Why Automata VRF

Automata VRF is a project that generates verifiable random numbers that can be easily integrated into dApps.

Automata VRF may serve as an alternative on a network, which is not supported by Chainlink, for existing projects that have already integrated Chainlink VRF interface in their contracts.

The randomness generation process is executed in a trusted oracle, usesDRAND as a reputable source of entropy, which is then verified by an on-chainAutomataVRFCoordinator contract.

Theoretical example of a dApp that has integrated with Automata VRF contract.

?

Projects may use one of the two approaches below to integrate Automata VRF.

Approach A: Direct integration with Automata VRF Coordinator

This approach is most intuitive for integrating a trusted on chain oracle for smart contracts, as the contract can simply invoke:

- getLatestRandomWords()
- : get at most2**32
- random words at a time.
- getLatestRandomness()
- : fetch the latest verifiable randomness, produced by the off chain oracle. Consumers may use this value as a seed to generate one or more random values using more complex algorithms.
-

Code example of theHundoRando contract using Approach A:

```

Copy // implements a contract that generates a random number from 1 to 100

import{IAutomataVRFCoordinator}from"@automata-network/contracts/vrf/IAutomataVRFCoordinator.sol";

contractHundoRando{ IAutomataVRFCoordinator vrfCoordinator; uint256randomNum;// stores a random number between 1 to 100 uint256currentRound;

constructor(address_vrfCoordinator) { vrfCoordinator=IAutomataVRFCoordinator(_vrfCoordinator); }

functiongetNewRando()public{ uint256newRound=vrfCoordinator.getCurrentRound(); require(newRound>currentRound,"not the latest rando!"); currentRound=newRound;
uint256[]memoryrandomWords=vrfCoordinator.getLatestRandomWords(uint64(1)); randomNum=randomWords[0] %100+1; } }

```

Approach B: Chainlink VRF Interface via theSubscription Method

Code example of theHundoRando contract using Approach B:

```

Copy // implements a contract that generates a random number from 1 to 100

import{VRFConsumerBaseV2}from"@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
import{VRFCoordinatorV2Interface}from"@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";

contractHundoRandoisVRFConsumerBaseV2{ // stores the AutomataVRFCoordinator address here
VRFCoordinatorV2Interface vrfCoordinator;

// stores a random number between 1 to 100 uint256randomNum;

constructor(address_vrfCoordinator)VRFConsumerBaseV2(_vrfCoordinator) {
vrfCoordinator=VRFCoordinatorV2Interface(_vrfCoordinator); }

///@devhandles rawFulfillRandomWords() callback here functionfulfillRandomWords( uint256requestId,
uint256[]memoryrandomWords )internaloverride{ randomNum=randomWords[0] %100+1; }
```

functiongetNewRando()public{ // most projects would receive the requestId here // once integrated/migrated to AutomataVRF, they get the roundId instead // it is totally fine to treat a roundId as its own unique requestId // because the oracle actively submits new randomness within a defined time period // unlike Chainlink VRF, where randomness is only produced when explicitly requested by the consumers // in other words, the requestId is not specially created for any particular consumers uint256requestId=vrfCoordinator.requestRandomWords(bytes32(0),// NOT-APPLICABLE: keyHash uint64(0),// NOT-APPLICABLE: subId uint16(0),// NOT-APPLICABLE: minimum request confirmation uint32(0),// NOT-APPLICABLE: consumers pay gas directly uint32(1)// one random word);

// additional implementation here, usually involves the handling of requestIds } }

```

Interface that integrators use

?

```

Copy abstractcontractVRFConsumerBaseV2{ errorOnlyCoordinatorCanFulfill(addresshave,addresswant); addressprivateimmutablevrfCoordinator;

/ *@param_vrfCoordinator address of VRFCoordinator contract*/ constructor(address_vrfCoordinator) { vrfCoordinator=_vrfCoordinator; }

/ *@noticefulfillRandomness handles the VRF response. Your contract must*@noticeimplement it. See "SECURITY CONSIDERATIONS" above for important *@noticeprinciples to keep in mind when implementing your fulfillRandomness* @noticemethod. * *@devVRFConsumerBaseV2 expects its subcontracts to have a method with this*@devsignature, and will call it once it has verified the proof *@devassociated with the randomness. (It is triggered via a call to* @devrawFulfillRandomness, below.) * *@paramrequestId The Id initially returned by requestRandomness* @paramrandomWords the VRF output expanded to the requested number of words */ functionfulfillRandomWords(uint256requestId,uint256[]memoryrandomWords)internalvirtual;

// rawFulfillRandomness is called by VRFCoordinator when it receives a valid VRF // proof. rawFulfillRandomness then calls fulfillRandomness, after validating // the origin of the call functionrawFulfillRandomWords(uint256requestId,uint256[]memoryrandomWords)external{ if(msg.sender!=vrfCoordinator) { revertOnlyCoordinatorCanFulfill(msg.sender,vrfCoordinator); } fulfillRandomWords(requestId,randomWords); } }

```

NOTE: If your project integrates Chainlink VRF via thedirect funding method, then you may not be able to directly integrate Automata VRF, unless you deploy your ownVRFV2Wrapper contract that points toAutomataVRFCoordinator .

Previous Verifiable Random Function Next How does Automata VRF work Last updated25 days ago On this page Was this helpful?
```