all of them.
  - Turn them off with `unrealStudio.codeLens.ghostImplementations`.

Changed

- Constructors that take a `const FObjectInitializer&` are generated with `: Super(ObjectInitializer)` (with `Ctrl+.` in the `.h` and with the ghosts).

UStudio 1.15.5 (2026-10-30)

Added

- **Perforce file badges for every state, like P4V.** Files and folders in the Explorer, the Solution Explorer and the editor tabs now show `✓` checked out by you, `+` add, `×` delete, `→` moved, `⇔` integrate, `!` needs resolve, `↓` newer revision / deleted in the depot / not synced, 👤 checked out by someone else, 🔒 locked or exclusive (`+l`) and checked out by someone else, `✎` modified or made writable without check out, `⏸` shelved, `?` not in the depot, `∅` not mapped (outside the workspace view), dimmed for files ignored by `P4IGNORE`, and `•` on folders with changes inside. Two badges combine (`✓!`, 👤↓ ...). Hovering a file shows who has it checked out and in which changelist, and its `#have` / `#head` revisions. Before, only your checked-out files, newer revisions and edits without check out were marked.
  - UStudio asks the server about a folder when its files appear on screen (`p4 fstat`, `p4 add -n` preview, `p4 where`), and again when you come back to VS Code and at each status check.
  - `unrealStudio.perforce.fileDecorations` : `all` (default), `changes` (only your changes, out-of-date and locally modified files, no per-folder queries) or `off`. Colors: `ustudio.perforce.*` in `workbench.colorCustomizations`. **What the Perforce file badges mean** in the P4 status bar menu lists them all.
  - In a Perforce workspace the read-only 🔒 badge gives way to the Perforce badges (read-only means not checked out), so 🔒 now means a Perforce lock.
- **Your own C++ and Python templates (Pro).** Create them from scratch with **New Template**, from what you're looking at with **Save as Template** (step 3 of the class wizard, or the code in Python Studio), or right-click a `.h` / `.cpp` / `.py` file in the Solution Explorer, the Explorer or the editor > **Save as Template**. A class becomes a template on its own: its name, API macro, own `#include` and `.generated.h` turn into placeholders, and the `.h` and `.cpp` are read together.
  - Placeholders in C++: `{{ClassName}}`, `{{BaseName}}`, `{{API}}`, `{{Module}}`, `{{Header}}`, `{{Project}}`, `{{Copyright}}` (from Project Settings > Description), `{{Year}}`, `{{Date}}`. In Python: `{{FileName}}`, `{{Project}}`, `{{Year}}`, `{{Date}}`. Click one to insert it.

    - Templates appear under **My Templates** in the class wizard and Python Studio, with Edit, Duplicate and Delete. Each has a class prefix, default name and folder, and can be header-only.
    - Saved as JSON in `~/.ustudio/templates` (every project, VS Code and Cursor), or in the project's `.ustudio/templates` to share them with the team through Git or Perforce.
- **Syntax highlighting in every code preview.** The class wizard, gameplay templates, Python Studio, the `.uasset` wizard (C++ summary, DataTable `.json` / `.csv`, companion header) and the New Module / Plugin wizard (`Build.cs`, `.uproject` diffs) show code with line numbers and the colors of VS Code's Dark+ / Light+ themes: Unreal macros and their specifiers, types, functions, strings, comments, `#include`. The Python Studio code and the template editors stay editable, with colors as you type and `Tab` / `Shift+Tab` indenting.

Fixed

- **Python Studio opened with no templates.** The template grid stayed empty until you typed in the search box, so step 3 had no code. It now shows the 29 templates and selects the first one.
- **Saving a read-only file showed VS Code's "Overwrite" error instead of offering to check it out.** The read-only warning only appeared when you started typing, as a notification that hides after a few seconds and appears once per file. It never appeared for changes restored when VS Code reopened, changes typed before the project finished loading, or Do Not Disturb mode. **Overwrite** removes the read-only flag without telling Perforce. Now **Ctrl+S** on a read-only file asks first, in a dialog that stays until you answer: **Check Out and Save**, **Check Out to Changelist...** or **Make Writable and Save**. With `unrealStudio.perforce.autoCheckout` set to `always`, the file is checked out and saved without asking. Saves from the File menu and **Save All** get the same dialog (one for all the files), and VS Code's error closes once the file is saved. **Ctrl+S** on an unchanged read-only file no longer tries to write it.
- The edit warning appears again after you undo all your changes, or close the file and open it again.

UStudio 1.15.4 (2026-10-30)

Fixed

- **"too many errors emitted, stopping now" when opening a header.** Since 1.15.2 the module's precompiled header also reached headers opened on their own: an engine header opened with F12 (`Actor.h`, `Array.h`...) or a header of a module with a private PCH that includes it was then defined twice. Headers no longer get the PCH; `.cpp` files keep it. UStudio adds the rule to the project's `.clangd` and writes one to the engine folder (`<engine>/.clangd`), since engine files opened from the project use its compile flags.

- `#include` **errors until the next build:**

    - A module added to a Build.cs: its headers weren't found, and once found, its `<MODULE>_API` macros were missing, so every declaration failed (`unknown type name 'ENHANCEDINPUT_API'` → too many errors).
    - A header in a `Public` folder that was empty or didn't exist at the last build.
    - The `.generated.h` of a module that was never compiled.

  UStudio now adds the include folders, the `<MODULE>_API` macros and the public definitions of the new modules when you save the Build.cs, and removes them once UnrealBuildTool knows about them.

- Clang-only errors MSVC accepts (`-Winvalid-constexpr` in engine headers, `-Wc++11-narrowing`) are no longer reported, matching the real build.

- Generated unity files (`Module.*.gen.1.cpp`) are left out of `compile_commands.json`, like the other generated sources.

UStudio 1.15.3 (2026-10-30)

Fixed

- `#if` **errors in code that compiles, until the next build.** `#if ENABLE_DRAW_DEBUG`, `#if ENABLE_VISUAL_LOG` and other macros that come from the precompiled header showed *'ENABLE_DRAW_DEBUG' is not defined, evaluates to 0* and greyed out the block until you compiled. It's an error because Unreal compiles with `/we4668`. The same happened to

a `#if` on a definition just added to `PublicDefinitions` in a Build.cs. clangd now sees what the next build will compile:
    - The PCH of each module is also read from the `.dep.json` files MSVC writes. In 1.15.2, **Fix IntelliSense** dropped it again for modules built without unity (fewer than 32 files, the usual case for game modules and plugins), until the next build.
    - A module that was never compiled gets the PCH the rest of the project uses, until its first build.
    - Definitions added to a Build.cs reach clangd when you save it, through a header in `Intermediate/UStudio/Definitions`. UStudio deletes it once UnrealBuildTool writes them to `Definitions.h`. Values computed in C# (`"X=" + ...`) are left to the build.

UStudio 1.15.2 (2026-10-30)

Added

- **IntelliSense in `Build.cs` and `Target.cs`.** Inside `PublicDependencyModuleNames`, `PrivateDependencyModuleNames`, `Public/PrivateIncludePathModuleNames`, `DynamicallyLoadedModuleNames` and `ExtraModuleNames`, typing `"` suggests every module of the engine, its plugins (Fab ones included), the project and `Engine/Source/ThirdParty`, and shows where each one lives. Modules already listed are left out. If the module's plugin isn't enabled yet, picking it also enables the plugin in the `.uproject` (or lists it in the `.uplugin` of a project plugin). Editor-only modules carry a warning when the Build.cs belongs to a runtime module.
- `ModuleRules` and `TargetRules` members (`PCHUsage`, `bUseUnity`, `OptimizeCode`, `DefaultBuildSettings`, `IncludeOrderVersion`…) complete with a snippet at the start of a statement. After `PCHUsageMode.`, `CodeOptimization.`, `TargetType.`, `BuildSettingsVersion.`, `UnrealTargetPlatform.` and the other rule enums, their values complete. Names and docs are read from your engine's UnrealBuildTool sources, so they match its version (UE 5.7: 111 module members, 285 target members). Hover shows the docs; `Ctrl+Click` on a module name opens its `Build.cs`.
- **`.uproject` and `.uplugin`: real plugin and module names.** `Plugins` › `Name` suggests every installed engine, Fab and project plugin with its friendly name and description (it was a fixed list of 19). `Modules` › `Name` suggests the modules in `Source/` that aren't listed yet, and `AdditionalDependencies` suggests every module. Hover describes the plugin; `Ctrl+Click` opens its `.uplugin` or the module's `Build.cs`.

Fixed

- **False IntelliSense errors in code that compiles.** clangd didn't see the precompiled header Unreal builds each module with, so anything a file got only from the PCH was an error. For example, `IMPLEMENT_SIMPLE_AUTOMATION_TEST` without `#include "Misc/AutomationTest.h"` gave "a type specifier is required" and "use of undeclared identifier", whether the macro was written on one line or several. `compile_commands.json` now force-includes each module's PCH, as UnrealBuildTool does, and new files inherit it. This needs one build of the project; UStudio updates it after each build and on startup. The first time clangd opens a file it can take a few more seconds, while it parses the PCH.