➡

Writing Solidity Functions

1. Understand the Role of a Checker

A Checker acts as a bridge between conditions and smart contract executions. Its purpose? To check conditions and determine whether a task should be executed by Gelato. Every Checker returns two main things:

- canExec (Boolean)
- : Indicates if Gelato should execute the task.
- execData (Bytes)
- : Contains the data that executors will use during execution.
-

Solidity functions must adhere to the block gas limit for checker calls; exceeding it will cause the call to fail.

1. Solidity Function Example

Before we delve into complexities, let's understand the structure of a simple Checker:

```
Copy contractCounterChecker{ ICounterpublicimmutablecounter;

constructor(ICounter_counter) { counter=_counter; }

functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=counter.lastExecuted();

canExec=(block.timestamp-lastExecuted)>180;

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); } }
```

In the above, the checker checks the state of a counter and prompts Gelato to execute if 3 minutes (180 seconds) have elapsed since its last execution.

1. Making your Checker Reusable

Avoid hardcoding addresses. Instead, allow the passing of arguments to your checker. This lets you reuse the checker for multiple tasks:

```
Copy functionchecker(address_counter) external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=ICounter(_counter).lastExecuted();

canExec=(block.timestamp-lastExecuted)>180;

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); }
```

1. Advanced: Checking Multiple Functions

Suppose you're automating tasks across different pools. Instead of creating multiple tasks, iterate through your list of pools within a single checker:

```
Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) { uint256delay=harvester.delay();

for(uint256i=0; i<vaults.length(); i++) { IVault vault=IVault(getVault(i));

canExec=block.timestamp>=vault.lastDistribution().add(delay);

execPayload=abi.encodeWithSelector( IHarvester.harvestVault.selector, address(vault) );

if(canExec)return(true,execPayload); }
return(false,bytes("No vaults to harvest")); }
```

1. Incorporating Feedback with Logs

With the Gelato Web3 Functions UI, you can use custom return messages to pinpoint where your checker might be "stuck":

```
Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=counter.lastExecuted();

if(block.timestamp-lastExecuted<180)return(false,bytes("Time not elapsed"));

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); return(true,execPayload); }
```

1. Limit the Gas Price of your execution

On networks such as Ethereum, gas will get expensive at certain times. If what you are automating is not time-sensitive and don't mind having your transaction mined at a later point, you can limit the gas price used in your execution in your checker.

```
Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) { // condition here

if(tx.gasprice>80gwei)return(false,bytes("Gas price too high")); }
```

This way, Gelato will not execute your transaction if the gas price is higher than 80 GWEI.

Previous Test, Deploy & Run Typescript functions Next Test, Deploy & Run Solidity Functions Last updated4 months ago On this page * 1. Understand the Role of a Checker * 2. Solidity Function Example * 3. Making your Checker Reusable * 4. Advanced: Checking Multiple Functions * 5. Incorporating Feedback with Logs * 6. Limit the Gas Price of your execution