

Integrations

We have some proof of concept works for integrating with roll-ups. We are working to prove the system's capabilities and provide a reference implementation for others to follow.

They are being actively developed, so they are in a state of flux.

- [Arbitrum Nitro](#)
- [Optimism](#)
- [Polygon CDK](#)

info Due to protocol limitations, the maximum transaction size is 4 MB.

Integrating your own roll-up

The aim of NEAR DA is to be as modular as possible.

If implementing your own rollup, it should be fairly straightforward, assuming you can use `da-rpc` or `da-rpc-go` (with some complexity here).

All the implementations so far have been different, but the general rules have been:

- find where the sequencer normally posts batch data, for optimism it was the `batcher`
- , for CDK it's the `Sequence Sender`
- and plug the client in.
- find where the sequencer needs commitments posted, for optimism it was the `proposer`
- , and CDK the `synchronizer`
- . Hook the blob reads from the commitment there.

The complexity arises, depending on how pluggable the commitment data is in the contracts. If you can add a field, great! But these waters are mostly uncharted.

If your roll-up does anything additional, feel free to hack, and we can try to reach the goal of NEAR DA being as modular as possible.

Getting started

Makefiles are floating around, but here's a rundown of how to start with NEAR DA.

Prerequisites

Rust, go, cmake & friends should be installed. Please look at `flake.nix#nativeBuildInputs` for a list of required installation items. If you use Nix, you're in luck! Just do `direnv allow` , and you're good to go.

[Ensure you have setup](#) `near-cli` . For the Makefiles to work correctly, you need to have the `near-cli-rs` version of NEAR-CLI. Make sure you setup some keys for your contract, the documentation above should help. You can write these down, or query these from `~/.near-credentials/**` later.

If you didn't clone with submodules, sync them:

`make submodules` Note, there are some semantic differences between `near-cli-rs` and `near-cli-js` . Notably, the keys generated with `near-cli-js` used to have an `account_id` key in the json object. But this is omitted in `near-cli-rs` because it's already in the filename, but some applications require this object. So you may need to add it back in.

If using your own contract

If you're using your own contract, you have to build the contract yourself. And make sure you set the keys.

To build the contract:

`make build-contracts` The contract will now be in `./target/wasm32-unknown-unknown/release/near_da_blob_store.wasm` .

Now to deploy, once you've decided where to deploy to, and have permissions to deploy it. Set `NEAR_CONTRACT` to the address you want to deploy to, and sign with. For advanced users, look at the command and adjust it as needed.

Next up:

`make deploy-contracts` Don't forget to update your `.env` file for `DA_KEY` , `DA_CONTRACT` and `DA_ACCOUNT` for use later.

If the da-rpc-sys

image isn't released yet

We use an FFI library for any go applications that need it, until this is released you've gotta build it locally.

`make da-rpc-docker` This should tag an image that the integrations can use until we eventually publish the package.

Build the `da-rpc-sys` FFI lib:

`make da-rpc` This will ensure you install the prerequisites for local development and output the header files for the go client.

`make da-rpc-docker` This will build a docker image for you, which builds `adylib` for use by the docker images. These automatically require these in the dockerfile when you start the local networks.

If the light client image hasn't been released yet

As part of deploying the devnets, we also deploy the light client.

To build this image, there's a makefile entry for it:

`make light-client-docker`

Deploying Optimism

Configure `./op-stack/optimism/ops-bedrock/.env.example` . This needs copying the `without.example` suffix, adding the keys, contract address, and signer from your NEAR wallet, and should work out of the box for you.

If deploying Optimism on arm64

You can use a docker image to standardize the builds for `da-rpc-sys` and `genesis`.

- `da-rpc-sys-unix`
- This will copy the contents of `da-rpc-sys-docker`
- generated libraries to `theGOPKG/da-rpc`
- folder.
- `op-devnet-genesis-docker`
- This will create a docker image to generate the genesis files
- `op-devnet-genesis`
- This will generate the genesis files in a docker container and push the files in `./devnet`
- folder.
- `make op-devnet-up`
- This should build the docker images and deploy a local devnet for you

Once up, observe the logs

`make op-devnet-da-logs` You should see got data from NEAR and submitting to NEAR

Of course, to stop

`make op-devnet-down` If you just wanna get up and running and have already built the docker images using something like `make bedrock images` , there is a `docker-compose-testnet.yml` in `ops-bedrock`

you can play with.

Deploying Polygon CDK

First, we have to pull the docker image containing the contracts.

make cdk-images why is this different to op-stack ?

When building the contracts `incdk-validium-contracts`, it does a little bit more than build contracts. It creates a local eth devnet, deploys the various components (CDKValidiumDeployer & friends). Then, it generates genesis and posts it to L1 as some arbitrary block. The block number that the L2 genesis gets posted to is non-deterministic. This block is then fed into `thegenesis config incdk-validium-node/tests`. Because of this, we want an out-of-the-box deployment, so using a pre-built docker image for this is incredibly convenient.

It's fairly reasonable that, when scanning for the original genesis, we can just query a bunch of blocks between 0..N for the genesis data. However, this feature doesn't exist yet.

Once the image is downloaded, or advanced users build the image and modify the genesis config for tests, we need to configure an env file again. The envfile example is at `./cdk-stack/cdk-validium-node/.env.example`, and should be updated with the respective variables as above.

Now we can do the following:

cdk-devnet-up This will spawn the devnet and an explorer for each network at `localhost:4000` (L1) and `localhost:4001` (L2).

Run a transaction, check out your contract on NEAR, and verify the commitment with the last 64 bytes of the transaction made to L1.

You'll get some logs that look like:

[illegible]

And if I check the CDKValidium contract `0x0dcd1bf9a1b36ce34237eeafef220932846bcd82` , the root was at the end of the calldata.

```
0x438a539900000000000000000000000000000000000000000000000000000000600000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfff922660000000000000000000000
```

Deploying Arbitrum Nitro

Buildserver/datool :

make target/bin/daserver && make target/bin/datool Deploy your DA contract as above

Updated aserver config to introduce new configuration fields:

"near-aggregator" :

```
{ "enable" :
```

```
true , "key" :
```

```
"ed25519:insert here", "account":
```

```
"helloworld.testnet" , "contract" :
```

```
"your_deployed_da_contract.testnet", "storage":
```

```
{ "enable" :
```

```
true , "data-dir" :
```

"config/near-storage" } } , target/bin/datool client rpc store --url http://localhost:7876 --message "Hello world" --signing-key config/daserverkeys/ecdsa Take the hash, check the output:

target/bin/dataool client rest getbyhash --url http://localhost:7877 --data-hash 0xea7c19deb86746af7e65c131e5040dbd5dcce8ecb3ca326ca467752e729151855Edit this page Last updated on Mar 7, 2024 by Damián Parrino Was this page helpful? Yes No

[Previous RPC Client](#) [Next Arbitrum Nitro](#)