What is Gelato VRF?

Gelato VRF offers real randomness for blockchain applications on Arbitrum by leveraging Drand, a trusted decentralized source for random numbers. With Gelato VRF, developers on Arbitrum get random values that are both genuine and can be checked for authenticity. Explore Gelato VRF's support for all supported networkshere .

Applications of Gelato VRF

The potential applications of a reliable and transparent random number generator on the blockchain are vast. Here are just a few use cases:

- Gaming and Gambling: Determine fair outcomes for online games or decentralized gambling applications.
- Decentralized Finance (DeFi): Use in protocols where random selections, like lottery systems, are required.
- NFT Generation: Randomly generate traits or characteristics for unique digital assets.
- Protocol Decision Making: In protocols where decisions need to be randomized, such as selecting validators or jurors.

How does Gelato VRF work?

Gelato VRF (Verifiable Random Function) provides trustable randomness on EVM-compatible blockchains. Here's a brief overview:

Core Components:

- Drand: Gelato VRF utilizes Drand, a decentralized randomness beacon ensuring unpredictability and unbiased randomness.

Top-level Flow:

- Contract Deployment: Use GelatoVRFConsumerBase.sol as an interface for requesting random numbers.
- Requesting Randomness: Emit the RequestedRandomness event to signal the need for a random number.
- Processing: Web3 functions fetch the random number from Drand.
- Delivery: The fulfillRandomness function delivers the random number to the requesting contract.

Quick start guide

In order to get your VRF up and running with Gelato, you need to make your contract VRF Compatible.

Step 1: Setup your development environment

Ensure you have eitherFoundry orHardhat set up in your development environment.

Step 2: Install the Gelato VRF contracts

- For hardhat users:

npm

install --save-dev @gelatodigital/vrf-contracts * For Foundry users:

forge install gelatodigital/vrf-contracts --no-commit

Step 3: InheritGelatoVRFConsumerBase

in your contract

// SPDX-License-Identifier: MIT pragma

solidity

0.8 .18 ;

import

{ GelatoVRFConsumerBase }

from

"./GelatoVRFConsumerBase.sol" ;

contract

YourContract

is GelatoVRFConsumerBase { // Your contract's code goes here }

Step 4: Request randomness

function

requestRandomness (bytes

memory data)

external

{ require (msg . sender ==

. . .) ; uint64 requestId =

_requestRandomness (data) ; } Step 5: Implement thefulfillRandomness function

function

_fulfillRandomness (bytes32 randomness , uint64 requestId , bytes

memory data ,)

internal override { } }

Step 6: Pass dedicated msg.sender

When you're ready to deploy your Gelato VRF-compatible contract, an important step is to include the dedicatedmsg.sender as a constructor parameter. This ensures your contract is set up to work with the correct operator to fulfill the randomness requests. It's crucial to ensure that only authorized requests are processed.

// SPDX-License-Identifier: MIT pragma

solidity

0.8 .18 ;

import

{ GelatoVRFConsumerBase }

from

"./GelatoVRFConsumerBase.sol" ;

contract

YourContract

is GelatoVRFConsumerBase { constructor (address operator) GelatoVRFConsumerBase (operator)

{ // Additional initializations }

// The rest of your contract code } Once your contract is ready & deployed, grab the address and Deploy your VRF instance !