UNREAL ENGINE

Fortnite | Epic Games

# Workflow on Fortnite:

## Collaboration on large teams with UnrealGameSync

# Contents

# Introduction

Epic Games released the original *Fortnite* game in 2017 and followed it shortly thereafter with the *Save the World, Battle Royale*, and *Creative* modes. With a player base surpassing 350 million, *Fortnite* constantly delivers new content, gameplay modes, and features that keep players coming back.

To keep up this pace, we have, out of necessity, evolved an approach to implementing and delivering changes to *Fortnite* effectively and efficiently. Many outside studios have asked us how we manage this feat.

This paper explains the evolution of our development setup, the reasons behind it, and the details of its implementation for *Fortnite* and other titles in the Epic Games catalog. Our solution emphasizes the need for constant iteration with a pragmatic approach to stability, centered around the Unreal Editor and UnrealGameSync tools.

We offer this information to technical directors, build engineers, technical artists, and game designers wanting to distribute the Unreal Editor to their teams for the purpose of streamlining their own development processes on large projects.

Fortnite | Epic Games

Fortnite | Epic Games

## The challenges of large-scale game development

During the development of a game, programmers, designers, and artists work hard to make it engaging, visually appealing, and bug-free. And while the game is being iterated on, developers are faced with the challenge of keeping the build and tools stable for content creators while also continuing to add new gameplay features to the latest version of code.

Developers at Epic Games deliver hundreds of changes a day to *Fortnite*'s mainline, and our infrastructure and workflow are designed to sustain this volume of changes with as little friction as possible. However, it wasn't always this way.

# Legacy workflow

In the early days of development on *Fortnite*, content creators relied on a process developed during the Unreal Engine 3 timeframe called the *editor promotion pipeline*.

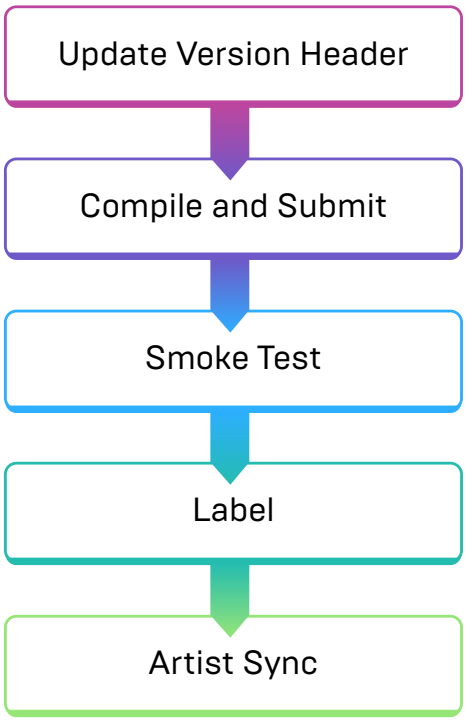


*Figure 1: Traditional editor promotion pipeline*

The latest *Fortnite* source code would be synced and compiled on a build machine, then the resulting binaries would be submitted to Perforce. Our QA team would sync down and run those binaries, and—if they passed a test plan—apply a Perforce label to them to mark them as good. We called this process *promotion*. Content creators could then sync down the latest promoted binaries via the Perforce label.

## The build-to-fix cycle

If errors were found during compilation or bugs were identitifed during testing, QA would file bugs for developers to triage and fix.

As the size and complexity of the project grew, so did the surface area that required testing and the likelihood of bugs being identified. QA was not well-placed to know how severely bugs impaired active workflows on a particular project, so would typically err on the side of caution and enter anything they found as a high-priority issue in our bug-tracking software.

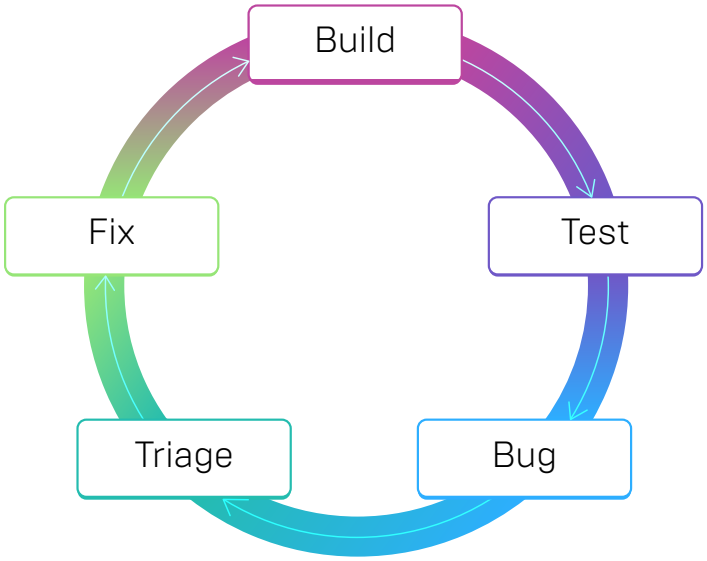


*Figure 2: Build-to-fix cycle*

An iteration in the build-to-fix cycle would take half a day at best—sometimes even a couple of days if a developer wasn't able to reproduce an issue or was away from their desk. Only then could the entire process be repeated, and by the second iteration, there was a strong chance that someone had submitted new changes that had broken something else.

As the project grew, turnaround slowed to a point where it could be a week between promotions, and this challenged the collaboration between engineers and content creators.

Fast turnaround times are crucial for many Unreal Engine projects because engineers and designers often collaborate on gameplay functionality through Unreal Engine's Blueprint visual scripting system. Iteration is paramount to game development, and bottlenecks in the pipeline of delivering new code to designers result in fewer opportunities to review and improve the quality of the final product.


Fortnite | Epic Games

## Content-centric and code-centric

There were other problems with this workflow. With the build system constantly submitting new binaries to Perforce, engineers would frequently get mismatches between binaries they built themselves and those they were syncing to. If they were lucky, this would trigger an obscure dialog box from the OS referencing mangled symbol names that couldn't be found. If they were unlucky, the editor would load but crash with an obscure callstack caused by memory corruption from class layouts not matching between modules.

Engineers would attempt to avoid this by manually excluding precompiled binaries from their Perforce clientspec, but the process was unreliable and hard to keep up to date with the introduction of new files. In addition, having to filter files by a large number of wildcards can be taxing on the Perforce server, and can increase sync times.

On the occasions that engineers needed to change content, they would have to check in their code changes, wait for the editor to be promoted, and only then submit their content changes. If they tried to check in some changes made with their local build, the changes would be *unversioned*, and any developer using the promoted editor would not be able to load them.

Understanding the need for versioning is important for deploying the engine to a team.

## Versioning

Unreal Engine's property serialization model is flexible and convenient for continuous development, where you can iterate on code that exposes properties to the editor without having to resave any existing assets.

For example, if you add a property to a class, and a previously created asset is saved without that property, the engine will initialize the property to its default value the next time the asset is loaded. If the default value for a property hasn't been overridden, anything loaded from disk will pick up any changes to default values automatically. And if you remove a property, any asset which had a value for it would just ignore that property on load.

This lack of explicit versioning can create problems. If a new property is added, and an asset saved out with it, an older version of the editor will not know how to serialize it and will discard it—treating it as if it had been removed.

We solve this problem by compiling the build changelist into the editor executable, saving it into any assets it creates, then preventing the editor from loading anything created by an editor newer than itself. Rather than micro-managing versioning for each code change, we version the whole editor and everything it saves out.

Battle Breakers | Epic Games

# Case studies

While struggling with a cumbersome workflow on *Fortnite*, we had an independent project start up in parallel called *Battle Breakers*. *Battle Breakers* had a particularly fast and effective pipeline for updating the game, and we considered how this process might scale up to a larger project like *Fortnite*. We also took a look at how another title, *Paragon*, evolved that solution for its workflow.

## Battle Breakers

*Battle Breakers* had a team of 10 or so people—a mix of artists, designers, and engineers all working in the same room. They were passionate about their project and driven to make it succeed, and did not want to be held up by overly cautious processes.

Rather than adopting the standard editor promotion pipeline, each developer—artists and designers included—got a copy of Visual Studio Express and learned how to sync from Perforce, generate project files, load the solution in the integrated development environment, and build their own editor.

This workflow was highly effective.

If there was a compile error, or if the editor crashed, the content developer could turn to an engineer a few feet away and ask for help. Visual Studio was already installed on their machines and they had access to the source code with full symbols, so the engineer could efficiently debug the editor and help them get back to work quickly.

An unexpected side effect was the broad increase in communication between artists, designers, and engineers. People got more insight into each other's workflows and shared their ideas on how to improve the editor and tools, as well as the game itself.


Battle Breakers | Epic Games

## Scaling to Fortnite

In considering how the *Battle Breakers* approach would scale up to *Fortnite*, there were a few obvious concerns.

### Stability

The first was stability. If no one was gating the quality of the editor, we'd expect bugs to be discovered by those making user-facing content such as artists and game designers. But what would be the impact?

Artists typically rely on core editor workflows that gameplay engineers don't change very often—the processes for importing meshes, textures, animations, and so on. On a game project, the code changes would not typically affect these processes. Engine changes are typically developed in separate streams, which are merged into the game stream only after thorough QA testing.

For game designers, there would definitely be a tradeoff—a sacrifice of stability in favor of increased productivity. However, there was a strong case to be made for designers being the best people to exercise the most relevant code paths to the game. While QA might be operating from a test plan that is six months old, designers are usually more interested in seeing what is in the game right now and using the latest code from the gameplay programmers immediately, right at the cutting edge.

### Data integrity

There was a concern that someone could check in a change that would corrupt assets or result in data loss. However, anecdotal evidence indicated that this kind of extreme content corruption was rare, and difficult for QA to catch through superficial testing.

### Compile times

It would be undesirable to require content creators to lose time waiting for the editor to compile, but there was a hope that incremental compiles could be reasonably timeboxed. As long as we could ensure the build was more likely to succeed than fail, it could be scheduled to happen during downtime—in the evening or first thing in the morning. Additionally, Epic employed Xoreax's IncrediBuild in order to significantly speed up compile times and minimize impact to the team.

### Interface complexity

Professional tools like Visual Studio Express are daunting for novice users—the interface is unintuitive for their workflow and there is a bewildering array of options that can put them in a bad state. It seemed sensible that we would create a simplified interface for those users who need it.

### Flagging problems

If we were to remove some of the processes guarding downstream developers against bugs, it was paramount to make it easy and fast to flag issues as they arise. For *Fortnite*, the team is spread out globally across numerous time zones, not collected in the same room. Any solution we adopted for *Fortnite* would have to include a system of notifications that worked for this type of team.

## Toward a solution

Rolling out this kind of workflow to a team the size of *Fortnite*'s would require a dramatic culture shift and would have to be an all-or-nothing transition. But the existing process in place at the time was slow and cumbersome, so there was not much to lose.

Over a weekend, our IT department installed Visual Studio Express on *Fortnite* content creators' development PCs, along with a custom tool developed in-house called UnrealGameSync (UGS). UGS had recently been developed at Epic specifically for this new type of workflow.

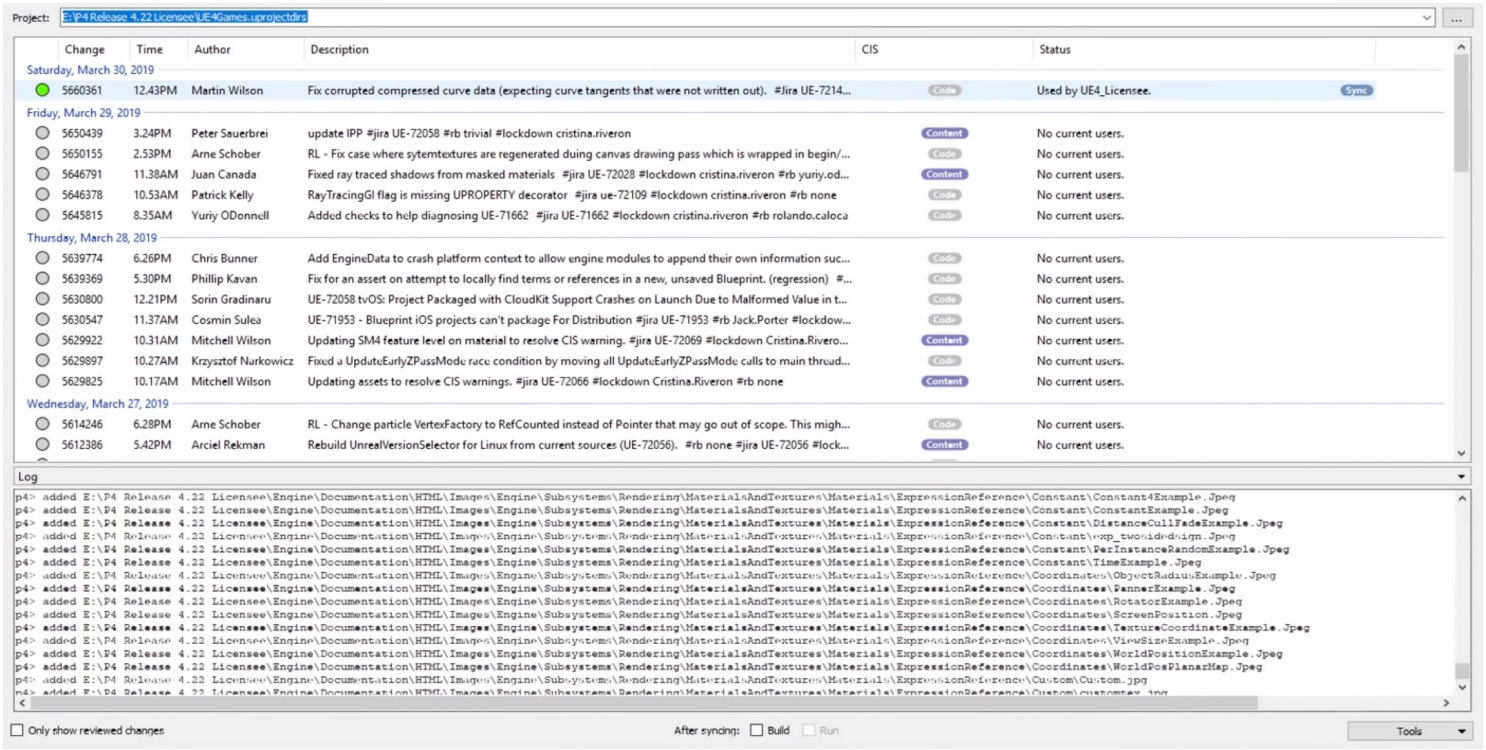


*Figure 3: UnrealGameSync provides a graphical front end for syncing UE projects from Perforce, optionally building those projects with the Microsoft Visual Studio compiler. This image is from an early version of UGS.*

Conceptually, UGS promotes code and content integration in a collaborative environment. It shows a live stream of changes submitted to source control and provides a streamlined interface for syncing and building changes, seeing what other users are doing, and flagging problems that are found.

From within UGS, the content creator would select a project from their local hard drive. UGS would automatically detect the Perforce settings for the project, the workspace in use, and so on, then fetch a list of the changes being checked in that affected that project.

If the artist double-clicked a change, it would sync down and compile the required files locally, then launch the resulting editor.

### Collaboration

For a successful compile, the changelist within a project would be flagged with a green icon visible to anyone else working on the project. For a fail, it would show a red icon.

If the editor crashed at startup or other issues occurred, the artist could return to UGS and mark the change as bad, then comment on what happened. This would give the engineering team an idea of where to start looking for the problem.

If an engineer was investigating an issue that someone had flagged, they could mark the build as “under investigation”. This would flag all subsequent changes as bad until they marked it as good again.

### Scheduling builds

One of the first feature requests implemented was a user-settable schedule to sync a branch and trigger a build after hours, ensuring that the latest stable editor would be waiting when people arrived at the office in the morning.

The version synced for a schedule would typically be the latest version of the code tagged with a green icon to indicate a successful build.

### Catering to other disciplines

Engineers were also encouraged to use UnrealGameSync to update the versioning metadata used to build the editor. This allowed them to modify content using their locally compiled binaries, instead of submitting their code and waiting for a promoted build and then modifying the content.

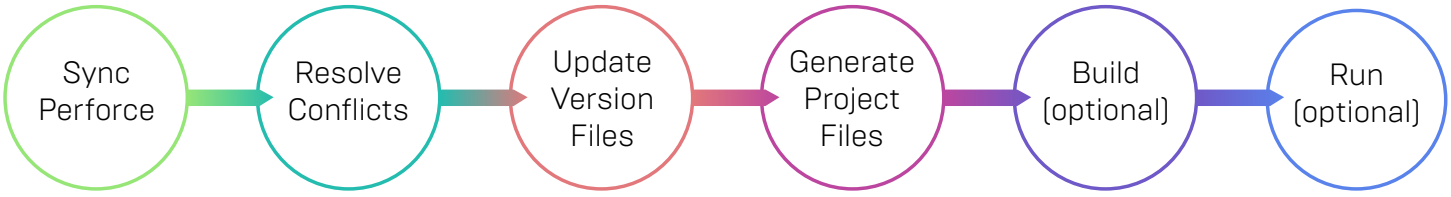


*Figure 4: The UnrealGameSync pipeline*

Fortnite | Epic Games

## Centralizing feedback

UGS very quickly became a workflow staple, and it really helped to recover that rapid iteration cycle that was vital to fast production.

That was four years ago. Today, the *Fortnite* team is still using UnrealGameSync.

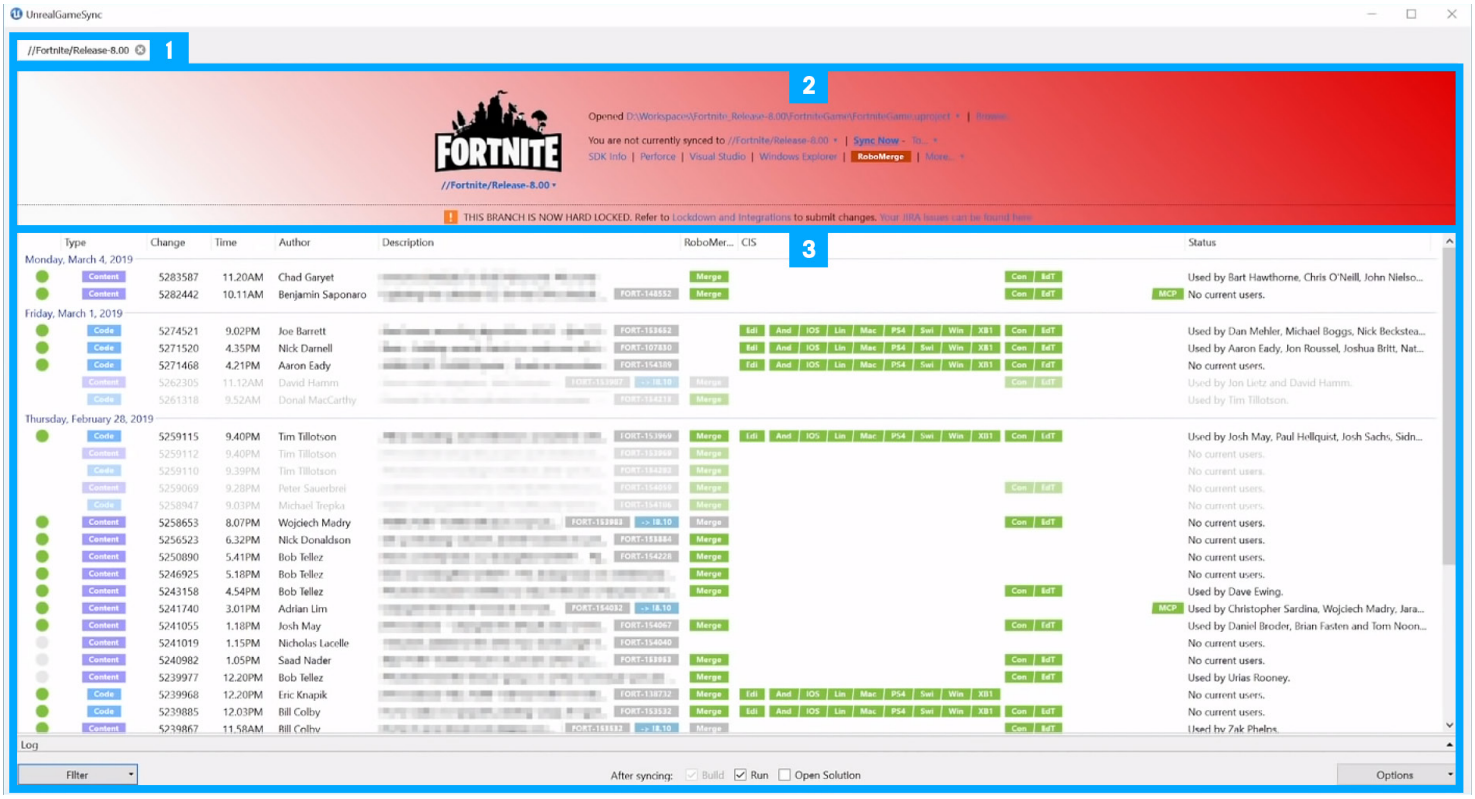


*Figure 5: UGS customized for Fortnite team (key in the table that follows)*

As we continued to iterate with the *Fortnite* team, several other features were added. The UGS areas in the table below correspond to the numbers in Figure 5.

| UGS area | Feature notes |
|---|---|
| **1** | Multiple tabs make it possible to view different projects or streams at the same time. |
| **2** | A status panel at the top shows the selected project and provides access to other tools. The panel also displays which SDK versions are required for the current branch.<br><br>The user can switch between streams easily. Perforce fast-stream switching has had a bad reputation because it is easy to do incorrectly, but exposing it through a tool like this made it possible to do reliably. |
| **3** | Badges on the right surface results from the build system, such as whether a verification build is in progress, or has passed or failed. Clicking on a badge takes the user directly to the build log.<br><br>Because UnrealGameSync is constantly polling in the background for changes in the branch, it knows when something new has been checked in. If a badge turns from green to red, UGS can provide a popup notification to the developer that submitted the changelist that broke the build. |

*Table 1: UGS areas and features added for Fortnite*

For more details about these features, please see the UGS Reference.

For *Fortnite*, we run a continuous build cycle on our build farm to populate UnrealGameSync with badges.

- Every 10 minutes, we compile the editor incrementally. It has unity builds disabled to reduce the surface area of each change, which also catches any missing **`#include`** directives for submitted code.

- Once an editor has been compiled, a commandlet loads up any content that has been changed since the last time it was run. We found that a common source of errors was artists forgetting to submit a piece of content, and just trying to load it up was a quick way of generating errors and warnings describing such problems.

- After the content check, we run a quick automated test that spawns the editor twice—once with the **`-game`** argument, and once with the **`-server`** argument. This test checks that a client and server can talk to each other, that the login flow and matchmaking process works correctly, and that you can make it into a match. It also checks all of the four basic actions that you can do in *Fortnite*: move, shoot, build, and harvest.

- A second machine incrementally compiles the game for all target platforms.

These two machines running these tests have proven to be of massive value, and strike a good compromise between quick feedback and broad coverage.


Paragon | Epic Games

## Paragon

*Paragon* was the next project to try out UnrealGameSync. *Paragon* was a multiplayer online battle arena (MOBA) with a quick ramp-up to a large art team and an aggressive content pipeline. The designers wrote most of the gameplay logic with pre-existing Blueprint functionality, so iteration between design and engineering was less critical than on *Fortnite*.

With such a large art team, the overhead of having artists compiling their own binaries was deemed detrimental to throughput. However, we wanted to retain several other benefits of UGS: allowing engineers to save correctly versioned content, displaying feedback for build health issues, and keeping pre-built binaries out of the workspace.

### Zipped binaries

The solution was a mode where editor binaries from a build server would automatically be archived to a ZIP file and submitted to a location outside the branch. Once the ZIP files were submitted, content creators could choose to have UGS download and extract them without having to manage a second workspace or build them locally.

These archives are published for a specific, matching changelist number, effectively matching the build-from-source workflow. Any changes that don't have matching precompiled binaries are shown as grayed out in the UI. Syncing content-only changes after a change with matching binaries is permitted.

Users who do not want precompiled binaries can sync and compile if they prefer.

### Client-side filtering

We worked with a number of off-site contractors and outsourcing studios on *Paragon*, which presented its own challenges. *Paragon* had a large number of high-resolution character assets, and a developer would typically need only a few at a time to do their work. For anyone working over a VPN connection, syncing the full lineup of characters was a waste of bandwidth and highly time-consuming.


Paragon | Epic Games

Our first attempt to solve this was to create virtual streams in Perforce, which would narrow the user's workspace to a subset of directories rather than the entire branch. This worked on a small scale, but the number of permutations escalated quickly. Constantly shipping updates meant a large number of branches too, and content creators would want their personal customizations on every branch.

Instead of managing these permutations on the Perforce server, UnrealGameSync supports client-side filtering, which queries the server for the contents of the stream and selectively syncs (and removes) files, leaving only the ones that the user needs.

The UI presents a customizable, à la carte set of components to include or exclude which users can supplement with their own filters. For example, the user could exclude any platforms that were not relevant at that moment, or leave out cinematic content or specific localized assets.

These selectable components have the advantage over old-style Perforce clientspecs of centralizing the configuration for each one. Since the configuration file defining the components is submitted to source control itself, the filters can be automatically updated for everyone on the project without requiring any explicit configuration changes by individual users.

## Clean Workspace tool

The UE directory layout has intermediate files for plugins and game projects scattered all over the source tree. The Clean Workspace tool within UGS can be used for cleaning out any intermediate files, as well as finding any missing files that should be in the workspace.

All the differences between the local workspace and server are shown in a tree, and any known intermediates that are safe to delete are selected by default. This means that the tool will not accidentally delete any work that you have created and just forgotten to check in.

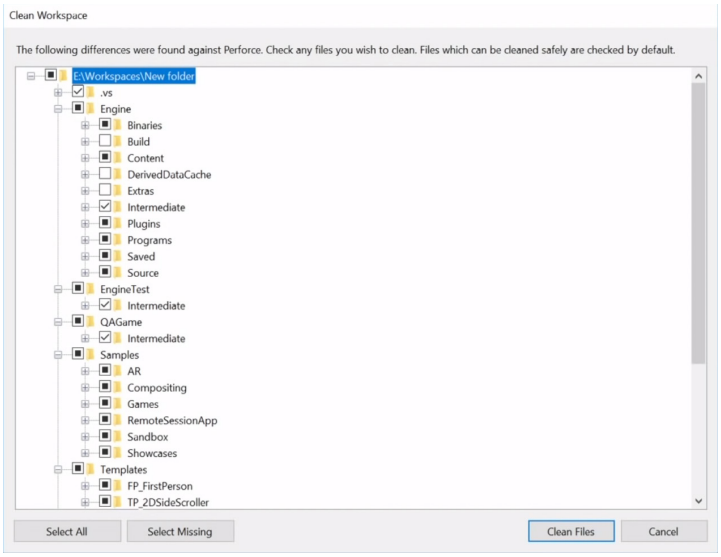


*Figure 6: Clean Workspace tool*

Unlike the Perforce Clean command, the Clean Workspace tool checks only for the existence of files at expected locations, then checks the files' lengths and read-only statuses. Depending on your PC, the entire process can take about 10 to 20 seconds—not even enough time to grab a fresh cup of coffee.

## Unreal Engine

The UGS workflow's advantages aren't limited only to our game projects. Unreal Engine's QA department also uses UGS when regressing bugs.

### Bisect mode

Oftentimes, QA knows that a regression in functionality happened between a known good changelist and a known broken changelist, and they need to narrow it to the offending changelist as efficiently as possible. From the UGS panel, the tester can select a range of changes, then run what is called Bisect mode. This mode filters the view to include only the changes selected. From this view, the user can mark the changes as good or bad.

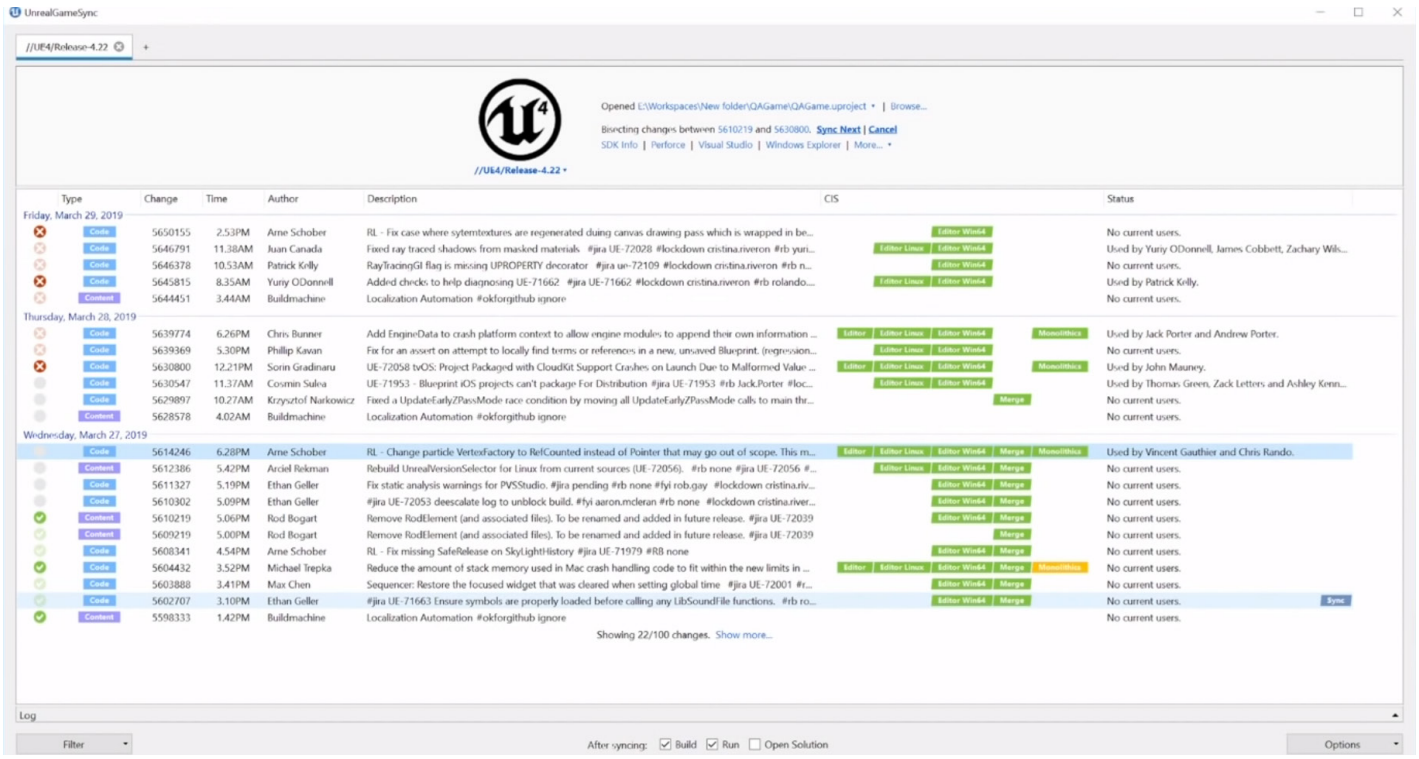


*Figure 7: UGS Bisect mode*

If the tester syncs to a change, tests, and doesn't find the bug immediately, the tester can select the Sync Next command, and UGS will sync to the middle between the last good and the last bad change, helping the tester perform a binary search for the change that introduced the bug.

### Config files

Finally, we added a lot of little customizations that were useful for our own projects.

There is a message-of-the-day feature underneath the status panel which usually gives information about the dates that we are going to a branch. The message-of-the-day also provides a link to your list of bugs in our bug database.

We have a Perforce trigger that requires that a bug number be included in the description of any submitted changes. We set up UGS to parse that information and create a link back to the corresponding bug in the bug-tracking software, which a user can open by clicking a badge next to the change in UGS.

All these features are driven through config files, based on the config files that are checked in to a branch. This makes it possible to tweak UGS for the unique needs of different projects.

# Using UnrealGameSync for your projects

*Fortnite*'s success depends on our team members' ability to effectively collaborate and communicate with one another. The workflow challenges we encountered on *Battle Breakers* and *Paragon* led us to develop UGS, which helped us convert the collaborative benefits of working together in a room into a global, distributed development environment.

We distribute the latest version of UGS with Unreal Engine and you can use it with any version of UE. Its source code is available in Perforce and on GitHub, and its use is covered by our Unreal Engine EULA.

You can also find more information on setting up UGS on the UGS Reference page in the Unreal Engine documentation.

## Authors

Ben Marsh

Robert Gervais

## Editors

Michele Bousquet

Su Falcon

## Layout and media design

Jung Kwak