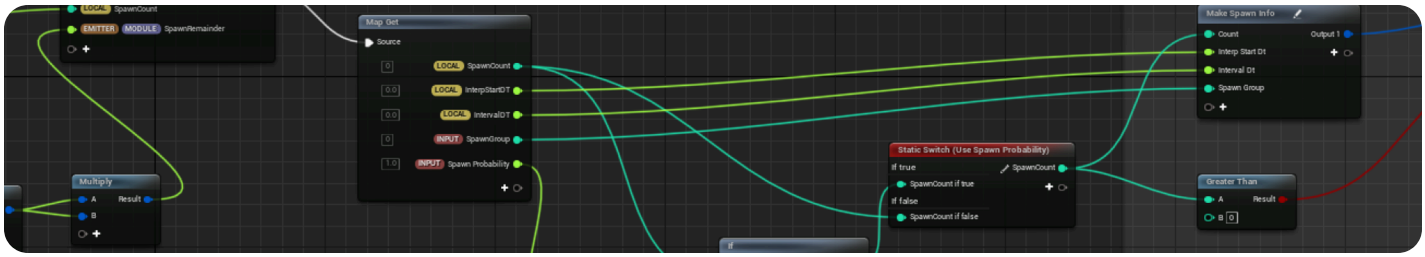
Versioning Modules and Emitters in Niagara

Niagara has a built-in versioning system for people that are creating their own custom modules and emitters.



Overview

Niagara gives you the ability to create your own custom modules using **Niagara Scripts**. When you create your own module, you may want to roll that module out to a team, or use it in many projects. As you iterate on a module to add or improve functionality, you want to make sure that you don't break existing assets that already use those modules.

For custom modules that do not use versioning, the default behavior is that changes are pushed directly to assets that use this module. In contrast, enabling versioning means that users will need to manually upgrade to new versions of that module when they become available.

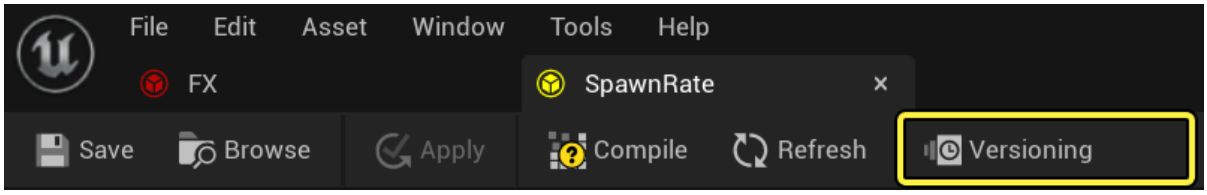
For this reason, you can now create versions of modules directly within Niagara. This is not intended to replace a version control system such as Git or Perforce, but rather is an internal versioning system built directly into Niagara.

You can also save emitters as assets, then apply Niagara versioning to those emitters in the same way you would with Niagara scripts.

Module Versioning

To enable versioning, first open up the module in the Script Editor. Any module can be opened by double-clicking on that module from the **System Overview** in the **Niagara Editor**, or by double-clicking the Niagara Script from the **Content Browser**.

On the toolbar, click the **Versioning** button.



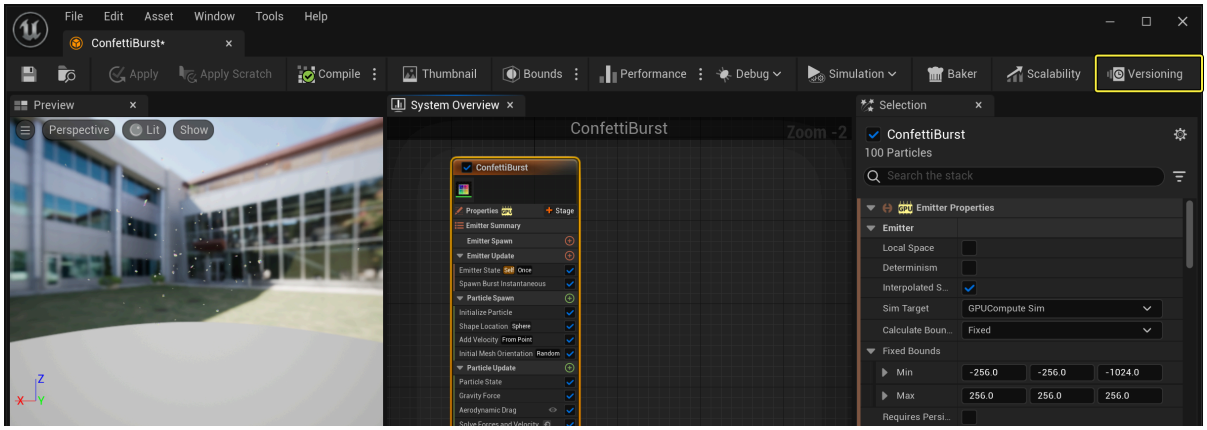
Click image for full size.

Emitter Versioning

Like modules, you can also version an emitter for reuse in multiple projects. You must save your emitter as an asset to use this function. There are two ways to do this:

1. Create a new emitter: From the Content Browser, create a new Emitter asset by right-clicking then selecting **FX > Niagara Emitter**.
2. Convert an existing emitter: From inside a Niagara system, right-click on an emitter and select **Create Asset From This**.

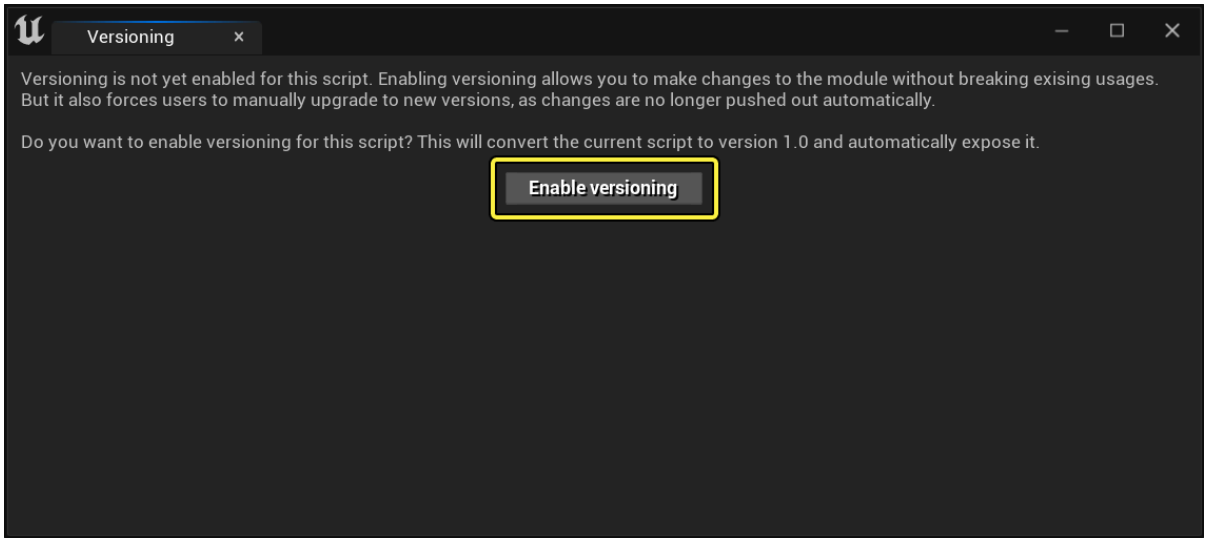
From the emitter asset, you can then locate the **Versioning** button in the top toolbar to enable versioning. This works the same way that it does from a Niagara script asset.



Click image for full size.

How to Enable Versioning

If this is your first time setting up versioning for this module, a popup dialog will appear. This dialog explains that after enabling versioning, any users of a module will need to upgrade to new versions manually when changes are made. Click **Enable Versioning** to accept.

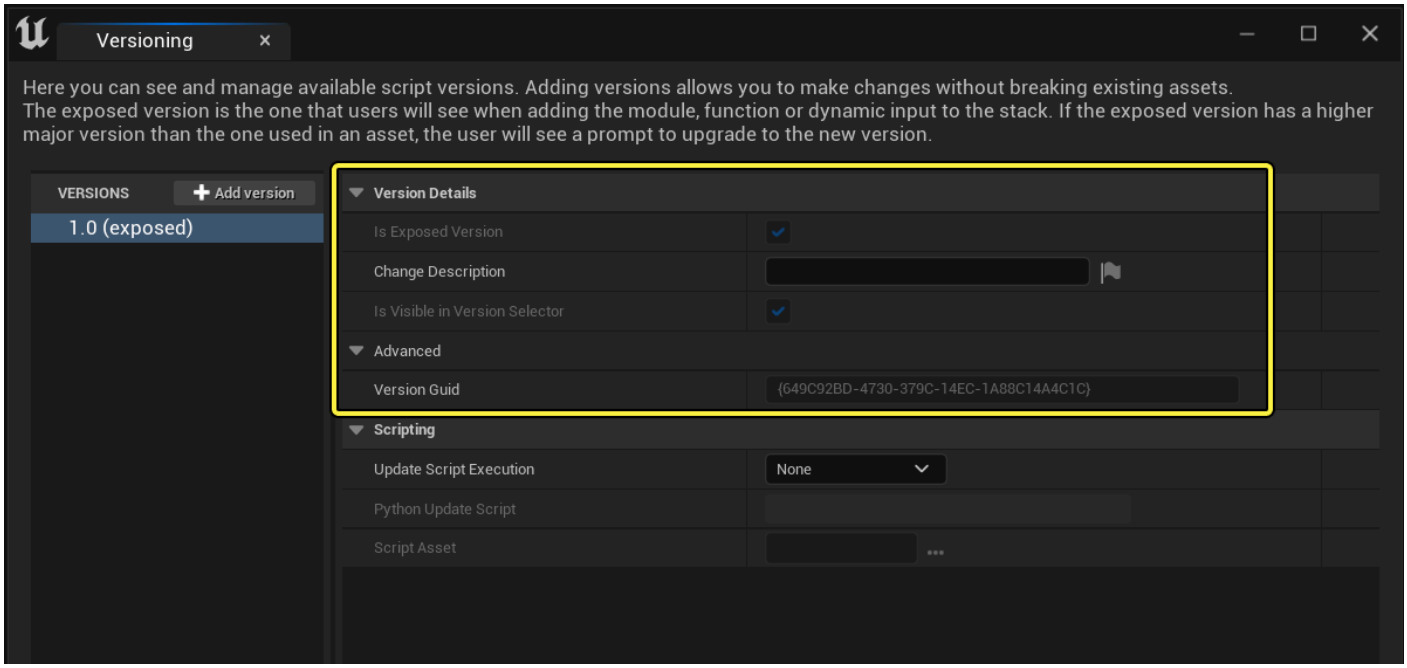


Click image for full size.

Now, you can edit the properties of your versions or create new versions.

Version Details

Each version that you create has some version details for you to set.

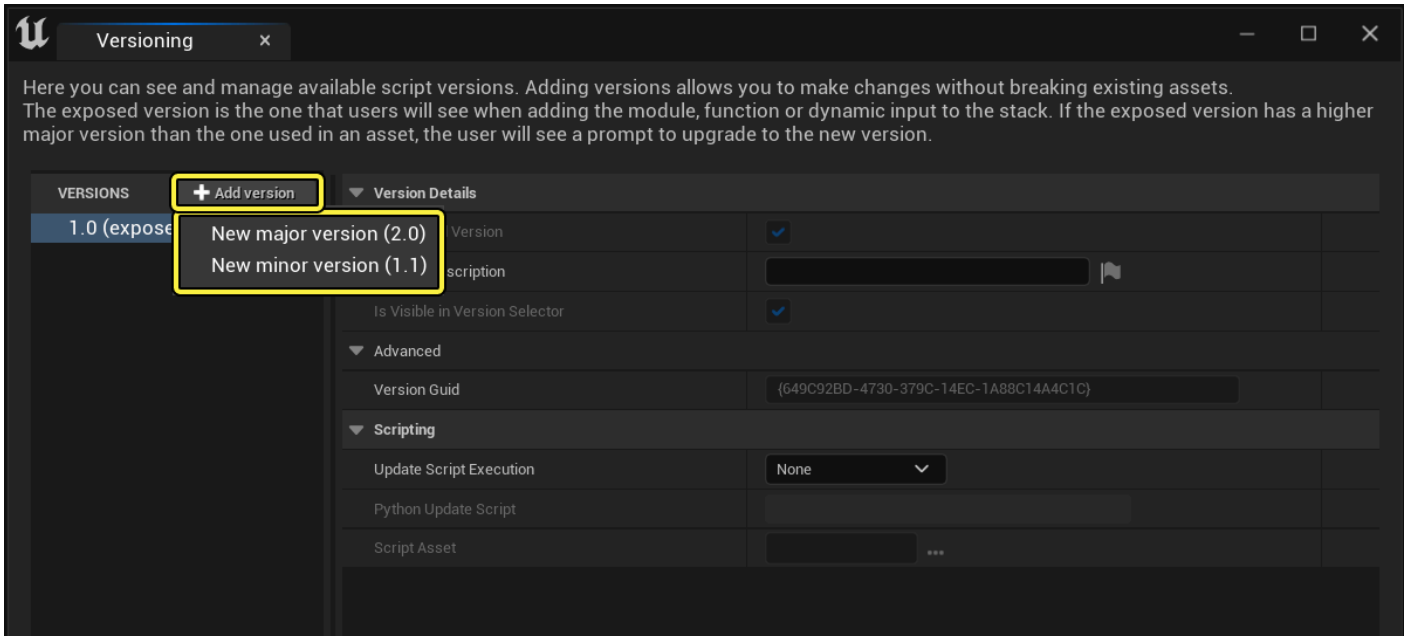


Click image for full size.

Parameter	Description
Is Exposed Version	Enable so this version also becomes the default for any time this module is used. Anyone using this module will be able to see this version in the version selector.
Change Description	Write some text to give clarity to users on what is new in this version.
Is Visible in Version Select	Enable to make this version available to users in the version selector. You can leave this unchecked when you are iterating and testing new versions of a module, but don't want anyone to have access to it yet.

Creating New Versions

To create a new version, from the **Niagara Script** view, click the **Versioning** button to open the panel. Click on **Add version**.



Click image for full size.

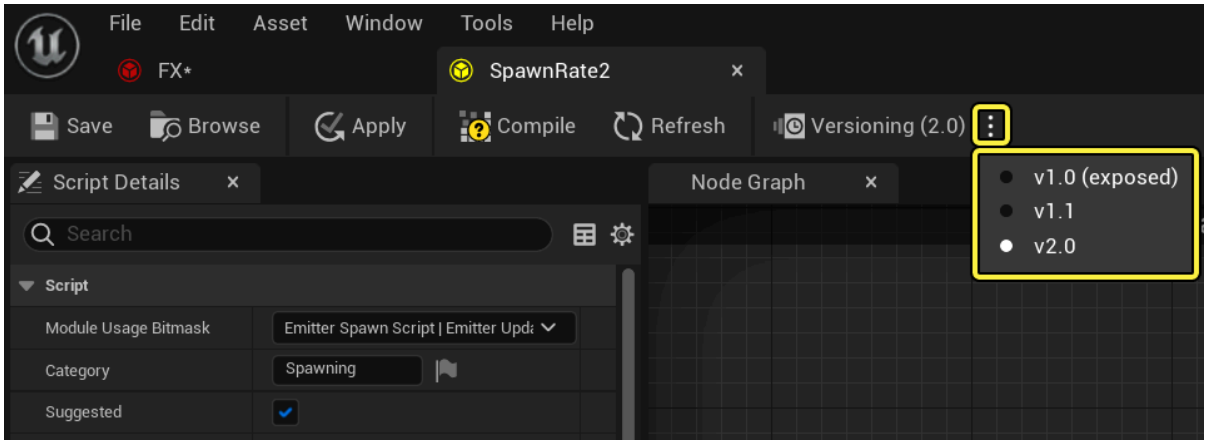
When you create a version, you first need to indicate whether it is a **major version** or a **minor version**. A minor version should be used for small changes that would not be a breaking

change. A major version is for changes where, once you migrate to that version, you cannot migrate back to the old version without breaking the properties that have already been set up.

There is no internal difference between a minor and a major version, this is simply a language distinction to help users identify the risk involved in upgrading.

Select **New major version** or **New minor version** to continue. You can then set the **Version Details**. While you're working on setting up your new version, it's best to leave **Is Exposed Version** unchecked. You can also uncheck **Is Visible in Version Selector** if you don't want anyone to be aware that you're working on a new version. Once you're satisfied with your changes, you can enable these options to propagate them out.

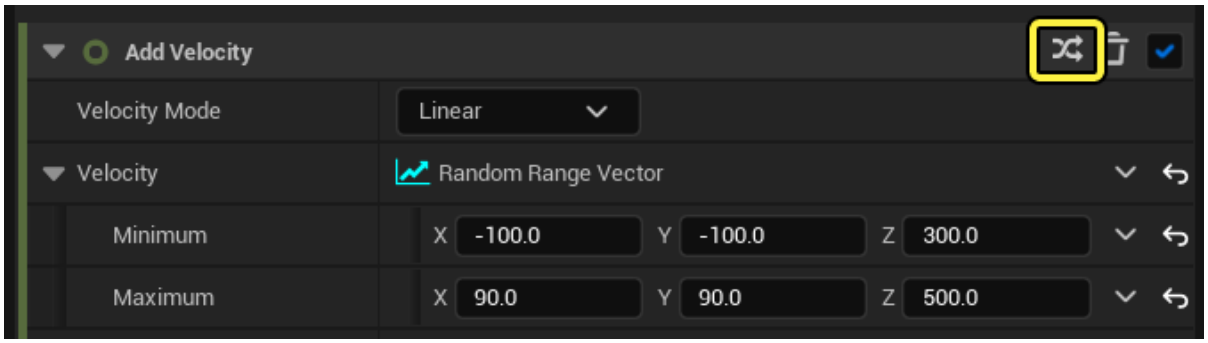
Once you create a new version, you can close the window to return to the Niagara Script Editor. At any time, you can switch the active version that you are editing from the three-dots menu in the toolbar.



Click image for full size.

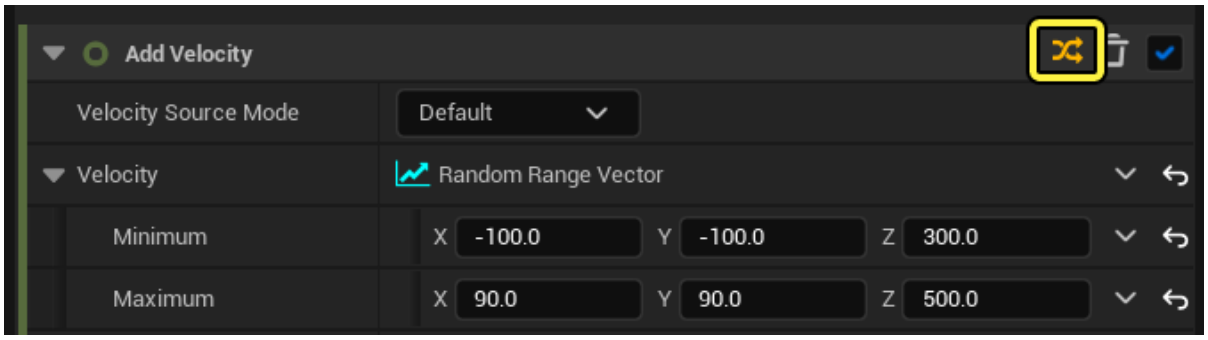
Using Different Versions

In the **Niagara Editor**, when you select a module in the stack that uses versioning, you will see a version selector icon in the **Selection** panel. If the module has versioning enabled, but there is no new version, the version icon shows up as grey.



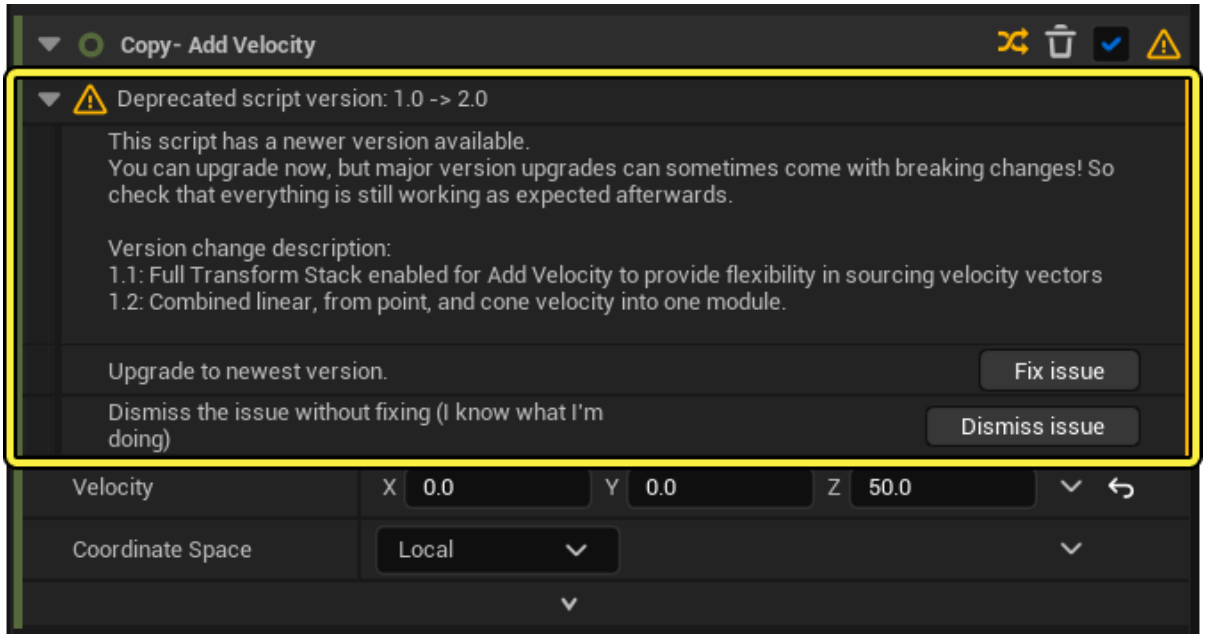
Click image for full size.

As soon as a new minor version is available, the icon turns orange.



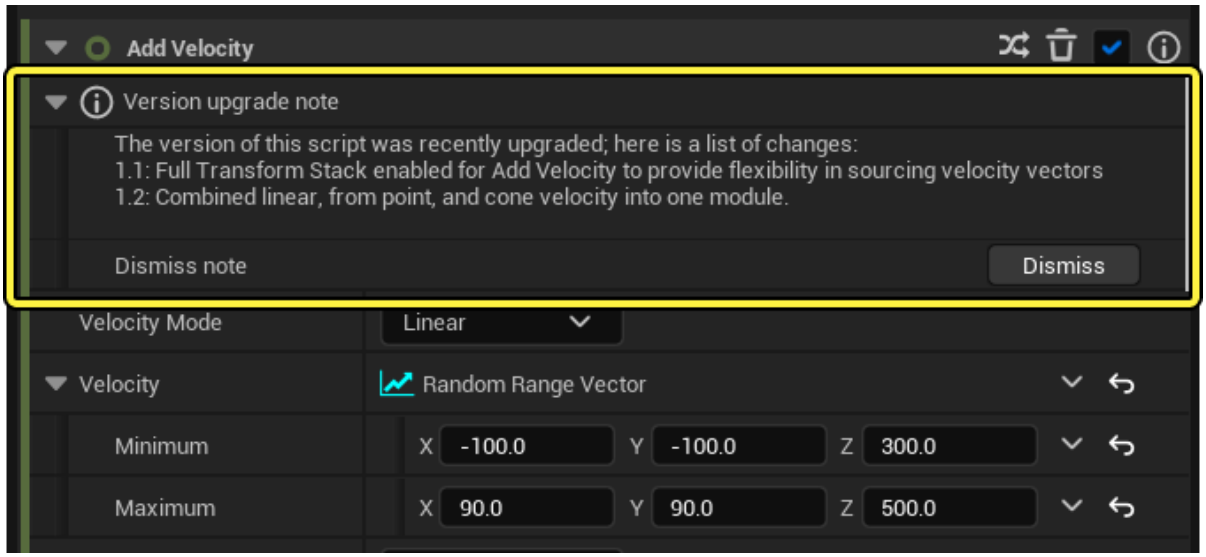
Click image for full size.

If a new major version is available, then there is also a message printed out notifying users of the new version.



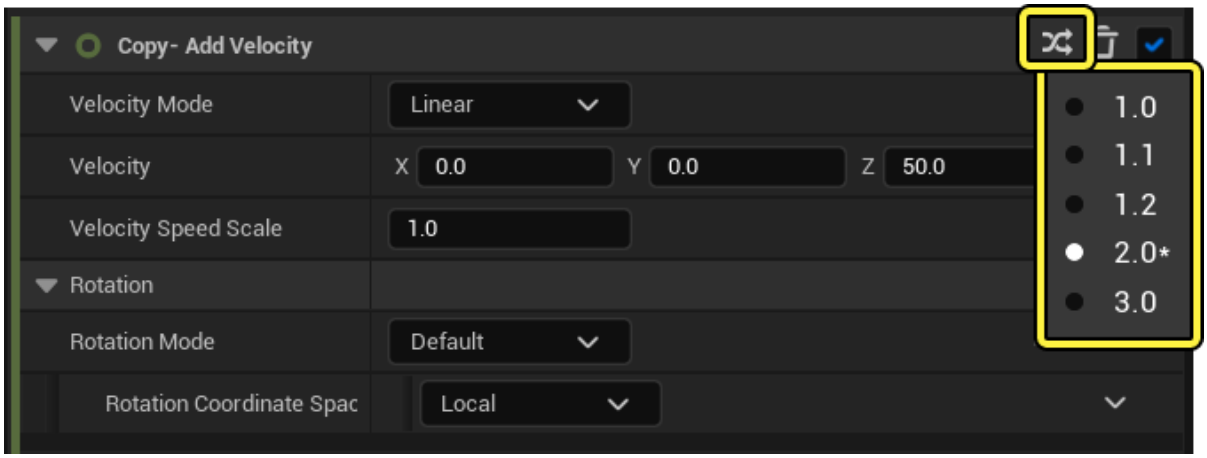
Click image for full size.

Any version description notes will show up in the Selection panel until dismissed.



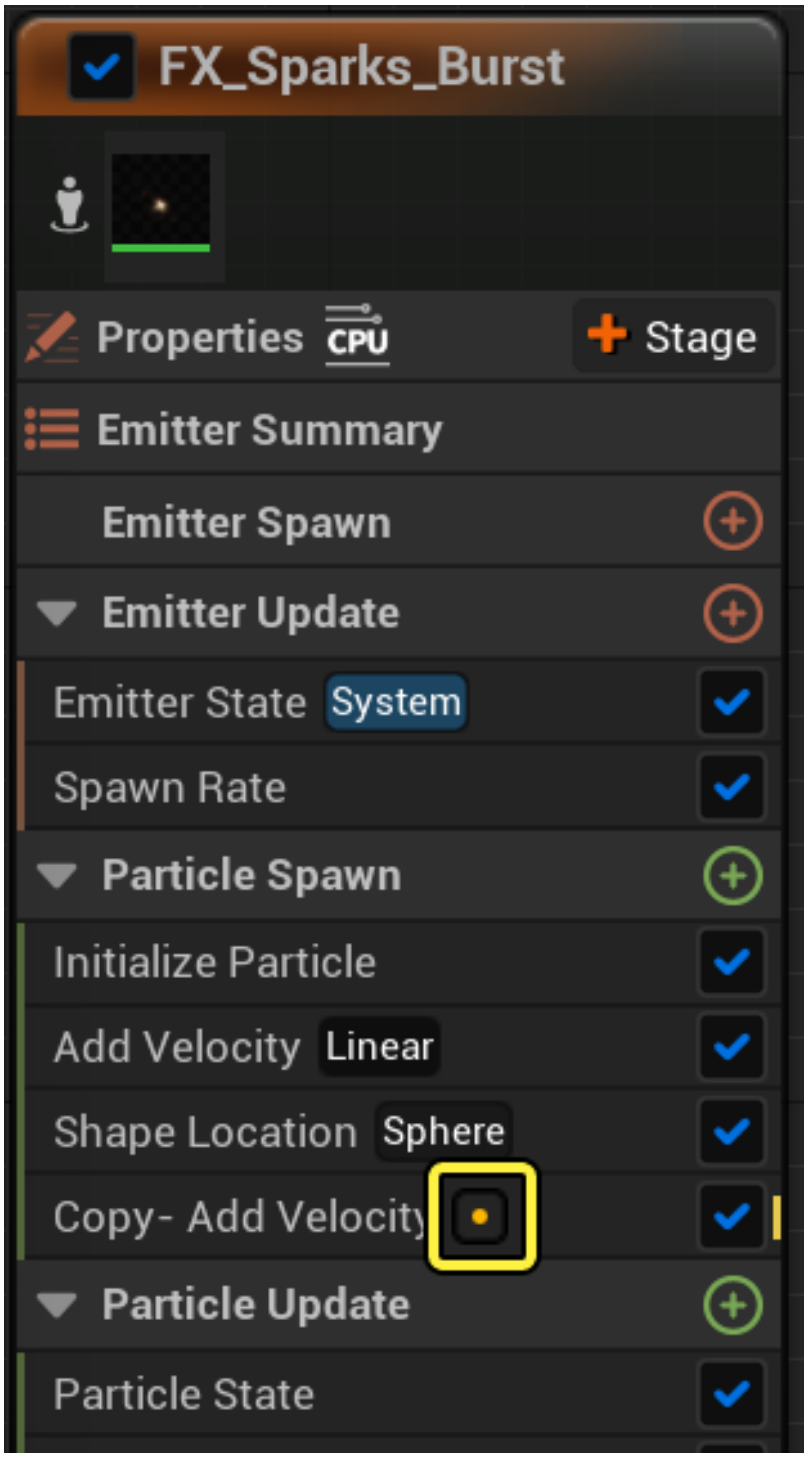
Click image for full size.

At any time you can click on the version button in the Selection Panel to change from one version of the module to another. If you hover over the version number, you will see a tooltip with the description for that version.



Click image for full size.

When a new major version is available, you will also be informed by a warning icon in the Emitter stack. The warning icon shows up on the right side of the module, as well as the group that the module is in.

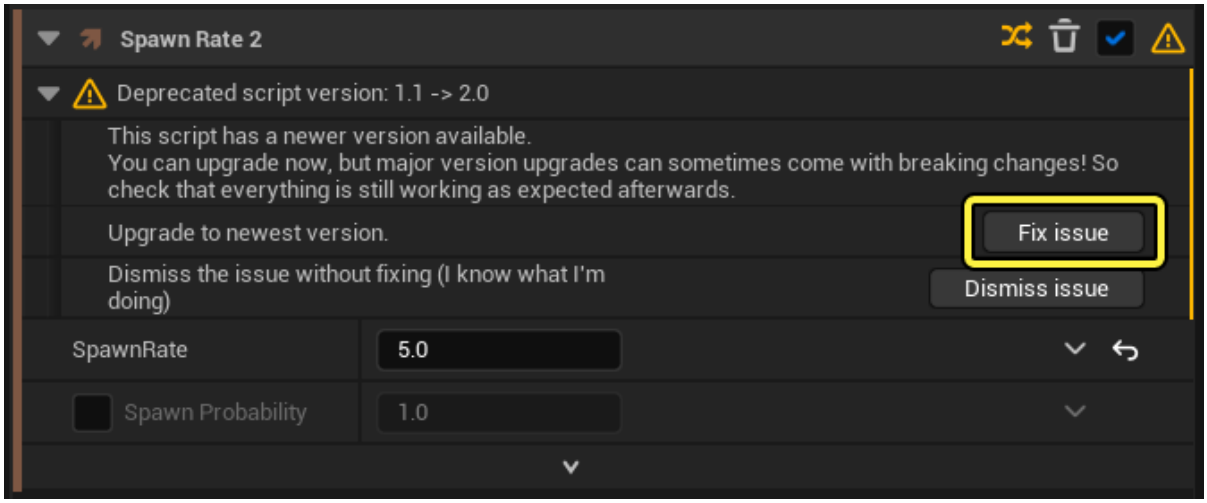


Click image for full size.

> Once you switch to a new version of a module, it is not always possible to go back. It's a good idea to save your project first, then switch to the new version and validate that everything is working properly.

To switch to a new version, in the **System Overview**, select the custom module you want to update. Click the version switcher, then select the version you would like to use. If the new

version is a major version, you can also upgrade to this version directly from the Selection panel by clicking on **Fix Issue**.



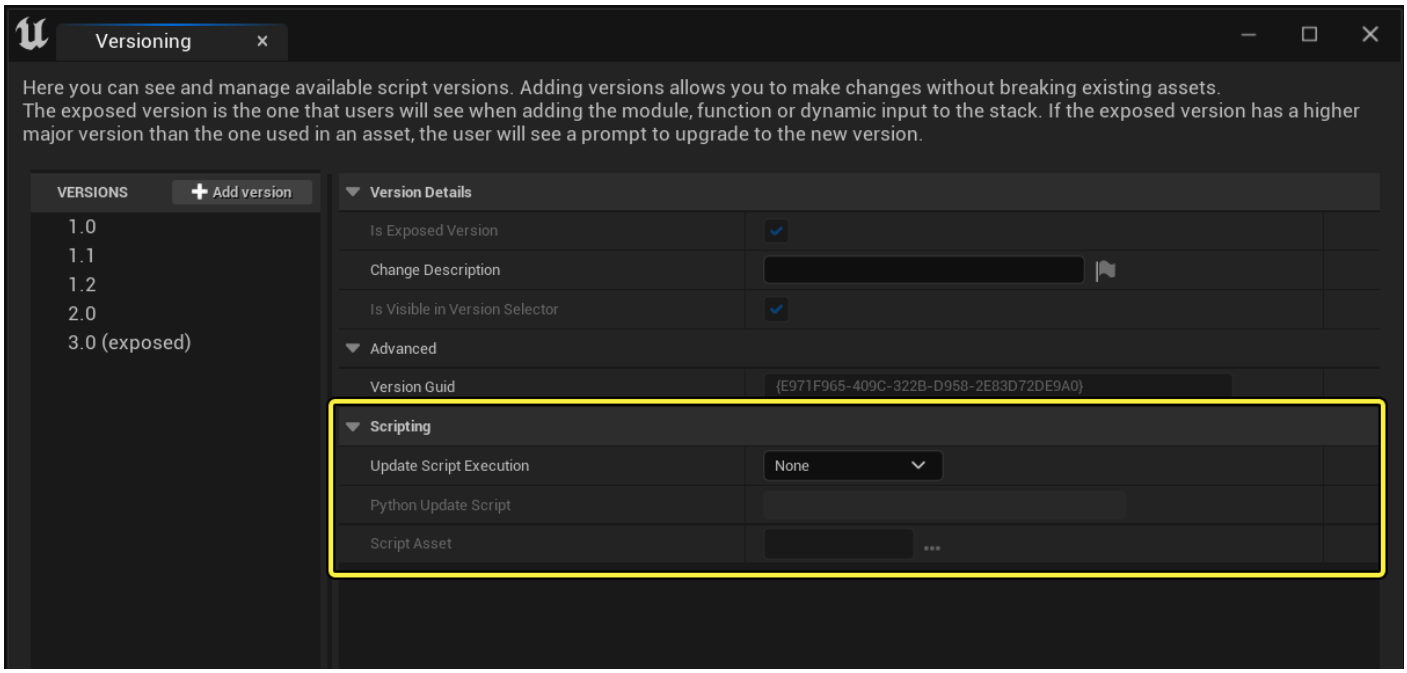
Click image for full size.

Python Integration

> This feature is still experimental and may change in the future.

When you upgrade from one module version to another, the system will do its best to map the properties of the old version to the new version. However, if the desired outcome is not clear you might want to write your own upgrade script to tell the system how to properly upgrade the version. This will ensure that users don't have to redo all their work after upgrading.

To provide an update script, click the **Versioning** button on the Niagara Script toolbar to open the versioning panel. You will see a section of this panel called Scripting. By default, **Upgrade Script Execution** is set to **None**, meaning that you are not providing a script.



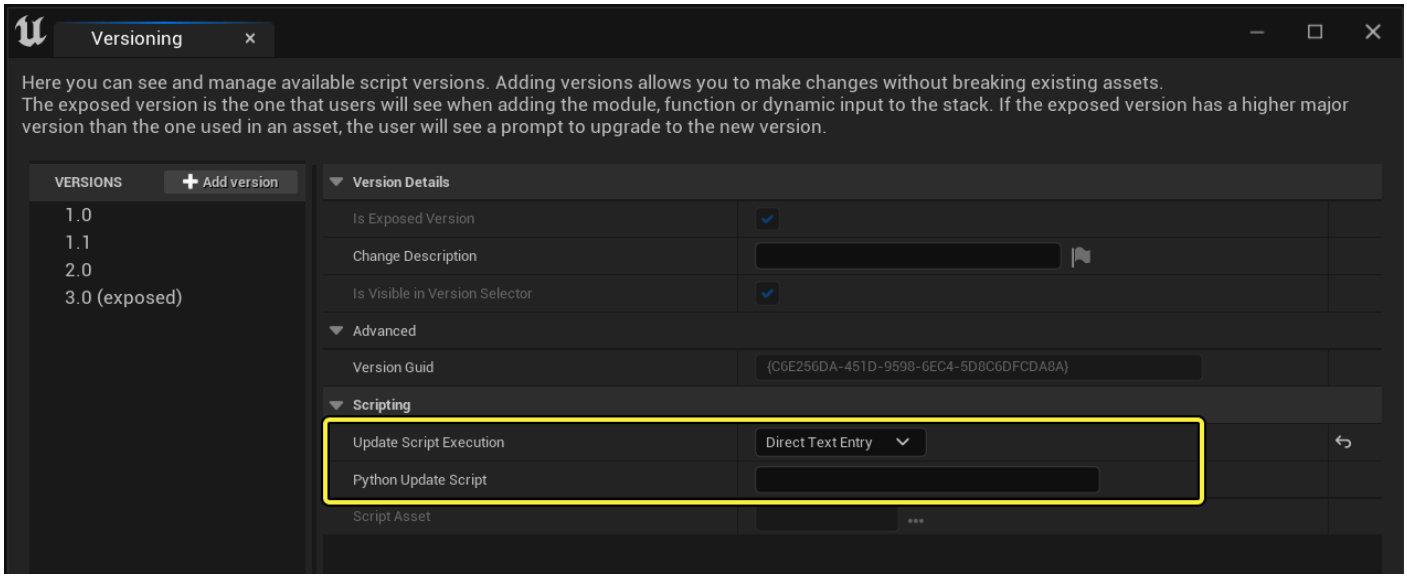
Click image for full size.

There are two ways to enter a script:

1. Copy and paste the text directly into the **Versioning** panel.
2. Link to an external asset.

Direct Text Entry

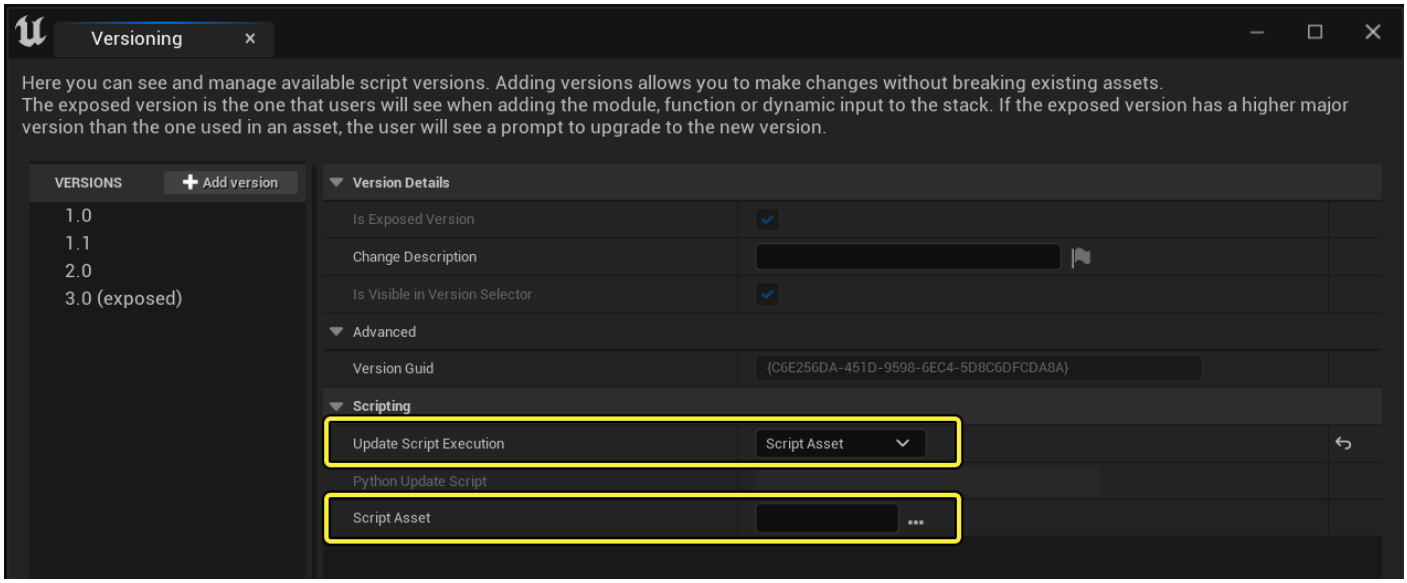
To copy and paste directly, first click **Upgrade Script Execution** and select **Direct Text Entry**. You can now copy and paste your script into the field **Python Update Script**.



Click image for full size.

Adding an External Script

To link to an external script, first click **Upgrade Script Execution** and select **Script Asset**. You can now click the three dots menu to the right of the Script Asset field to browse to your script file.



Click image for full size.

Writing Python Scripts

You can use a python script if it's not clear from one version to another how the pre-existing Niagara script should map to the new one. For example, if the old version takes a bool input and the new version instead uses an enum that has more than two values, the upgrade script can map the existing bool input to two enum values. Here is an example script that does that:

Get bool input, set new input as enum

```
1
2 flying = upgrade_context.get_old_input("Is Flying")
3
4 if not flying.is_local_value():
5
6 print("Is Flying input used a dynamic input that could not be transferred to
  the new Movement Mode input")
7
8 elif flying.as_bool():
9
```

```
upgrade_context.set_enum_input("Movement Mode", "Flying")

else:

upgrade_context.set_enum_input("Movement Mode", "Walking")

```

Copy full snippet

The `upgrade_context` variable is provided to the script and contains the old as well as the new inputs.

Calling `get_old_input(string input_name)` on it returns an input object you can use to get the current stack values. Similarly, you can use `set_XXX_input(string input_name, XXX value)` to provide a value for a new input.

Any `print()` calls you make will show up as warnings in the stack after the script is done.

For more in-depth documentation on what you can do with python in Unreal, check out Scripting the Editor using Python.

Python API Listing

See below the Niagara Versioning object API. Click here for the full Unreal Python API Documentation.

upgrade_context API

```
    get_old_input(string input_name)
```

Copy full snippet

This returns a `UNiagaraPythonScriptModuleInput` (see below). If the input does not exist this will return a blank default object instead of throwing an error.

To set a new input by type:

```

set_float_input(string input_name, float value);
```

```
set_int_input(string input_name, int value);

set_bool_input(string input_name, bool value);

set_vec2_input(string input_name, Vector2D value);

set_vec3_input(string input_name, Vector value);

set_vec4_input(string input_name, Vector4 value);

set_color_input(string input_name, LinearColor value);

set_quat_input(string input_name, Quat value);

set_enum_input(string input_name, string value);

```

Copy full snippet

UNiagaraPythonScriptModuleInput API

```
bool is_set()


```

Copy full snippet

This returns `true` when the user sets a value.

```
bool is_local_value()


```

Copy full snippet

This returns `true` when the input is set to a local value and not a linked attribute or dynamic input.

If `bool is_local_value()` returns `true`, then you can convert the input to a python value as follows:

```
float as_float()

int as_int()

bool as_bool()

Vector2D as_vec2()

Vector as_vec3()

Vector4 as_vec4()

LinearColor as_color()

Quat as_quat()

string as_enum()


```

Copy full snippet