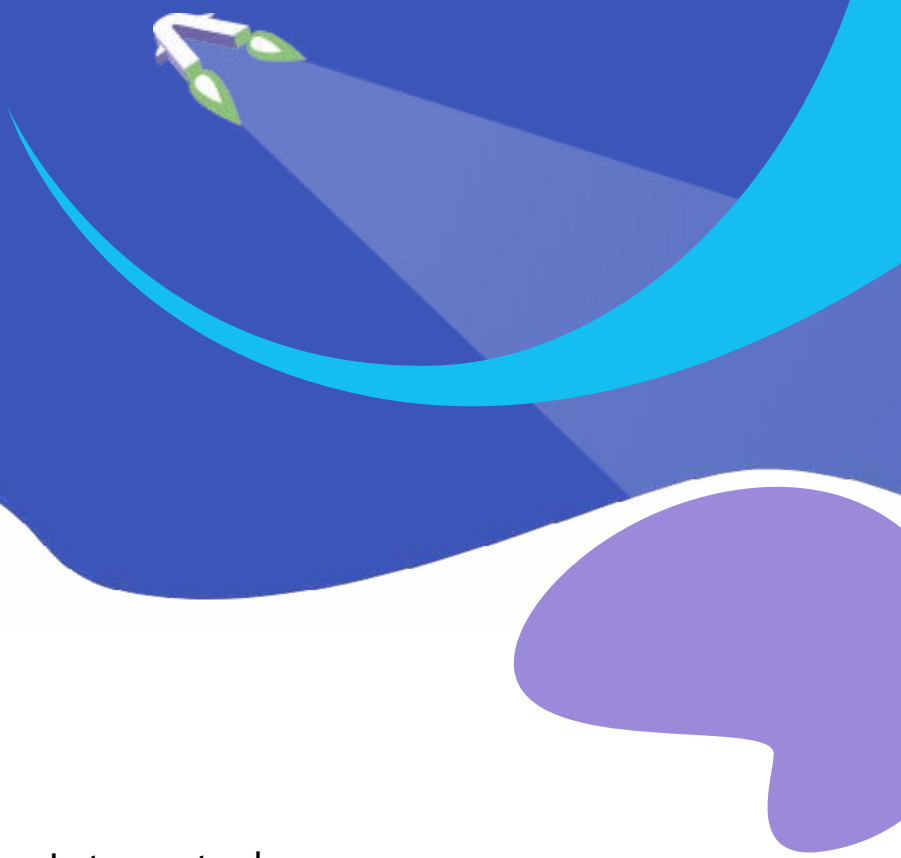
BEAMABLE

Architecture for Scalable, Reliable and Efficient Game Operations

Beamable Technology White Paper



One Line of Code launches Integrated,

Full-Stack LiveOps for Unity

Introduction

In this White Paper, you'll learn how this one line of code gives you reliable infrastructure to build a live services game:

```
var beamable = BeamContext.Default;
```

This code is your the gateway to a full-stack, integrated platform that lets you create custom server logic with C# microservices – along with a suite of LiveOps tools and services to build and operate games that players love.

All games are now live games: your players have become more demanding, more discriminating, and more distracted than ever before. They are hungry for engagement, immersion and community. That means releasing live games with social features, deep economies and regular content updates–and you need to build games faster than ever.

Gone are the days when you could launch a rough beta and expect to scale–indeed, the risk today is one of runaway success: what happens when your game launches and millions of people want to start playing it? Will that be an opportunity to scale, or a risk of disaster?

At the same time, the market has become more competitive than ever: in 2021 alone, over $85 billion flowed into game deals. After only one month in 2022, that number was already exceeded! That means that while there's more capital than ever to build games, it is spread across a dwindling supply of experts in backend programming, server scaling, DevOps and live operations.

Competitive advantage in game development centers on delivering great game experiences–storytelling, art, engagement, interaction–while ensuring that you can reliably scale, support and manage your game as it grows. Quickly building and iterating these features and content is critical.

Our team at Beamable's team has a background building games that needed to scale to over 20 million players. We lived through all the complexity involved in supporting games with deep economies and rich social interaction and then scaling up to support millions of active players.

In the following sections, you'll learn:

- **How a full-stack, integrated approach to LiveOps helps you build and iterate faster**
- **What we learned from the lifecycle of several live game projects**
- **The core issues that contribute to complexity, mistakes and lost opportunities**
- **Beamable's architecture, built around a highly-scalable Serverless Game Backend**

Contents

What We Learned from the Lifecycle of Live Games

Let's imagine a brand new game project that's going to use Unity, and consider what happens across the course of development, and ultimately live operations. This could be a new studio, or an existing publisher spinning up a new title–and join this game on its journey from idea to LiveOps.

- **Competition between feature ideation vs. implementing streamlined workflows**
- **Problems with shipping and deploying sets of data, content and code**
- **The brittleness of fragmented architectures with multiple tech stacks**
- **The large workload associated with patching together detached architectures**

Prototyping

At the dawn of your project, you're in Unity. You're making random scripts, game objects, iterating rapidly on gameplay. This is mostly client-centric code, because building everything takes time and you want to focus on getting the game's ideas onto the screen quickly.

It's usually too early to start worrying about performance, scalability and workflows. But you might want to test out what it's like to drop-in features like competitive leaderboards or or social scaffolding. At this stage, it's all about moving fast, trying things and finding the fun.

Early Iteration

As you find the fun of your game, the work starts expanding outwards. You add features and additional gameplay loops, art, rules and statistics.

Usually, there's a point in any project where your game designer needs to do something as basic as changing a single value, like the hit points on an enemy. If the game designer works inside Unity, they frequently need a coder to make changes for them–or else it's easy to do things as simple as missing a semicolon that could break code. That's when the engineers are caught in a dilemma:

- **Let designers and other less-technical contributors change code anyway, and just fix it as issues arise?**
- **Start separating code from data and stats, along with tooling to allow designers to change the data and stats?**
- **Have the engineers make changes for the designers?**

In all of these cases, the end result is the same: progress is slowed-down. Agility is diminished. And valuable opportunities to build the things that players care about are missed, because you're spending time on tooling, increasing technical debt and QA.

Expanded Playtesting

At some point, the game is functional enough to start playing. This becomes a critical time in the project because it's usually when games have the opportunity to go from "concepts on the screen" to truly great experiences. But this stage introduces numerous challenges:

- **Addition of identity, social, economic features**
- **Different builds/versions on each developer and playtester's device**
- **Data sets that need to match to specific builds: data for individual development machines, the production playtesting environment, QA, etc.**
- **Often, one person becomes a single-point-of-failure on builds**
- **Distinctions between server code (with their own versioning, build issues, etc.) vs. the compatible client code**
- **The need to distribute binaries on different platforms**
- **QA needs the ability to set up test cases with initial states based on specific data conditions**

The rabbit-hole goes deep on these problems. Making tools that synchronize data with code and traverse the complexity of the development to staging to production deployment process is complicated. Tooling and automating the process takes enormous energy.

Live Operations

LiveOps begins when you start having players from outside the studio. It usually results in exponential rises in the number of players: first a small, invitational alpha community; then, a closed beta; and eventually towards a geographically-locked soft-launch and then an official launch.

New challenges include:

- **Risks of community backlash stemming from new issues like data privacy, GDPR compliance, purchase issues, hackable client code**
- **Designers and LiveOps managers need to set up tournaments, events, new items, offers and other data that are changed "out of band" from code builds**
- **The data/content/code synchronization issue becomes even more critical–errors here could mean the game is down and players are angry.**
- **Customer support is dealing with real customers: suddenly, they need to view player transaction histories, make changes to player data, review purchases, etc.**
- **Measure and learn from performance metrics such as lifetime value (LTV)**

Multi-Game Operations

As a studio or publisher grows and eventually has multiple games, they're often presented with new opportunities and challenges:

- **Reusing shared code with social and economic features across multiple games**
- **Consistent branding and visual language across social interfaces**
- **Calculating aggregate lifetime value across a network of games**
- **Cross-selling and cross-promotion**
- **Benchmarking performance and comparing between multiple titles**

In the next section, we introduce Beamable's integrated, full-stack architecture for LiveOps.

Game Server Architectures

In this section, we'll explain how fixing the problems undermining the speed and efficiency of Unity live game development requires rethinking the architecture for live services. Typical game servers include some combination of:

- **Content servers for on-demand delivery of of content to end users**
- **Multiplayer Relay servers to synchronize client state between players**
- **Application servers for tamper-proof code execution and persistent data**

Beamable's approach unifies all of these services into an integrated, full-stack environment that works harmoniously with the Unity Editor, while leveraging the power of a Serverless Backend to make scaling, DevOps and LiveOps simple.

Principles of the Beamable Architecture

We saw an opportunity to help game developers build faster and manage games more effectively–while bringing people onto a reliable and scalable serverless architecture. We adopted two core principles in Beamable:

- **Full Stack: we saw an opportunity to reorient the developer experience towards the IDE they use on a daily basis, with the ability to organize, debug, locally instantiate and deploy their code without the fragmented mess that's usually associated with game development.**

- **Integrated: the workflows for game developers should work harmoniously with the workflows for day-to-day content creators and LiveOps teams. That meant creating a set of interoperable interfaces, access control systems and content management flows so everyone could work together–without the constant interruptions or being stuck waiting on an engineer to shepherd a change.**

A full-stack, integrated approach dramatically improves the speed of development and your team's responsiveness to change.

Background

The simplest thing a game maker can possibly do is ship a fully self-sufficient binary which is packaged with all its dependencies, can function without internet connectivity, and executes solely within the confines of an end-user device (e.g. Android, iOS, PC, console).

However, this removes the game maker from functionality which can make live operations more effective, responsible to change and improve the player experience.

Common reasons for adopting game servers include:

- **Application code must be tamper-proof for the purpose of multiplayer gameplay, authentication, or real-money purchases**
- **Gameplay content must be delivered over-the-wire and on-demand because it cannot all fit on the end-user app**
- **Gameplay content must be kept hidden until end-users meet qualifying requirements**
- **Gameplay code and content must be able to be updated at will and delivered to the end-user app remotely without requiring the app/binary to be updated**

The use-cases that require server functionality can vary tremendously in terms of their technical and cost requirements. These requirements generally (but not exclusively) fall in the following categories:

- **Simulation Fidelity (Good): How much of the game runs on the server side?**
- **Network Latency (Fast): How quickly are players notified of server-state changes?**
- **Infrastructure Cost (Cheap): How cost-effective is it to operate the servers?**

As such, games frequently use multiple techniques to manage the networked experience of players, oftentimes in combination.

Content Servers

Many games have a vast and diverse data domain: gigabytes or terabytes of data that may contain many different data formats ranging from binary assets to text/json/yaml files which define gameplay entities such as items, puzzles, and rules. This content may be updated on the order of minutes, hours and days rather than on the order of milliseconds.

This type of data is essentially static, and ordinarily only updated by releasing a new version of the application binary which contains new and updated assets. This poses a set of difficult choices for the game developer. Options include:

- **Accept that different players will be playing different versions of the game. This means fragmenting the community and more QA/engineering complexity.**
- **Force the player population to upgrade by deprecating the old version. This inevitably comes at the cost of player churn as not all players will upgrade.**
- **Or, ideally: Architect the app so that it can receive content updates over the wire. This is the approach Beamable takes with its content management system, which is integrated to the workflows content creators are accustomed to.**

Types of Over-the-Wire Updates

Games typically depend on different systems for updates, depending on whether changes affect code, configuration, or content. Beamable allows you to publish versioned content which is atomically updated and validated according to design rules. Some of these content change systems include:

Remote Config

A flavor of content services, commonly referred to as Remote Config, specializes in delivering structured (e.g. json) configuration data to the end-user to tune the behavior of the application. This can be something as simple as showing a different loading screen to indicate the arrival of Halloween, or as far-reaching as altering the balance of the game in a multivariate study to determine the optimal configuration for player retention. As such, remote config may support delivering variants of content to specific segments of players.

Binary Assets

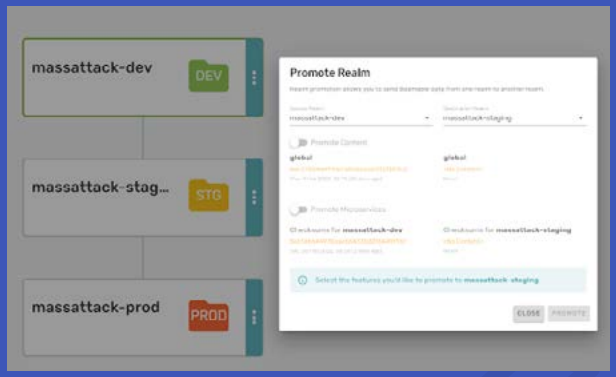
Game makers will often need to pull game engine assets from the cloud. These can be virtually anything, and usually include images, audio files, videos, as well as a variety of formats native to the game engine. Given that game engine formats often vary in their final form from one platform to the next, as well as in terms of their level of detail based on device or user preference – these assets can add up having several variants for a single piece of content. This in turn means a vast amount of data is generated and delivered to end users.

End-User Delivery

Content must be cost effective to store and deliver to edge devices (e.g. smartphones) anywhere in the world reasonably quickly. Content Delivery Networks (CDNs) solve this problem quite effectively, usually in combination with a simple storage solution which can be replicated to edge nodes (e.g. AWS S3 with CloudFront).

Workflow

A typical problem with all of these content management systems is the workflow that links the authoring process to all aspects of the DevOps lifecycle: versioning, testing, deployment, and live services. Beamable consolidates these steps into an integrated pipeline that frees developers from manual processes or labor-intensive scripting.



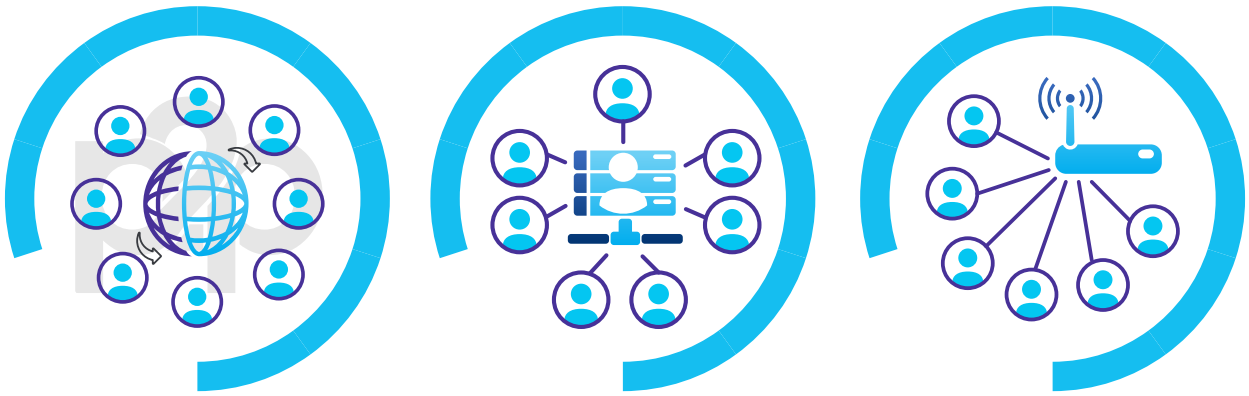
Multiplayer Relay Servers

In cases where a rich world-state needs to be synchronized in real-time, developers will often opt for a client-side simulation. Beamable provides a Relay Server that allows clients to communicate with each other in real-time and also supports tamper-resistant deterministic simulation.

Peer-to-Peer (P2P) versus Relay

The network synchronization can happen in one of several ways:

- **Direct Peer-to-peer: players establish a direct connection to each other, conceptually a mesh network.**
- **Client/Server Peer-to-peer: a player is elected the "host" and is the de facto server as well as a player. All players connect to the "host" player.**
- **Relay Server: Players connect to their nearest relay node server, which is responsible for synchronizing data between players.**



Direct P2P | Client/Server P2P | Relay Server

The main downside of P2P scenarios are as follow:

- **Can be prone to complications if players are behind a NAT router (i.e. don't have a public ip). Requires NAT-punchthrough server or port forwarding setup.**
- **Can be prone to inconsistent network latency, where a single player may cause an overall slow down.**
- **Can be prone to leaking somewhat sensitive information, such as player IP addresses**
- **Host can have an advantage over other players**

Relay Servers

Relay servers, such as Beamable, solve these problems at a cost which remains affordable. They're compatible with client/host topologies, and typically stream data between client peers in real-time over a reliable UDP protocol or over a (TCP) websocket, with no server-side simulation. Clients are responsible for ordering, processing and interpolating between packets to produce a coherent simulation.

Deterministic Simulation

Unless you use a deterministic simulation technology such as Beamable's, the simulation is vulnerable to cheating within the client runtime.

Certain game requirements lend themselves well to a networking model where each client runs a simulation which is guaranteed to be the same on every other client, irrespective of device, cpu chipset, or operating system – this is referred to as a deterministic simulation. By doing this, games can:

- **Cut down significantly on the data that is synchronized over the network to other players**
- **Detect when a player cheats as any tampering with the client state will result in a desync (i.e. divergence between client simulations)**

With a small amount of extra functionality, relay servers can support deterministic networking by keeping track of (a) the player action event log and (b) the current simulation frame. Practically speaking, this means that in addition to the aforementioned benefits, deterministic simulations are not subject to the weakest link problem where a single slow player halts the entire simulation.

In traditional Peer-to-Peer lockstep deterministic simulations, if a player in the game does not commit their intended actions for a given simulation frame (sim frame) by the time the frame is current, the simulation halts for everybody involved. This is not so with a Relay Server – the slow player can fall behind, and catch up later by syncing the event log they missed, and speeding up their client simulation to the current frame.

Application Servers

For use-cases that need guarantees against cheating or require persisting player actions into a shared world state, Application Servers (paired with a database) provide a means to securely execute gameplay code and persist the outcome.

Beamable provides an Application Server technology that is deployed in a serverless manner and subdivided into microservices that are independently scalable, versioned, debuggable and easily integrated into the Unity Editor workflow.

The scale of what is executed on the server may range from small, such as validating a purchase receipt and granting a player an item, to massive, such as simulating game engine physics between hundreds of players in real-time on a persistent world (e.g., an MMORPG).

Similarly, the scale of what is persisted ranges dramatically both in volume and diversity, from a friend relationship between players needed to implement a friend list, to the locations, trajectories, and details of millions of different game entities.

Advantages of Server versus Client Authoritative Code

With few exceptions, games are generally built into executables which are run in environments that are not secured against tampering.

A sufficiently motivated and technically inclined player may both inspect the state of an application on their device, as well as modify it to grant themselves an advantage. Developers therefore have three options:

- **Live with the fact that players can and will cheat**
- **Move the entire executable to a tamper-proof environment (e.g., streamed games such as with Google Stadia)**
- **Execute a subset of the game logic in a secure environment, such as Beamable's Application Server**

Web Application Stacks

Many game developers adapted their backends from off-the-shelf application servers built for websites–Ruby on Rails, Node.js, Django, or similar tech stacks.

Web applications expose APIs that communicate over HTTP requests/responses and are usually stateless. This enables them to be easily scaled horizontally. The web technology ecosystem is arguably the richest in terms of off-the-shelf components and support within standard libraries, making it extremely attractive no matter the programming language or workflow.

The challenge with Web-oriented application servers is the complexity that comes with polyglot language usage, data serialization, and multiple IDEs. On top of that, you need DevOps to provision, scale and deploy cloud infrastructure to host the Web application code. All of this results in workflow fragmentation that adds engineering overhead, brittle processes and compounding technical debt–resulting in slower and slower game development.

Microservices and Serverless

Microservices and Serverless approaches can be defined both conceptually and in terms of their practical implementations. Conceptually:

Microservices are a way to design an application, namely as a set of individually scaled services which own their data and only expose it via a well-defined api.

Serverless is a way to run an application, namely without concern, awareness, or need for dedicated infrastructure (physical or virtualized).

An early approach to serverless was to package code into “lambdas” which are sections of code that execute on the server and scale according to utilization. The problem with lambdas is their execution environment is detached from the IDE, adding additional steps for deployment, versioning, etc. They also tend to be hard to debug because the execution environment is opaque.

Microservice Architectures

Microservices, such as those integrated to Beamable, arose to fix the problems inherent in lambdas while incorporating many of the conveniences present in typical full-stack Web development.

The earliest websites were typically built as monolithic applications: a single deployable application supported by a database (Relational or NoSQL). Although conceptually simple and well suited to smaller apps, over time this often became unreliable and challenging to scale. Monolithic applications are especially vulnerable to the following:

- **Separation of concerns and data ownership is easily violated leading to brittleness**
- **Debugging and Identifying root causes is increasingly challenging as the code base grows in size and complexity**
- **Performance and cost optimization is harder to achieve when services cannot be resourced and scaled individually**
- **Developer agility is impeded as it becomes riskier to add rapidly to a large and complex application**
- **Adopting new or domain specific tech stacks isn't possible without changing the entire app**

Microservices address the above issues, while enabling a new generation of applications that can both scale traffic throughput and supercharge developer agility. Microservices are deployed inside of containers, which are automatically scaled via an orchestration system on top of automatically-provisioned hardware. These same containers can reside on each developer's local environment, which makes it easier to code, test, debug, manage dependencies and onboard new developers.

Beamable automates all of these elements by integrating the coding, management and monitoring of your microservices to the Unity Editor as well as Web-based dashboards.

Game Engine on the Server

There are also use-cases which require twitch multiplayer gameplay (i.e. rich simulation and very low network latency) as well as provide strong guarantees against cheating/tampering. In such cases, a full simulation of the game will run on a server and be streamed to clients (i.e. players), who will in turn submit their intended actions to the server.

The client simulation will attempt to be as close as possible to the game server simulation, often interpolating between data points to infer the current world state before it receives a confirmation from the server.

Both Unity and Unreal game engines provide headless modes, which allow you to run your application on the server. However, this can be computationally expensive due to engine overhead,

or architectural choices made by the game developer. There have been recent efforts, such as the Unity Data Oriented Tech Stack (DOTS), to make it more efficient to run engine code on the server.

Beamable provides a full OpenAPI/Swagger specification for all available APIs, and can therefore be invoked from anywhere. For Unity, Beamable also provides a full SDK which can be used on the server side or on the client side.

Workflow and Scalability Problems

There are two main categories of problems inherent in most game development projects. Most developer workflows are:

Fragmented:

unlike many full-stack application server frameworks, there's usually multiple IDEs, languages, and debugging can only happen within one part of the stack at a time–and things like deployment and shipping compatible versions of the front and backend increase in complexity exponentially.

Detached:

to make things worse, the ecosystems of software that have emerged to support games frequently don't work "out of the box" with each other. The result is that components that ought to save time (dashboards, server-side modules for common live services, content management systems, customer service tools) shift a substantial amount of labor towards systems-integration work. This systems integration work becomes a new source of ongoing technical debt that competes with the creation of the game features that players care about.

Backend as a Service

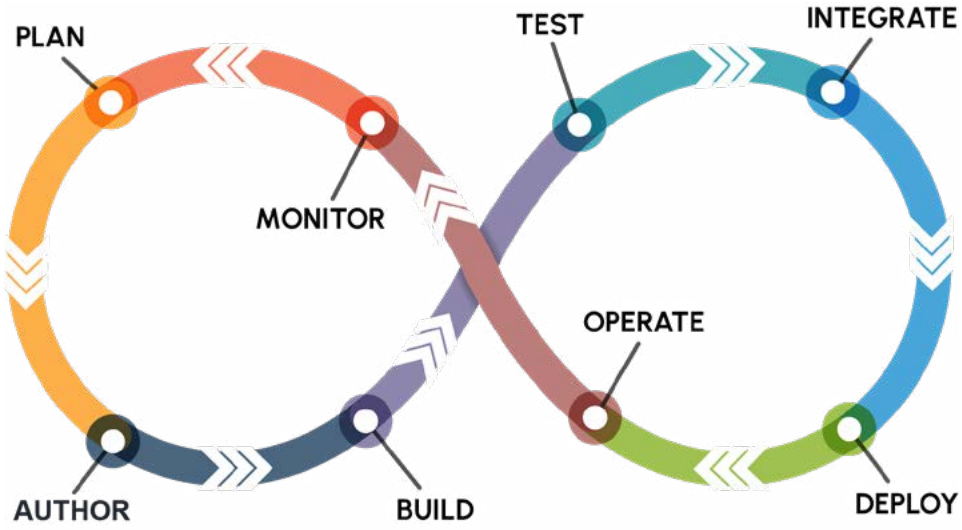
A few years ago, "backend as a service" (BaaS) platforms emerged to help game developers by supplying many of the common live services components: authentication, in-app purchases, social systems, guilds, events, storefront, etc.

The problem with these systems is that they never addressed the underlying problems that made game development so complex in the first place:

- **They introduced even more languages (e.g., CloudScript to control server-based behaviors)**
- **New workflows that didn't work within the tools used by game developers**
- **Brittle and complex methods for scaling, versioning and debugging the custom server code that sits alongside stock components**

BaaS caused work to shift from implementing some of the common backend components to integrating all of the backend and customized parts of game development–while also injecting ongoing complexity to DevOps.

DevOps for Live Games



When building a live services game, developers need to consider the big picture of DevOps. All of the following need to work coherently:

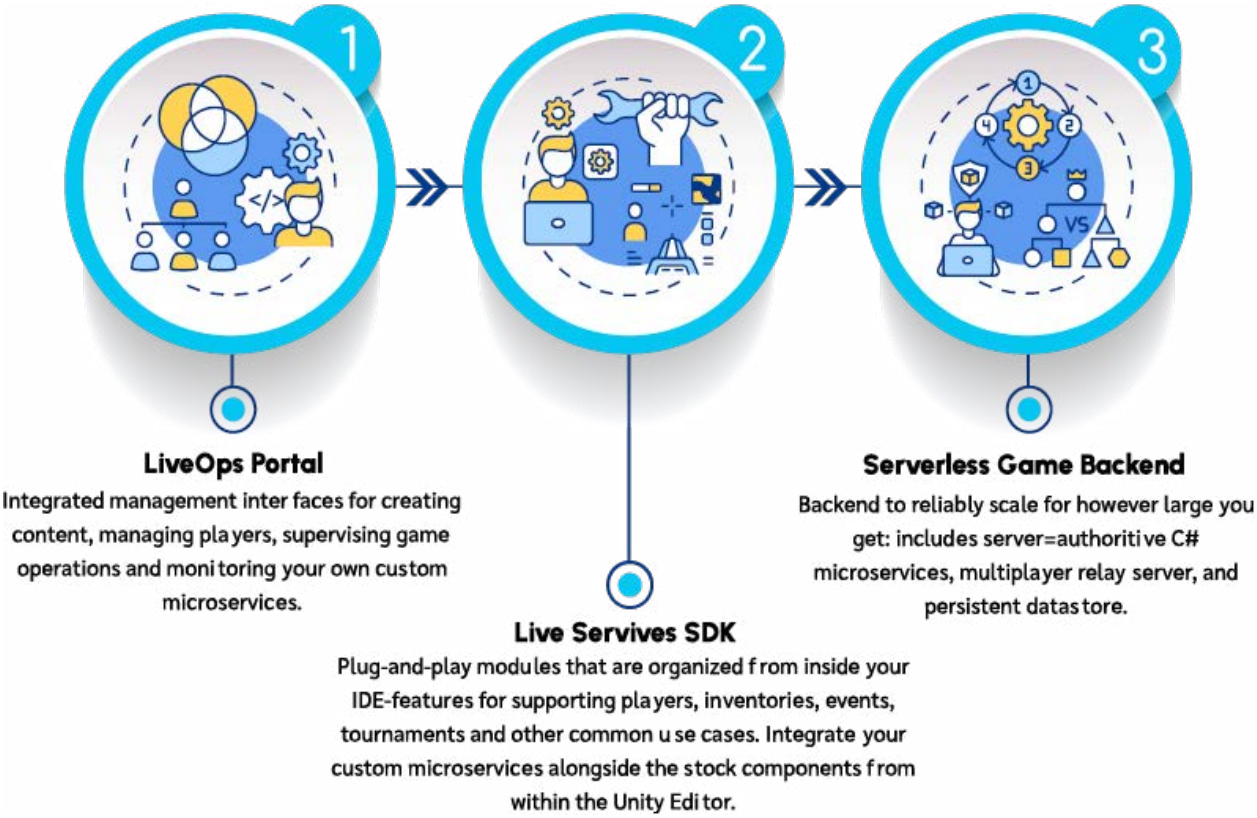
- **Author** new code, microservices, features, content, items, rules, etc. from within the creation environment each contributor is accustomed to.
- **Build** code, content and data into servers and client binaries
- **Test** builds within individual workstations, testing environments, playtesting servers, etc.
- **Integrate** changes from multiple developers, including coders, artists and designers
- **Deploy** changes to different environments, up to and including "production"
- **Operate** the live game (i.e., "LiveOps"): events, promotions, economy, etc.
- **Monitor** results, cost and performance
- **Plan** from the results and plan for the next set of features, code and content

Each of these steps need to have a cohesive process that simplifies implementation, workflow, and management interfaces–while empowering the right people to make decisions.

In the following section, we'll present a comprehensive overview of Beamable's architecture and how we set out to solve the fundamental problems of live services games.

Beamable: Integrated, Full-Stack LiveOps

There are three main layers to the Beamable architecture: the Serverless Game Backend, which hosts the microservices; the Live Services SDK which provides a wide range of off-the-shelf features that you'll need for your game; and a LiveOps Portal for day-to-day management by all of the users in your studio.



One Magic Line of Code Speeds Development

Beamable's integrated approach results in simplicity for your developers. After installing the plug-in for Unity, one line of code gets you building games faster:

```
var beamable = BeamContext.Default;
```

The immediate effect of adding the line of code into your game is that you gain a frictionless identity system for your players, and the dashboards in the LiveOps Portal start updating with your key engagement metrics. For developers, they gain access to a palette of drag-and-drop live services, and may now write server code that works inside Unity Editor's debugging and deployment.

DevOps for Live Games

Here is a chart of how Beamable improves the way you work:

Common Workflow	With Beamable's Integrated, Full-Stack LiveOps:
Code Authoring	Use the Unity Editor as the central place for authoring both client and server code in C#, with full access to all of the plugins and tooling you're used to–for both client and server-authoritative components.
Creating Server Workloads	Execute flexible, secure and automatically-scaling workloads using cloud-based microservices. Use whatever third-party services (drivers, databases, etc.) you prefer.
Multiplayer	Distribute and synchronize multiple game clients with cheat-resistant deterministic multiplayer support.
Debugging	Create a local instance of all your server-authoritative microservices alongside your client code. Trace code up-and-down the stack, set breakpoints and watch variables anywhere, etc.
Deployment	Ship server-authoritative code without having to provision or manage servers, networks, load balancers, scaling rules, monitoring/logging software, build processes.
Monitoring	Observe the performance of your custom microservices through a web-based portal or from within the Unity Editor to identify opportunities for improvements in latency and compute consumption.
Implementing Live Services	Common live services (events, guilds, players, inventories, etc.) are available through the Live Services SDK. Drag-and-drop from within the Unity Editor, and manage their configuration and data either from inside the editor or from a Web-based portal.
Using Persistent Data	Easily store key/value pairs that cover most data storage use cases from within Beamable microservices, and perform powerful queries including geospatial and time series analysis–or connect with your own database and storage services for more specialized use cases. Configuration datasets that change the production environment can be managed through the integrated deployment process.

Common Workflow	With Beamable's Integrated, Full-Stack LiveOps:
Content Authoring	Integrated tools allow content creators to use a web-based form, a spreadsheet or the Unity Editor to create new items, events, variable changes, etc.
Versioning	Use the standard versioning tools you prefer, to organize all of your code, content and data into comprehensive packages–without the clunky build processes to synchronize different sources of truth.
Monitoring	Observe the performance of your custom microservices through a web-based portal or from within the Unity Editor to identify opportunities for improvements in latency and compute consumption.
Live Operations	Web-based forms and access control for managing the key data structures inside Beamable: players, inventories, purchase histories, events, content, etc.

Comparing the Developer Experience

Here is a quick comparison to how a C# programmer would work with the full-stack of code in a game compared to other backend-as-a-service (BaaS) environments like GameSparks or PlayFab:

	Beamable C# Microser-vices	Amazon AWS Lambda	GameSparks Cloud Code	Microsoft PlayFab Cloud Script
C# Client Code	●	●	●	●
C# Server Code	●	●	◐	
Can Run Locally During Development	●	◐		
Full Debugging Support (Debug.Log, Breakpoints, etc...)	●	◐		
Source Control Integration	●			
Unity Workflow (ScriptableObjects, etc...)	●			

● Fully integrated support

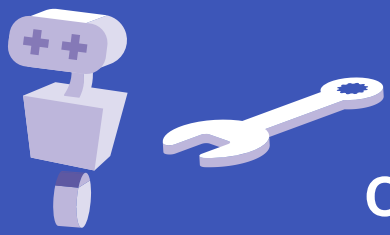
◐ Detached support, requires setup and manual workflow

In the following sections, we'll dive into how each of these workflows actually function within Beamable.

Code Authoring

Despite the "integrated" in "integrated development environment" (IDE), the authoring of server-authoritative code is almost entirely fragmented from the development process. If you're using Unity for game development, it is the Unity Editor that is the most efficient IDE to work from within–regardless of what part of the stack you're building.

To accomplish this, we started by making C# the primary scripting language for server-authoritative functions, alongside the native use of C# within Unity itself.



Creating Server Workloads

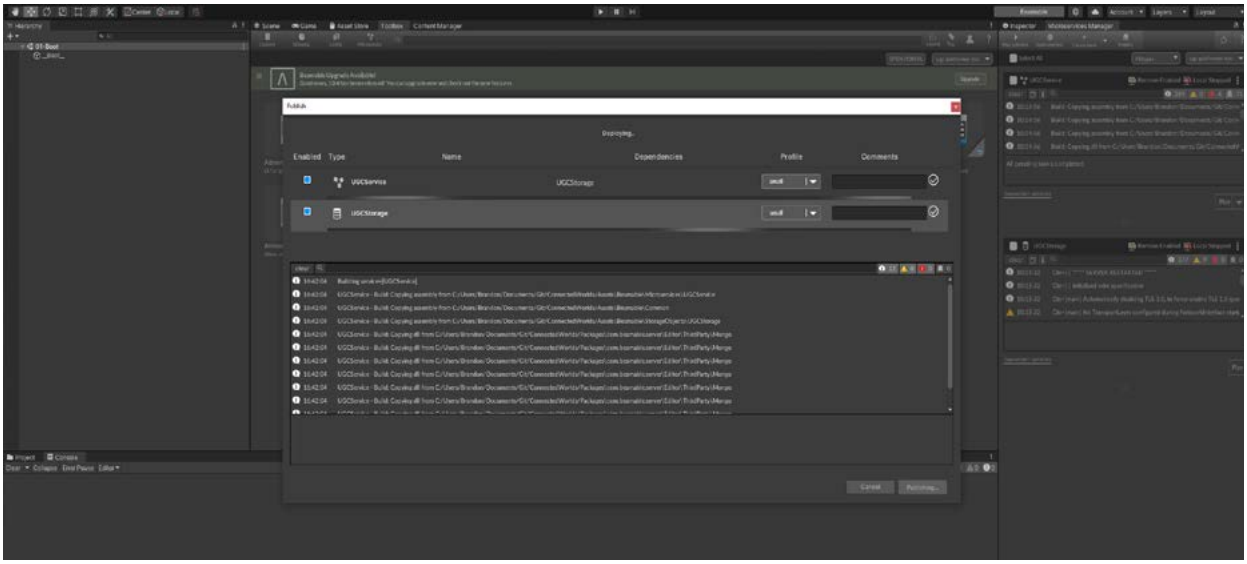
With Beamable, C# script implements your servers in a way that's consistent with how you're coding the front-end of your game. These server-authoritative workloads allow you to create multiplayer features that require coordination and rules between players, as well as single-player features that require greater security (for example, execution of server-based rules or cooldown timers that would ordinarily be hackable within a game's front-end code). Unlike lambdas, these microservices are flexible and allow for the use of third-party APIs and DLLs that you need for your game.

In keeping with Beamable's simplicity, the code to implement The C# is very simple:

```
[Microservice("HelloWorld")]
public class HelloWorld : Microservice
{
      [ClientCallable]
      public void ServerCall()
      {
            // This code executes on the server.
      }
}
```

Creating Server Workloads

Deployment of the microservice is similarly simple, and fully-integrated with Unity Editor:



Creating a C# Microservice from inside Unity Editor

Multiplayer Relay Server

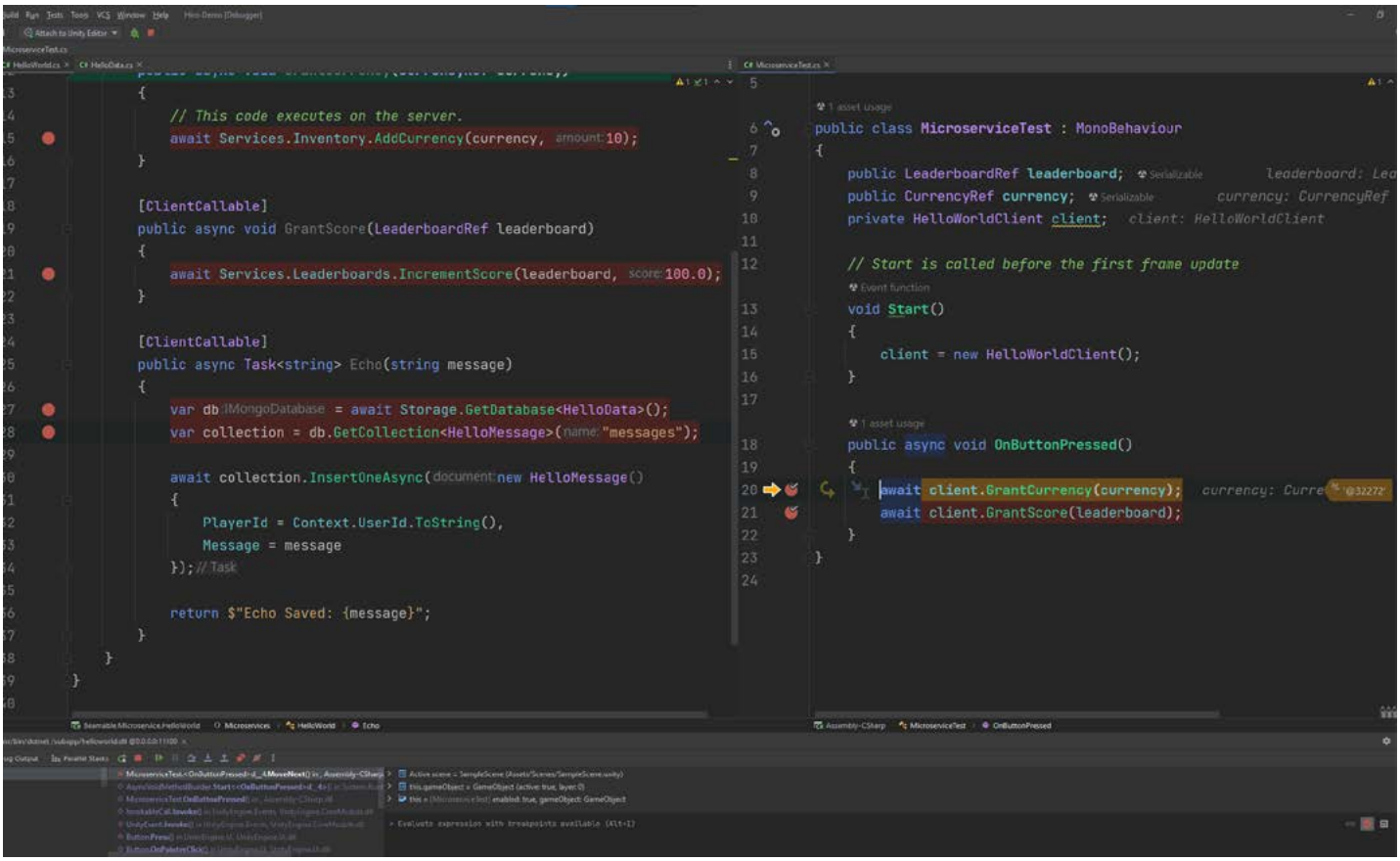
Synchronize user inputs across the network and distribute changes across each game client within an active session. Relay Server is ideal for multiplayer games such as real-time strategy, tower defense, MOBAs, card battlers, auto chess and others with deterministically-advancing gameplay.

Relay Server keeps track of the simulation frame for each player, maintains an event log, and conducts timekeeping. When building with Relay Server, you have the option to either use it as a simple communication channel between games (for cases where you aren't concerned about cheating) or you can implement cheat-resistant deterministic multiplayer, which allows the game clients to securely reach consensus about gameplay state between multiple players. In the latter case, attempts to hack the client result in falling out of sync and the cheater is excluded from further play.

Debugging

Because Beamable consolidates the full stack within one language and one IDE, it transforms the way you debug your game. You can do all of the ordinary debugging operations from any layer of the stack, including your server-authoritative microservices: set breakpoints, watch variables, trace execution. You don't need to rely on complicated multi-stack setups or depend on time-consuming logging to isolate issues.

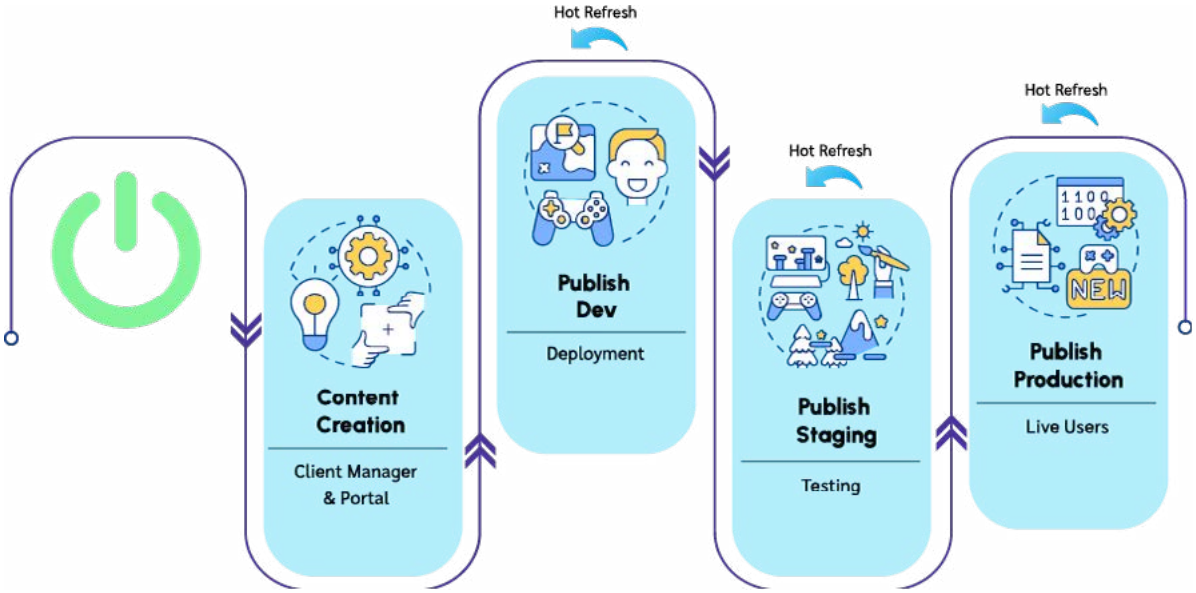
In multiplayer games that use Relay Server, you'll find the ability to replay event logs are helpful with reproducing issues.



Debugging a Microservice alongside its front-end code in Unity Editor

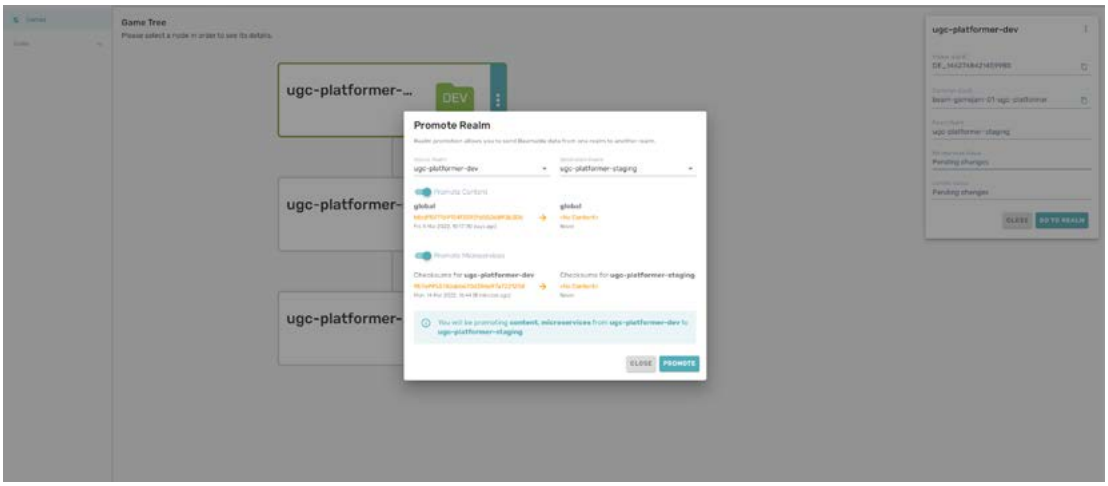
Deployment and Versioning

Because the microservices are fully-managed, you don't need to provision servers or do any of the complicated work normally associated with scaling your servers: no networks to configure, load balancers to setup, scaling rules to write, monitoring/logging software to install, build processes to script, etc.



Beamable automates the steps involved in moving all of the interdependent client code and microservices through all of the steps of your continuous integration, quality assurance and release process.

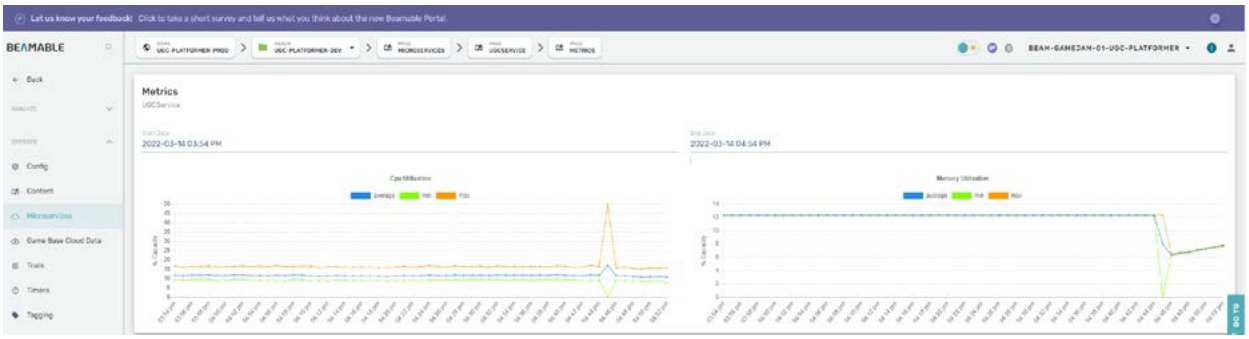
These C# code modules inside your microservices are deployed independently of the front-end code (there's no remnant of the backend within the code you ship to players), while maintaining a cohesive set alongside your client binaries.



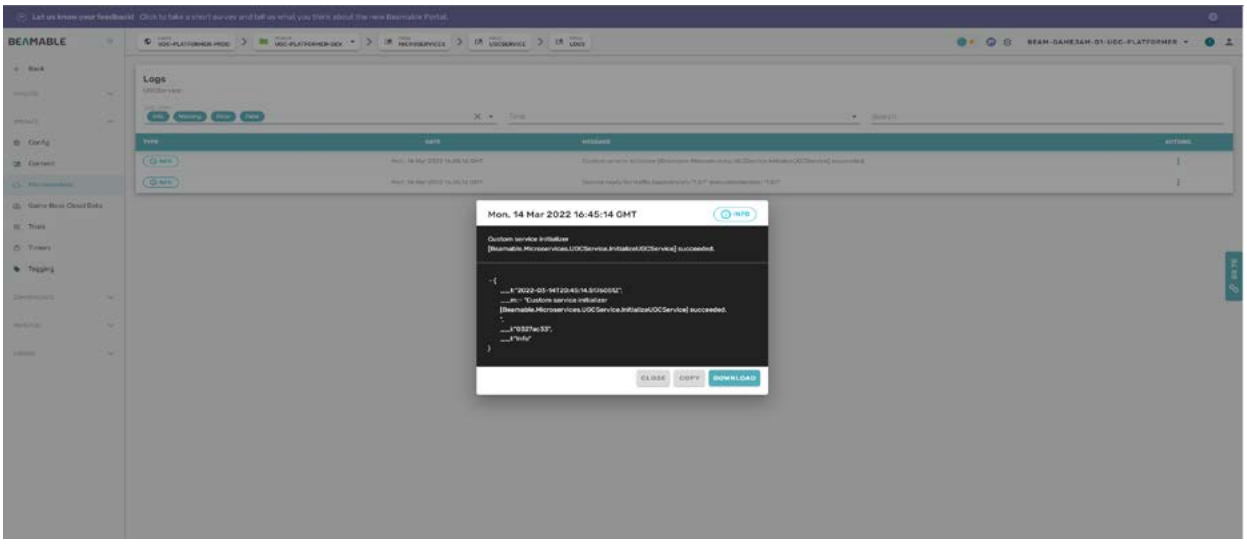
Promoting a deployment of content and code

Monitoring

As you create your microservices, you'll want to keep an eye on opportunities to improve your code's performance. Beamable's microservice management features provide a web-based interface for logging and monitoring of your code.



Monitoring Microservice performance from the integrated LiveOps portal



Viewing Microservice logs from the integrated Live Ops Portal

Using Persistent Data

Beamable's Live Services SDK provides many of the common patterns you'll need for working with persistent player and world data–managed inventory, player stats, leaderboards, identity, etc.

However, many games also need to create their own unique data. Beamable offers a simple solution: leverage the power of off-the-shelf MongoDB within your microservices to meet those needs. We call that **Microservice Storage**. With it, you can define your data structures in Unity and leverage C# Microservices to store and retrieve key/value pairs–and perform powerful queries including geospatial and time series analysis from your databases and collections. You can also access logs, look into the DBs or visualize live metrics from our LiveOps portal and a MongoExpress interface. All of this happens in a fully-managed environment so that you save countless hours of DevOps time that you'd normally have to spend integrating detached workflows and development tools.

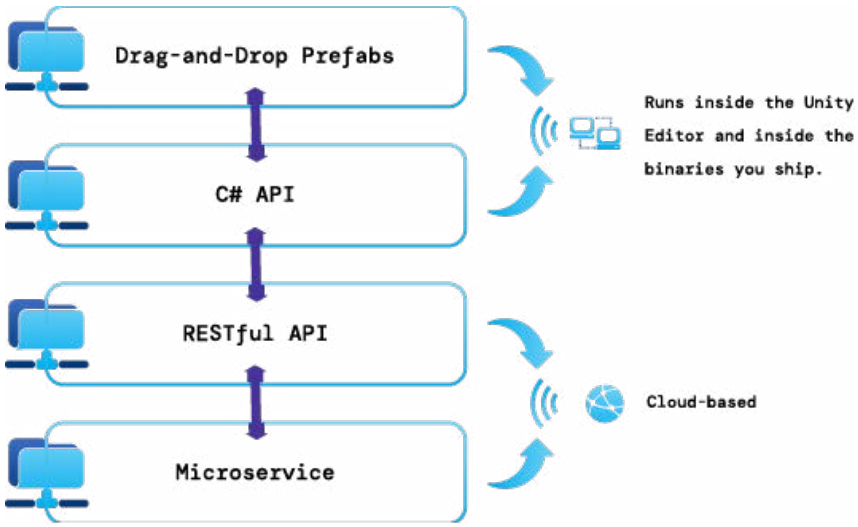
Because microservices allow you to utilize any third-party datastores or APIs you want, you could also make use of any persistent data system you prefer for those cases where you require specialized stores that optimize around certain kinds of systems.

Implementing Live Services

Beamable comes with a Live Services SDK that allows you to rapidly add common social, commerce and content features to your game. These services operate within the same highly-scalable microservice architecture that you build your own custom services in, giving you the confidence to reliably scale up to whatever number of players you have.

You can hook into these live services at several layers to help you build your game faster:

- **Many of the live services come packaged with a Unity prefab that brings drag-and-drop simplicity to the game development process.**
- **You can call each service via C# SDK that lets Unity developers operate in a native-friendly environment.**
- **A RESTful API is available in cases where you need to interact with the live services from outside the context of a C# project. This is often helpful for integrating with legacy code in other languages, third-party services, websites, etc.**



Customizing Prefabs

For many of the “out of box” live services features, we package prefabs that make it easy to get a feature up-and-running. Beamable Unity Style Sheet (BUSS) enables developers to change the style of the drag-and-drop prefabs. Developers can declare their desired style using familiar concepts from Web and UIToolkit. The styles are then applied in a cascading fashion to prefabs provided by Beamable. Styles are capable of expressing colors, gradients, borders, rounded edges, shadows, shapes, textures, fonts, and other text properties. Due to the cascading application of style, developers can override specific sections of their game. The styles are applied at runtime, and can therefore be mutated given game state or business needs, facilitating custom themes during events or holiday seasons.

There are two other major areas of prefab customization, layout, and behavioral. BUSS allows developers to adjust the layout of their prefabs by using the standard Box Model layout approach that is popular in other technologies like WebKit and UIToolkit.

The drag-and-drop prefabs are built from components that can be inspected and reassembled to form new behavioral variants, or you could choose to build your UI from scratching using the C# API once your requirements and aesthetics have stabilized.

Summary of Live Services

The following is a summary of the out-of-box services you can rapidly incorporate into your game:

Identity

This service includes maintaining player identity across all of the connected services (Managed Inventory, Chat, Mail, Events, etc.), as well as authenticating players and federating their identity with common third-party services like Facebook, Apple, Google and Steam–or allow username/password authentication.

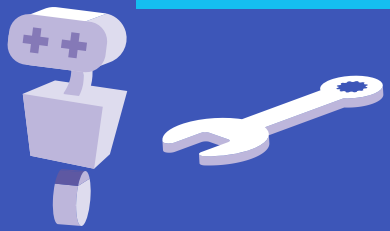
Stats

Player identities can be customized with your own statistics, allowing you to customize players for use with your own game (for example, defining a level or set of attributes for each player). This can be used for storing progress, as well as segmenting player populations for several applications: A/B tests that experiment with game changes for different audiences, or delivering different offers to different players.

Managed Inventory

Managed Inventory is a richly featured API which allows players to own items and currencies -- two behaviorally different entities which are ubiquitous in games. Items can be conceived of as Non-Fungible, meaning that each item instance has a type from which it inherits static properties which are universal to all instances in addition to dynamic properties which are specific to the item instance, making it potentially unique (e.g. a sword with a randomly generated +10 damage modifier). Currencies, on the other hand, can be conceived of as Fungible -- meaning that they are completely interchangeable (e.g. 10 gold). Currencies can also be used to represent stackable items (e.g. 10 Potions).

The Managed Inventory allows developers to atomically modify a



Managed Inventory

player inventory: that is, perform multiple operations (add, modify, remove) in an all-or-nothing manner. A common example of this is when granting a player an item in exchange for currency -- you want to do both at the same time or neither.

Finally, a substantial benefit of the Managed Inventory is that it is natively integrated with many other Beamable services, including Purchases, Announcement, In Game Mail, Event Rewards, and many more.

Note: Managed Inventory runs in a high-performance database, and the use of non-fungible and fungible inventory shouldn't be confused with blockchain implementations. However, the concepts do map to the concepts of non-fungible tokens (NFTs) and fungible currencies, and a developer could build on top of Managed Inventory to federate a subset or all of the inventory entities to the blockchain.

Stores

Stores allow you to create storefronts and offers in your game that utilize real-money or in-game currency purchases. You can configure store catalogs (SKUs), pricing formulas, and organize special and limited-time offers for players and created localized versions of store text. Offers may be segmented to different players according to whether they've met specific requirements (such as statistics linked to their identity), whether they've previously purchased a specific offer, membership in certain cohorts, or whether they're under specific purchase limits.

Payment gateways including Apple, Google, Facebook Steam and Windows Store are supported–or you can use our coupon system to let players redeem items for codes. Beamable's backend include robust receipt verification and anti-hacking systems.

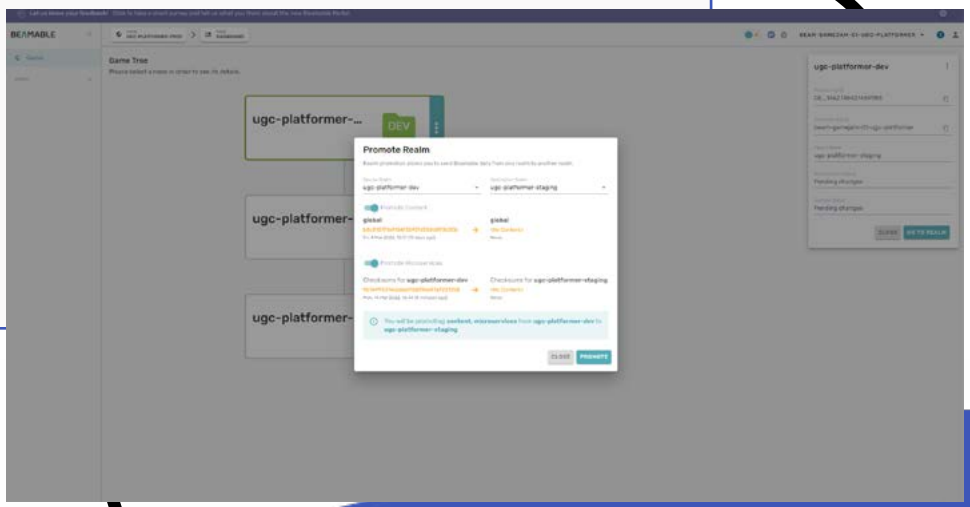
Events

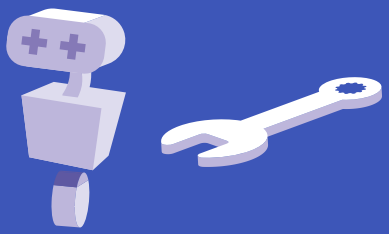
Players compete for a limited time to win points towards a leaderboard. Tournaments are a type of event that include progression/regression systems, stages and tiers to enhance competition. You can deliver rewards to players (from the Managed Inventory service) based on their rank in the rewards.

Content

Games usually include significant amounts of custom-content. This service allows you to create your own custom content objects that synchronize with the deployment of other code and data for your game.

- **Beamable provides this service integrated inside of the Unity workflow. Furthermore, content deployments are:**
- **Atomic: All or nothing deployments ensure that if the upload/ deployment fails midway players will not be left in a halfway point including new and old content.**
- **Versioned: Content is versioned and can be both integrated from and diffed against related environments (e.g. dev-> staging) so that it is easy to test, deploy, and rollback.**
- **Validatable: Syntactic, Semantic, and Referential validation is supported such that developers write custom rules and can be confident in the correctness of the content.**
- **Viewable: Content can be viewed from the web portal, unity, as well as inside of google sheets via a plugin Beamable publishes.**





Cloud Save

Allow players to store progress. The Cloud Data is fetched online and stored locally; scoped by game and player. As changes are detected, the system automatically keeps data in sync.

Messaging

Includes one-to-one in-game mail; one-to-many announcements; and notifications to the player's device.

Chat

Allow players to communicate with each other within the game.

Groups

Implement short-term player grouping (like "parties") or persistent groupings such as guilds.

Connectivity

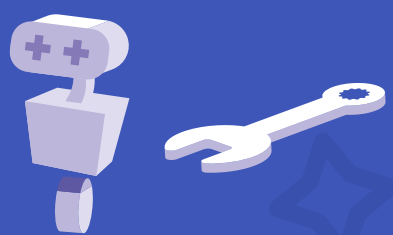
Check on network status within the game to provide error messaging or enable offline mode for some features.

Leaderboards

Create leaderboards for players to rank and compare themselves to each other according to the stats you record.

Matchmaking

Match players with each other according to experience (Elo) or competitive ranking.



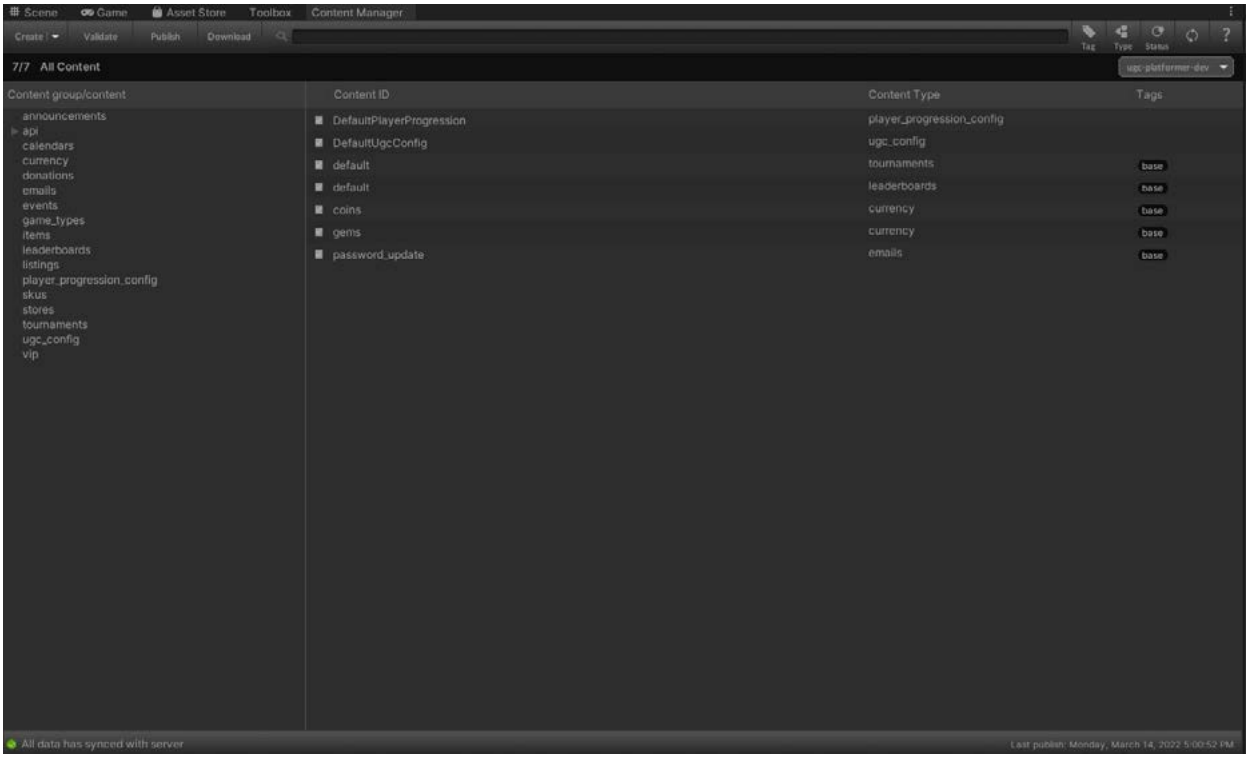
Content Authoring

Day-to-day content creation (making new items, announcements, events, etc.) involves a diverse set of stakeholders at a game studio. This can include:

- **Game developers who are comfortable with sitting down to the Unity Editor**
- **LiveOps personnel who prefer form-based inputs via a Web browser**
- **Game designers who like working within a spreadsheet to manage itemization, stats, power curves, etc.**

Beamable provides all of these content-creation workflows, while also integrating them with all of the other systems that are organized around code. That means syncing data with your servers, uniting changes with deployment and version control works as part of a cohesive and easy-to-manage system; no more complicated build scripts and backend tooling to get all your code and content working together as it makes it from various developers heads and onto your players' screens.

In addition, once content is published, Beamable initiates a "hot update" to the game clients, so that they refresh affected data to reflect new content without requiring the download of a new game client (or even a reload).

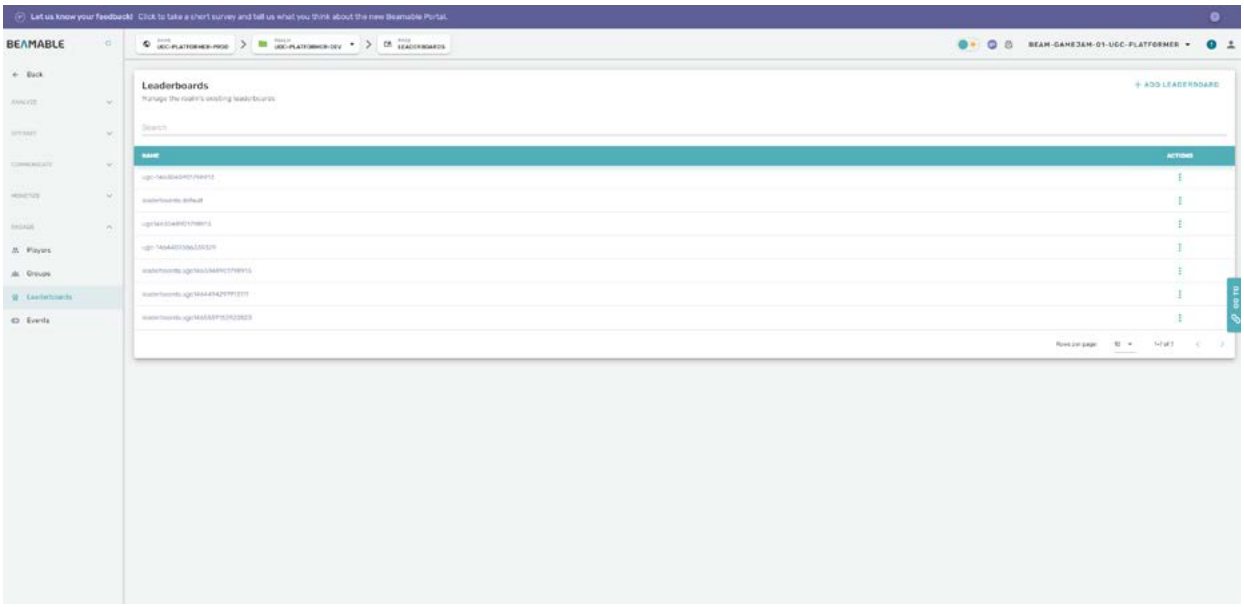


Versioning

Versioning fragmented sets of code, data and content is often a nightmare–resuting in complex repositories, check-in scripts or separate processes that need to be managed independently. With Beamable, you'll just be using the same version control system you normally use, and it will cover all of the data and content that Beamable manages.

Live Operations

Beamable's web-based LiveOps Portal provides a day-to-day management interface for all your users, microservices, and the features managed by the Live Services SDK. This allows a number of different users and use cases to interact with the portal, including your game developers, customer service, QA, designers and product managers.

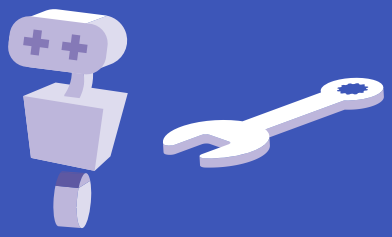


When you make a change to the any data via the portal, a message is transmitted to the affected users who are playing the game live, causing a "hot" change to things like inventories–without requiring a game reload.

Here are the major parts of the LiveOps interfaces you can manage by the portal, and some of the users and use cases covered by them:

Section	Features	Users & Use Cases
User Management	Add and remove users; grant permissions to different sections.	Administrators
Announcements	Post announcements to your players.	Product managers, Game Directors, Community Managers
Players	Look up players by ID, email, or third-party account associations (Facebook, Google, etc.). Drill into associated data such as Groups, Inventory, Stats, etc.	Customer Service (as a first step to looking into an issue), QA (to investigate issues and setup test cases)
In-App Purchases	View accounting history for each player.	Customer Service (to resolve customer support issues), QA (to investigate issues), Game Developers (as a debugging tool)
Stats	Change your customized data fields (as configured by the Stats service of the Live Services SDK) associated with each player.	Customer Service (to resolve customer support issues, e.g., you could change a player's level), QA (to investigate issues and setup test cases), Game Developers (as a debugging tool), Game Designers (to simulate designs and how they're impacted by different player stats)
Cloud Data	View and change the cloud data stored via the Cloud Save service for specific users.	Customer Service (to change a player's state in the game, in response to inquiries), QA (to investigate issues or setup test cases), Game Developers (as a debugging tool)
Inventory	View and change the inventory (currency and items) associated with specific players.	Customer Service (to grant rewards, or correct issues), QA (investigate issues, setup test cases), Game Developers (as a debugging tool)

Section	Features	Users & Use Cases
Groups	View and change the composition of a group	Customer Service (remove members, reassign control/permissions in response to issues), QA (setup test cases), Game Developers (as a debugging tool)
Leaderboards	Create leaderboards, view/change players on the leaderboard	Game Developers (to create new types of competitive systems), Customer Service (to investigate and resolve issues), QA (investigate issues and setup test cases)
Tournaments	Create competitive events, view/change state of players in the tournament, configure reward tables	Product Managers (to set up live events), Customer Service (to investigate and resolve issues), QA (investigate issues and setup test cases)



Beamable's Architecture Vision

If you're building a live services game, then Beamable's architecture enables you to build games faster.

The "magic line of code" that begins your journey is the result of a set of architectural principles:

- **Full-Stack: in that it is built around a set of live services, front-end and LiveOps components that let you work with C# throughout your game development.**
- **Integrated: rather than a set of detached components you're left to connect on your own, the Beamable services work out of the box–and with Unity Editor. Development, debugging and deployment work in harmony.**

Whether you're creating a highly-interactive multiplayer game, a game with social or economic features, or even a single-player game with a community built into the experience–then Beamable can help.

Architecture for Scalable, Reliable and Efficient Game Operations

Build Games Faster:

One Line of Code enables Integrated, Full-Stack LiveOps for Unity

BEAMABLE