How to Set Up Unreal Cloud DDC

Set up a Cloud DDC server for your team.

Unreal Cloud DDC is a distributed storage service primarily used as a Cloud-based **Derived Data Cache (DDC)**.

Unreal Cloud DDC stores compact binary objects in a content-addressable store that can be replicated across the world. Compact binary objects are self-describing key-value objects (similar to JSON) that have good binary serialization and an extensive type system.

One of the key types are **Attachment** types, which support objects that reference a large blob without containing it. Blobs are binary data files, such as images, audio files, or other multimedia content. These attached payloads are content-addressed, meaning the hash of the payload is used as its identifier. Content addressing is a common practice in distributed storage systems, most notably in git. Content addressing provides the ability to generate an immutable identifier of a payload to quickly determine if it is already available in a cache and if it needs to be replicated somewhere else.

Unreal Cloud DDC can significantly help teams speed up their cook processes in Unreal Engine for cases where they do not have access to a primed local (or fileshare) cache, for instance, when users are remote.

License

The source of Unreal Cloud DDC is covered by the regular Unreal Engine source license.

We provide container images on GitHub. These containers are provided under the MIT license.

Directories

- **Jupiter** - Most of the Unreal Cloud DDC functionality.
- **Jupiter.Common** - Lib - Legacy library with functionality that could be used outside of Unreal Cloud DDC.
- **Helm** - Helm charters for all components.
- **Benchmark** - Useful templates for doing simple HTTP benchmarking user SuperBenchmark (sb), as well as a Docker container that can be used to run benchmarks using Vegeta.
- **Composes** - A collection of different Docker compose files for easier local deployment and testing.

Dependencies

- DotNet Core 6 (and Visual Studio 2022 or VS Code)
- Docker
- Scylla

- Blob storage (S3, Azure Blob Store or a local filesystem)

Other Useful Things

- MongoDB
- Minio
- Docker Compose

Functional Test Requirements

Before starting the tests, start the prerequisite services using this command line:

Command line

```
docker-compose -f Composes\docker-compose-tests.yml -p jupiter-test up
```

Copy full snippet

Running Locally

If you want to try out Unreal Cloud DDC, start it up using `docker-compose`. Note that, as Unreal Cloud DDC supports multiple different backends, we provide different compositions for each use case.

To start, run:

Command line

```
docker-compose -f Composes\docker-compose.yml -f Composes\docker-compose-aws.yml up --build
```

Copy full snippet

You can replace the AWS compose file with `docker-compose-azure.yml` if you want to run with services closer to what is available on Azure.

Docker compose setups disable authentication to make it quick to get started. Generally, we recommend that you hook Unreal Cloud DDC up to an OIDC provider before deploying this.

Deployment

Unreal Cloud DDC currently only runs in production on AWS, but the storage and database requirements are generic and abstracted.

We have basic (untested) support for Azure services.

We provide helm values (under `/Helm`) that we use for Epic internal deployments to Kubernetes, but Kubernetes is not a requirement.

Docker images are published to the Epic Games GitHub organization.

Scylla

You will need to set up a Scylla cluster for Unreal Cloud DDC to talk to.

Unreal Cloud DDC supports running with Scylla Open Source (which is free) or their paid offerings. The paid offerings can help reduce the amount of effort you need to put into managing the cluster, as the Scylla manager will help you with maintenance tasks.

To learn how to set up a multi-region cluster, see Create a ScyllaDB Cluster - Multi Data Centers (DC) in the Scylla documentation:

Download the open-source version of Scylla here.

Scylla provides machine images for use in cloud environments.

AWS

This is the most tested deployment form, as this is how we operate it at Epic. The helm chart we install into each region's Kubernetes cluster is provided in this repo.

On-Premise

Unreal Cloud DDC can be deployed on-premise without using any cloud resources. You can either set up a MongoDB database for this (if you only intend to run this in a single region) or Scylla if you intend to run it in multiple regions but still on premise.

If you are starting with one region but might expand later, we recommend using Scylla, which will allow you to scale directly, while MongoDB would require you to drop all your existing state.

Azure

To deploy on AWS, you need to set Azure as your cloud provider and specify the `Azure.ConnectionString` setting with a connection string to your Azure Blob Storage.

Testing Your Deployment

Once you have a deployment up and running, you can connect to the machine and run curl commands to verify it's working as it should.

First, you can attempt to use the health checks. These should return the string `Health`:

Command Line

```
curl http://localhost/health/live
```

Copy full snippet

Next, you can attempt to add and fetch content into a namespace. This will insert a test string (`test`) into the `test-namespace`. You may need to use a different namespace depending on your setup. This also assumes you have authentication disabled. It should return a 200 status code with an empty "needs" list.

Command Line

```
curl http://localhost/api/v1/refs/test-namespace/default/0000000000000000000000000000000000000aa -X PUT --data 'test' -H 'content-type: ap
```

Copy full snippet

After that, you can attempt to retrieve this object. This should print the 'test' string and a 200 status code.

Command Line

```
curl http://localhost/api/v1/refs/test-namespace/default/0000000000000000000000000000000000000aa.raw -i
```

Copy full snippet

Monitoring

We use Datadog to monitor our services. As such, Unreal Cloud DDC is instrumented to work well with that service. However, all logs are delivered as structured logs to stdout, so any monitoring service that understands structured logs should be able to monitor it quite well.

Health Checks

All Unreal Cloud DDC services use health checks to monitor themselves, any background services they may run, and any dependent services they may have (DB / Blob store, etc).

You can reach the health checks at `/health/live` and `/health/ready` for live and ready checks respectively. Ready checks are used to verify that the service is working. If a ready check returns false, the app will not get any traffic (load balancer ignores it). Live checks are used to see if the pod is working as it should. If a live check returns false, the entire pod is killed. This only applies when running in a Kubernetes cluster.

Authentication

Unreal Cloud DDC supports using any OIDC provider that does JWT verification for authentication. We use Okta at Epic, so this is what has been tested, but other OIDCs should be compatible as well.

To configure authentication. Set up the IdentityProvider (IdP), and then set up authorization for each namespace.

IdentityProvider Setup

Specify auth schemes in the `auth` settings.

Auth Settings

```
auth:
defaultScheme: Bearer
schemes:
Bearer:
implementation: "JWTBearer"
jwtAudience: "api://unreal"
jwtAuthority: "<url-to-your-idp>

```

Copy full snippet

We recommend naming your scheme `Bearer` if it's your first and only scheme. You can use multiple schemes to connect against multiple IdPs. This is mostly useful during a migration.

The implementation field is usually `JWTBearer`, but we do offer an `Okta` field if you are using Okta with custom auth servers. For Okta using the org auth server, you will need to use `JWTBearer` as well.

Namespace Access

Access to operations within Unreal Cloud DDC is controlled using a set of actions:

- ReadObject
- WriteObject
- DeleteObject
- DeleteBucket
- DeleteNamespace

- ReadTransactionLog
- WriteTransactionLog
- AdminAction

These can be assigned either per namespace using the acls in each namespace policy or by assigning them to the acls in the Auth settings (which applies them across all namespaces and for operations not associated with a namespace).

Below is an example configuration that sets transaction log access for users that have access to it, admin access for admins and then per namespace access.

```
auth:
acls:
- claims:
- groups=app-ddc-storage-transactionlog
actions:
- ReadTransactionLog
- WriteTransactionLog

- claims:
- groups=app-ddc-storage-admin
actions:
- ReadObject
- WriteObject
- DeleteObject
- DeleteBucket
- DeleteNamespace
- AdminAction

namespace:
policies:
example-namespace:
acls:
```

```
23 - actions:
24 - ReadObject
25 - WriteObject
26 claims:
27 - ExampleClaim
28 - actions:
29 - ReadObject
30 - WriteObject
31 claims:
32 - AnotherClaim
33 open-namespace:
34 acls:
35 - actions:
36 - ReadObject
37 - WriteObject
38 claims:
39 - "*"
```

Copy full snippet

If you specify multiple claims in the claims array, these are ANDed together, which means all the claims need to evaluate to true. A claim statement like `A=B` requires claim `A` to have value `B` (or contain value `B` in the case of an array).

You can also specify the `*` claim, which grants access for any valid token no matter which claims it has. This is mostly for debug and testing scenarios, and shouldnt be used for production data.

Networking Setup

Unreal Cloud DDC derives a lot of its performance from using your normal internet connection and not relying on a VPN tunnel. Therefore, we strongly recommend that you expose Unreal Cloud DDC on a public internet endpoint. Use HTTPS and set up your authentication as described on this page to prevent anyone from accessing this data.

Unreal Cloud DDC also provides multiple ports that you can use to control the access level of the API.

Public Port

This is the port that you should expose to the public internet and what most users should be using to access the service. This port does not expose some of the more sensitive APIs (enumerating all contents, for instance). This is exposed on port `80` and as `http` within Kubernetes. If you are only operating a single region, this is the only port you need to expose.

Private Port

This can also be called the Corp port. Use this if you have a route exposed to your intranet. Its purpose is to have certain sensitive namespaces that are only exposed to users on your intranet. Use the `IsPublicNamespace` (set to false) in the namespace policy to enable it.

> We do not recommend using this for DDC, as it prevents users WFH (Working From Home) to access the namespace without a VPN (which is typically too slow for DDC use cases).

This is typically exposed on port `8008` and as `corp-http` within Kubernetes.

```
"PublicApiPorts": [ 80, 8081 ],
"CorpApiPorts": [ 8008, 8082 ],
"InternalApiPorts": [ 8080, 8083 ]
```

Copy full snippet

Internal Port

The internal port is only needed to be reachable by other Unreal Cloud DDC instances. This exposes everything that the private port does, but also certain APIs that are deemed sensitive (primarily, enumerating content via the replication log).

This is exposed on port `8080` and as `internal-http` within Kubernetes.

We recommend keeping this ingress only to other Unreal Cloud DDC instances via a private VPC or using anIP range allowlist or similar.

Note that this port is primarily used for the speculative blob replication (see *Blob Replication Setup* on this page).

Common Operations

Running a Local Cook Against a Local Instance

If running a local instance, you can run your local cook against it by passing the option `-UE-CloudDataCacheHost=http://localhost` This assumes that your project has already been set up to use Cloud DDC and that it uses `UE-CloudDataCacheHost=None` as its host override (this can vary between projects).

If this is working as intended, you should see output like this in your cooker:

Console Output

```
DerivedDataCache http://localhost: HTTP DDC: Healthy
```

Copy full snippet

Adding a New Region

The new region will need to contain:

- S3 storage
- Compute (Kubernetes cluster or VMs)
- Scylla deployments

To configure Unreal Cloud DDC for adding a new region, update your cluster sttings on all nodes to include the DNS of the new region.

You should also make sure to set your `LocalKeyspaceReplicationStrategy` for the new region.

For the Scylla configuration, see their documentation on Adding a New Data Center Into an Existing ScyllaDB Cluster.

Specifically, the important part is updating keyspaces

```
ALTER KEYSPACE jupiter WITH replication = { 'class' : 'NetworkTopologyStrategy', '<exiting_dc>' : 3, <new_dc> : 3};
ALTER KEYSPACE system_auth WITH replication = { 'class' : 'NetworkTopologyStrategy', '<exiting_dc>' : 3, <new_dc> : 3};
ALTER KEYSPACE system_distributed WITH replication = { 'class' : 'NetworkTopologyStrategy', '<exiting_dc>' : 3, <new_dc> : 3};
ALTER KEYSPACE system_traces WITH replication = { 'class' : 'NetworkTopologyStrategy', '<exiting_dc>' : 3, <new_dc> : 3};
```

Copy full snippet

We use a replication factor of 3 everywhere, so add the name of the new region (DC).

You will also need to alter the keyspaces of each of the `local` keyspaces to set the replication factor to 0 for the new region (see instructions pertaining to `LocalKeyspaceReplicationStrategy` in this section).

```
ALTER KEYSPACE jupiter_local_regionA WITH replication = { 'class' : 'NetworkTopologyStrategy', 'regionA' : 2, 'regionB' : 0}
ALTER KEYSPACE jupiter_local_regionB WITH replication = { 'class' : 'NetworkTopologyStrategy', 'regionA' : 0, 'regionB' : 2}
```

Copy full snippet

This makes sure that the local keyspace is only written to the local region. While this isn't crucial, this data will only ever be requested within that region and, as such, this saves a lot of bandwidth and storage within the Scylla cluster.

You will also likely want to update your replicators in your Unreal Cloud DDC worker configuration to replicate from this new region.

Setting Up Blob Replication

Unreal Cloud DDC has two methods of replication:

- On-demand replication
- Speculative replication

On-demand replication copies a blob from region A to region B as requests happen in region B that is missing the required blob. This type of replication is opt-in per namespace by setting the `OnDemandReplication` to true in the namespace policy.

> We do not recommend setting this for DDC namespaces as it causes response times to be very variable. For DDC, it's better to accept the cache miss and rebuild the content in that case, but generally rely on the speculative replication to transfer blobs so they are available everywhere without the added latency.

Speculative replication uses a journal kept in each region as refs are added to know which content to replicate. This will follow along as changes happen in a namespace and copy all blobs that are being referenced by these new refs. This will end up copying all content, including content that may never actually be used or needed in the local region, but has the benefit of most often having a local blob available once a ref is being resolved.

For DDC, where response times are quite important to keep low, we recommended relying on the speculative replication.

To set it up, add a section to your worker configuration like this (see `example-values-ABC.yaml`):

example-values-ABC.yaml

```
worker:
config:
Replication:
Enabled: true
Replicators:
- ReplicatorName: DEF-to-ABC-test-namespace
Namespace: test-namespace
ConnectionString: http://url-to-region-DEF.com
```

Copy full snippet

The replicator name can be any string that uniquely identifies this replicator (used to store the state of where that replicator has gotten to, as well as being used in the logging).

`Namespace` is the namespace to replicate.

`ConnectionString` is the URLto use to connect to the other regions of the Unreal Cloud DDC deployment.

This needs to be exposed using the internal ports (see `Networking setup`). You will also need to have credentials set up for Unreal Cloud DDC to use (in the `ServiceCredentials` section), and those credentials will need to have `ReadTransactionLog` access.